University of St Andrews

Academic Journal Team 15

Tim Beatham, Ben Bicknell, Iva Stratieva, Eeshaan Sarda, Natania Sophia

Submitted 31/03/2022

Abstract *(190005340)*

As part of CS3099, teams of four or five students were to work collaboratively on a software engineering project, both within and between teams. The project's overall objective was to create a federation of academic journals where code is published and peer-reviewed, each team creating one journal from the federation. Our team created a full-fledged journal that meets all basic requirements (e.g. posting submissions and comment threads under them), and greatly expands upon them (e.g. versioning of submissions, notifications and announcements). Our federation also meets requirements of allowing mutual logins and transfer of submissions.

Declaration

Table of contents

Introduction *(190005340)*

As a part of this module, we set out to create a web-based academic journal that publishes peer-reviewed code, and the accompanying online platform to review the software and supporting documents. Targeted at publishing research software, it facilitates code review by making it possible to attach comment threads to places in the code. This system was developed because existing code review systems such as GitHub do not adapt well to the requirements of reviewing in the context of a journal.

The basic functionality of the academic journal involves users being able to upload and comment on submissions, reviewers being able to review and assign verdicts to pending submissions, and editors being able to publish submissions. Another important part of the project is that the journal does not stand alone, but is instead part of a federation. Each journal was associated with a super group which allows a user with an account to log into any other journal in the super group (or federation), as well as to migrate submissions to other journals within the super group where needed.

In addition, we set out to provide additional features such as private discussions, versioning, supporting documents and rendering Jupyter documents to name a few. Indeed, when designing the academic journal we wanted to make the journal as much of a sociable experience as possible with the ability to comment on submissions and have private discussions with users of the system. Our system allows authors to add versions and assign co-authors to their submission. Administrators can view reports filed against users (for misconduct), ban such individuals and delete submissions from the journal if they breach the guidelines. The journal provides the ability for admins to make announcements that are then propagated to the rest of the users of the system.

Another aspect of this project was learning to work collaboratively and professionally as software engineers. With regards to this goal, we have been keeping up with a Scrum system of collaboration on this project. This means work was distributed over sprints of 2 weeks during which the most relevant next steps are pulled from a backlog of wanted features, and progress is evaluated at the end. Through Scrum, we kept to Agile development, where the goal is to iterate over and continually evaluate and improve upon the product, instead of following a fixed plan set at the beginning.

By the first major release we had addressed all of the outlined above with additional features identified by the product owner. Thus, the team has had success in implementing the expected features of the journal, even expanding upon it. The detail of this implementation is laid out in the following. While polishing off little details might be useful to reach the standard of a real academic journal website, the goals that we set out for our product were reached, as our system solves the problem of adapting code review to peer-reviewing.

Project details

We implemented all of the required basic features of the project, all of the suggested extensions and more extensions of our own.

The extensions we implemented:

- Dashboard with customisable user-specific widgets
- Customisable public and private user profiles (with biography, social media links etc)
- Email verification
- Resetting a forgotten password via email
- Tiered role system with users, reviewers (specific to a submission), editors and administrators
- Report system (where users can report another user for a violation of the rules)
- Ban system (where admins can ban users as a result of a report or other reasons)
- Real time notifications (for events such as a user’s submission being published)
- Real time announcements that are propagated globally
- Private discussions between multiple users
- Displaying images, PDFs and markdown files
- Custom Jupyter document viewer
- Adding supporting documents to a submission
- Pulling submissions from GitHub
- Migrating submissions to/from other journals
- Versioning of submissions
- Assigning multiple co-authors to a submission
- Stylized comments with support for markdown and embedding images
- Comment threads (nested comments replying to different users)
- Additional asymmetric encryption of requests containing sensitive information

Note that this list is not exhaustive as many of the other features we implemented were not required and could technically also be considered extensions.

Scrum

Throughout the course of the project we have been following the scrum methodology with the goal of refining our internal processes. We initially set the goal of rotating the responsibility of each team member each week. We constantly refined our sprint processes by addressing what we think went well (continue), what we need to watch out for in further sprints (stop) and finally what we think we could introduce to improve our sprint processes (start). From this we identified that rotating some roles proved too much of an overhead. Additionally we add the responsibility of assigning features to individuals the responsibility of the scrum master.

We defined the roles of *scrum master, product owner, tester, hci analyst* and *the supergroup representative.*

Standup

Throughout the course of the project we met up three times a week. The purpose of which would be to discuss features each individual had been working on and to identify if any from team members needs providing.

We formatted the meetings by asking around in a circle what they did since the last meeting and what they plan to do in the next meeting. Initially the use of this was limited but as the project developed and the first release date got nearer these meetings proved invaluable.

Throughout the year we refined this further by discussing what each group member related to the supergroup protocol. This stimulated discussion and ensured that we could identify in the supergroup protocol.

Sprint Planning

Sprints last two weeks in length at the start of a sprint we would meet in person to plan it. At the start of each sprint we would take features from the product backlog and assign them to the current sprint, splitting them into further cards in order to plan how we would approach each card. We identified an acceptance criteria in accordance with the product owner and tester and estimated the amount of time in hours it would take to implement. We used GitLab to control and coordinate each sprint.

Throughout the sprint we refined this meeting so that the sprint board would be already set up and description added by the scrum master so that we could concentrate on communicating how we would implement each card.

Sprint Review

At the end of the sprint we evaluated our progress and how we should refine our processes. This was in person and the bulk of the time was spent evaluating each point in start, stop and continue (Morris, 2017, 137). The useful feature of this approach was that it promoted transparency and openness. It would give change for everyone to air their views and kept morale up by concentrating on both the positive and negative aspects of the sprint.

Product Backlog

Our product backlog was developed and maintained in a shared spreadsheet. Each week we would identify features and estimate how much they would cost to implement in terms of a weighted fibonacci system. This was online and we used planning poker to independently provide estimates on each item. Throughout the week the product owner would add items to the backlog and order features in terms of importance.

By the end of the project we had a large backlog consisting of bugs, technical debt and feature improvements. If we were to continue we would carry on where we would be able to carry on where we left off and prepare for the next iteration.

Interaction Flow

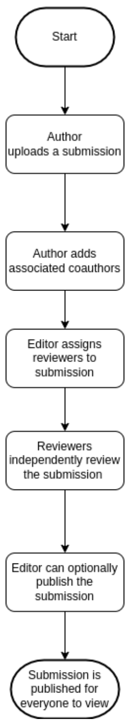

Start
Author uploads a submission
Author adds associated coauthors
Editor assigns reviewers to submission
Reviewers independently review the submission
Editor can optionally publish the submission
Submission is published for everyone to view

Figure 1 - The process of reviewing a submission

The above figure shows the process a submission goes through in the academic journal. An author starts out by uploading a submission and adding associated co-authors to the submission. From that point on an editor will be able to add reviewers to each independently review the submission until each editor has made a verdict. When all editors have made a verdict an editor has the ability to publish a submission for the public to view. When initially embarking on the project this is what we ultimately set out to achieve.

Backend Design

TypeStack *(190007422)*

We built our backend with TypeScript and Express. Initially we were using plain Express and validating every request body/params separately, however we converted to using TypeStack, a series of TypeScript packages for Express.

An example is '*routing-controllers*', which greatly simplified our controllers, as the syntax to create a new endpoint and specify required roles, request bodies, parameters etc was much more succinct. TypeStack also provided '*class-validator*', which allowed us to create classes to model request bodies or parameters and annotate the parameters with decorators (e.g. whether a field was required, was of a certain length etc) and it would perform validation on the provided input, automatically returning an error with the relevant validation issue if the validation failed. This also aided us greatly, allowing us to concentrate on the data and logic rather than the intricacies of validation.

Additionally, it provided powerful dependency injection facilities, allowing us to easily share services/resources across the entire backend project. An example of this is our email service, which was used by multiple different controllers to send automated emails such as an email asking a user to verify their email address and one for a user to reset their password if they have forgotten it. We made sure to define an interface for each of our services, and inject them into each controller (that uses them), which allows us to easily unit test our controllers by mocking dependencies.

The power of the set of libraries can be seen in our framework's ability to incorporate websockets into the site. We were able to create a service for managing rooms a user is registered to and inject them into the necessary controllers without fundamentally changing the business logic.

Database (*180003863)*

We use MongoDB as the database for our backend. We chose it as it is fast, scalable and has very good package support in most languages, such as Mongoose for NodeJS/TypeScript which we make use of. Mongoose is an ODM library that allows us to define our database schemas, validate these schemas and also attach helper methods onto each model object.

In some cases we made use of 'hooks' to modify/update certain fields in models before/after they are saved. One such example is where we utilised the pre save hook on the user model to hash the user password before it is saved to the database.

A diagram of the database models:

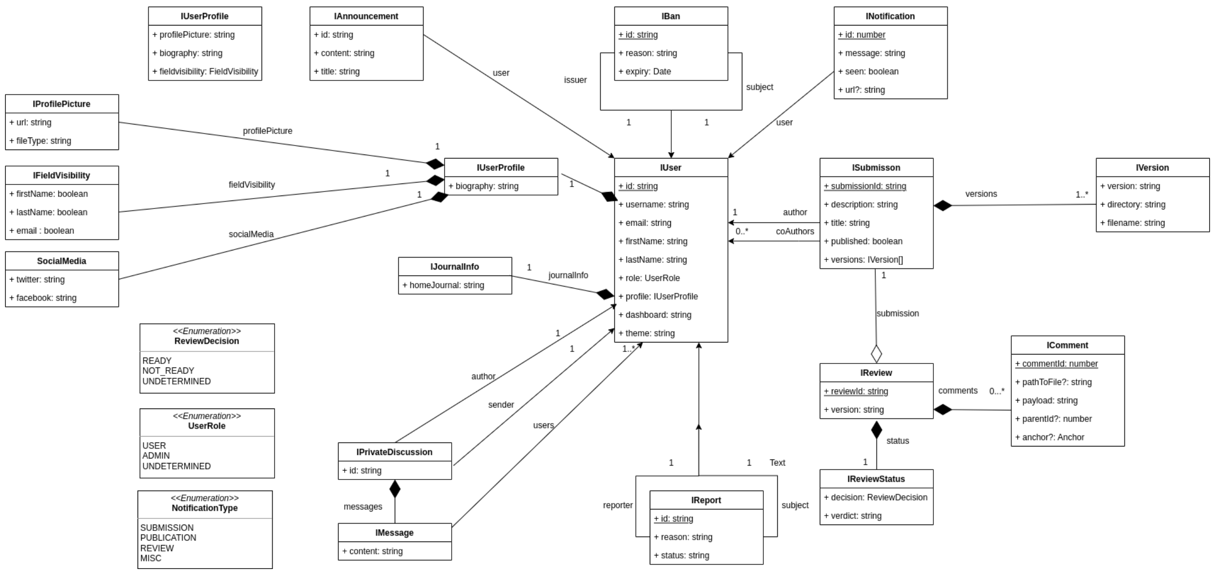


Figure 2.

MVC *(190007422)*

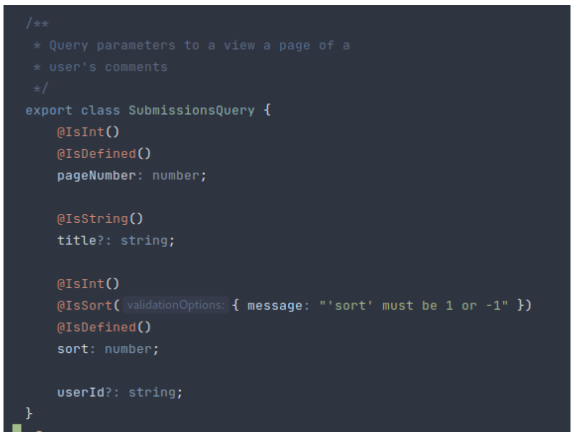
We have followed the MVC design pattern (Smith, 2022) as closely as possible. Interaction between the mongoose database was interfaced through repositories, which handles basic CRUD operations and means that our controllers can focus on flow of control rather than interaction with the database.

Each controller has one or models injected into them so that the controller focuses on the business logic and what is returned to the user. We adopted this methodology into our solution to make it easier to test. We could unit test controllers by stubbing dependencies allowing us to test the flow of control of each controller - additionally we could test our models by setting up an in memory database and testing that the documents were created and manipulated correctly.

Of course we had separate projects for the frontend and backend, so we could not follow this pattern exactly as we have no explicit ‘View’ part of the acronym.

Request Validation

For request validation we used the ‘*class-validator’* library from TypeStack which allows us to add decorators to request bodies and queries in order to validate the value and type. This, along with ‘*routing-controllers*’ described earlier, means that we can easily change the validation as the need arises. The below example shows our validation for searching for submissions in the system.



```
/**
 * Query parameters to a view a page of a
 * user's comments
 */
export class SubmissionsQuery {
    @IsInt()
    @IsDefined()
    pageNumber: number;

    @IsString()
    title?: string;

    @IsInt()
    @IsSort( validationOptions: { message: "'sort' must be 1 or -1" })
    @IsDefined()
    sort: number;

    userId?: string;
}
```

Figure 3.

As shown in *Figure 3* we have an *‘IsSort’* validator to ensure that the sort can only take up a discrete set of values.
When a user makes a request the instance of the *‘SubmissionsQuery’* object is automatically created and validated. Thus allowing us to focus on business logic. We often found we would fall into the trap of adding a decorator and presuming that the endpoint was correctly validated. To prevent this trap we wrote integration tests which covered the validation of these properties to ensure that validation was still correctly tested.

Middleware *(190007422)*

Authorization

We created our own middleware for user authorization. JWT session cookies are passed on every request and are automatically validated before each request to the backend. We verify that the cookie is a valid JWT and the user has the correct role. If all these conditions hold then we inject a `*SessionUser’* object into the request. Otherwise we clear the cookie and throw an error (which is handled as an unauthorised request).

We specified that an endpoint was authorised with the authorised decorator. This allowed us to specify which endpoints should and should not be validated. The decorator takes the role as a parameter which allows us to restrict access to roles.

Error Handling

When an error is thrown in our controller it is automatically handled by our middleware. We get the reason from the instance of the error and the status code from the error and automatically send the response to the user in a readable fashion.

The responses from our API all follow a clear pattern, returning JSON data with a status (either ‘success’ or ‘failure’), and any other according data. If the status is ‘failure’ then the response also contains the reason for the error. This maintains a consistent structure, and makes unit testing easier as we can treat them as normal methods.

Logging

If we are interested in the request body that is passed into a method that we can log the requests with the logging middleware. Unlike our other middlewares this is not a global middleware and mainly used for debugging during development.

We currently did not feel the need to store persistent log files, in the future we would of course add this or use a remote solution such as DataDog.

Session Cookies *(190007422)*

Our session cookies are a signed JWT (IETF et al., 2015). They store details about the authorised users using the system. We store the user’s id, email and username and role. We need the user’s ID to identify the user but the rest of the values are mainly used for ease of development. We store the role in the JWT so that we can automatically validate the user has the required role on each request and separate this logic from the controller itself.

Although anyone can view the payload of a JWT (as it is simply base64 encoded), we verify the signature of the JWT before trusting (or using) the data within it, so this does not open us up to a vulnerability.

Users

Each user has a unique ID which is suffixed with our team identifier (t15) to make it unique across the super group. We decided to separate the user’s name into first name and last name so that we have the ability to easily address them by first name in some cases. (For example: welcome Jesse, instead of Jesse Marsch). Of course there is the case that some users do not have a last name or that some user’s may have more than one surname therefore there is very little validation that we do for these fields. They are simply used to address the user, and can be changed by the user at any time.

We represent user roles with an enum (of integers of a power of 2), and store an integer for each user’s role(s). We chose to do this instead of using a string as it allows us to represent a user having multiple roles without having to store a list of their roles (and can check whether a user has a given role via a bitwise operation).

The user’s profile is separated into a subdocument which also includes the field visibility of the information about them.

Because this can also store the profile of an SSO user it means that we make the combination of the team number and email property unique as two users can potentially have accounts on two different systems.

Submissions

We do not separate submissions and publications into separate models. We decided instead that a submission has a field which indicates whether or not a submission has been published to the journal or not. We do so as a publication is really just the same as a submission and thus it does not benefit from storing it in a separate model. Each version has one or more related versions.

Each version has to have a unique identifier within a submission. The version subdocument consists of the version identifier, the directory in which the version resides in and the fileName of the version it stores. When a user makes a review it is a review of the latest versions.

Reviews

A review is stored as a separate document to a submission. We made this decision as a submission can be deleted from the system but there should still be a way to record the review that took place and thus we store it in a separate document.

Each review has zero or more comments associated with it. The comments are stored in a subdocument. The comment contains the ID of the parent to that comment, thus we store comments in a tree structure. Where each comment can have one or more children in which users have replied to that comment and thus creating threads of comments.

Each comment may have an anchor which specifies the start and end location of the source file in which the comment should be placed. This allows us to support commenting on a single line of code, as well as large chunks of code across multiple lines.

Each comment can optionally have a path to a file associated with it specifying which file the comment was made. If no file is specified it is treated as a general comment on the submission as a whole, otherwise it is associated with the given file.

Reports

A user can report a submission; these reports are stored in the report model. Each report has a unique ID and an associated id of the subject being reported and the user being reported. Only admins can view the reports and this contains information for an admin to locate the user and ban them should they see fit.

Bans

A user is banned for a period of time which is recorded through the bans model. It has the associated id of the subject being banned and the issuer who banned the user. An admin has the ability to set an expiry of a ban so that a ban can expire after a given length of time and the user can use the system.

SuperGroup *(190007422)*

SSO

In our system we have the concept of an SSO user. This is a user from another journal who has logged onto/been exported into our journal. We decided to store these in our user collection in our database so that they have a profile on our system. This means they can view their posts on our system, change their profile picture and even have roles such as admin on our website. We made this decision to actually store the user so that we do not have to treat users and SSO users differently. Although this of course potentially leads to data integrity issues if they change their details on the other journal, suggesting we should have a supergroup endpoint to update user details. However as of yet this has not been implemented within our supergroup.

The process of logging into our journal from another journal is described in the SSO documentation but there are a few design decisions that we made. One design decision we made was that we generate a signed JWT to authenticate the communication with the other journal, and verify that the other journal has authorised this sign in attempt. We choose to use a JWT as it is stateless and allows us to verify the integrity of both the sign in attempt and our communications, without having to store anything in the database.

Exporting / Importing Submissions

The process of exporting and importing a submission is specified in the super group protocol. However there are some key design decisions that we made in order to achieve export and import of submissions.

The process of importing a submission is handled through the below diagram.

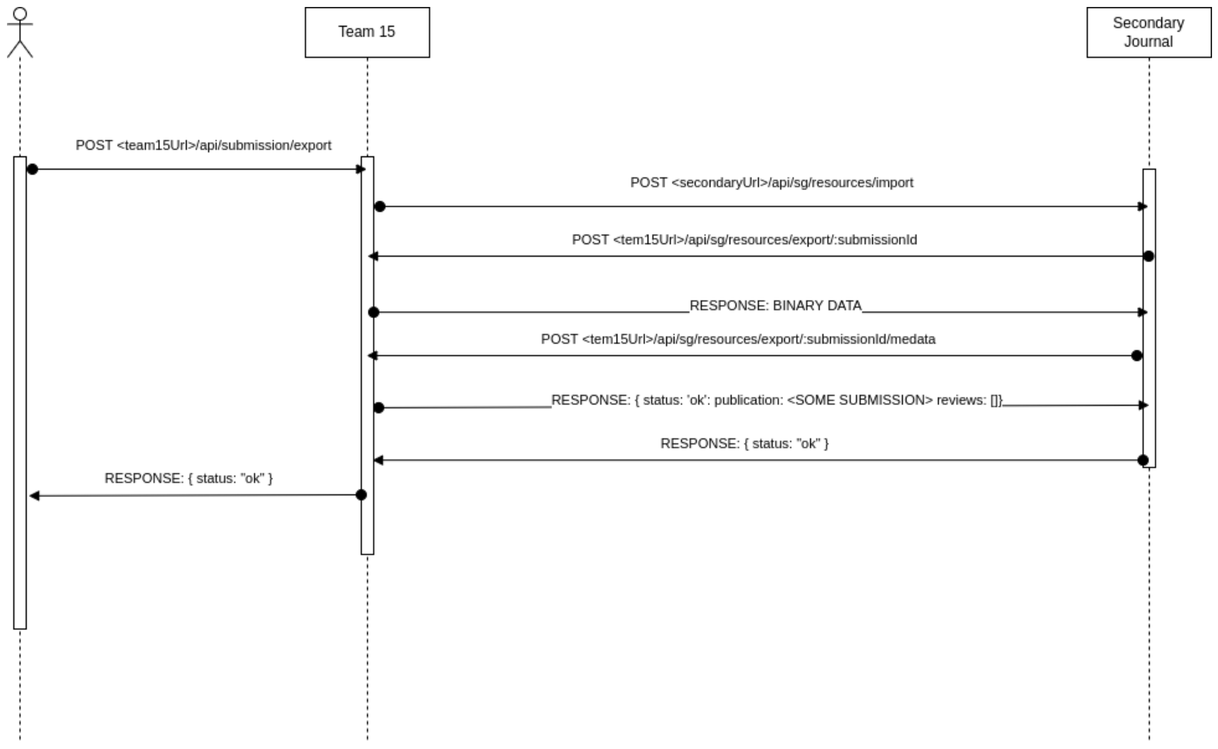

Team 15
Secondary Journal
POST <team15Url>/api/submission/export
POST <secondaryUrl>/api/sg/resources/import
POST <tem15Url>/api/sg/resources/export/:submissionId
RESPONSE: BINARY DATA
POST <tem15Url>/api/sg/resources/export/:submissionId/medata
RESPONSE: { status: 'ok': publication: <SOME SUBMISSION> reviews: []}
RESPONSE: { status: "ok" }
RESPONSE: { status: "ok" }

Figure 4.

Another journal requests that we want to export a submission to our journal. We thus generate a secure token which is passed in the header of each request to validate both the identity of the journal and the validity of the journal.

When importing a submission we get the submission's metadata which includes the ID of each user. Each ID we create is unique within the super group as we append the team number suffix to each ID. Therefore from this we are able to identify the team that made the request and thus make a request to the public profile of that individual. Each time we check if the author is in the database and if it is not we import the user into our database. We have a policy of all or nothing, meaning that if even one of the IDs are not valid we choose to not import the submission into our system.

When importing a submission we make use of transactions which are achieved through a replica dataset in mongodb. This ensures that if importing reviews fails but the submission exceeds we discard the submission. It is important to do this as it could lead to data consistency issues if the request is not valid.

Services *(180003863)*

Our backend API has various different ‘services’, that are injected into controllers via dependency injection. Examples of this include the email service, ban service, decryption service, notification service and more.

The email service allows us to send automated emails to users for various purposes. The most common one is email verification. Originally we were sending emails via SMTP (using the NodeMailer library), however unfortunately SMTP appeared to not be supported on the host servers. As a result we migrated to using SendGrid (an automated email service that has a free version supporting up to 100 emails per day). This worked nicely and didn’t require many changes as the API was designed to be similar to sending emails with NodeMailer.

The ban service has the role of unbanning users after the duration of their ban is over. In order to do this we run a background worker to check for and delete all expired bans every 5 minutes. To allow us to check every 5 minutes we made use of a local Redis server and the Bull npm package to run cron tasks at intervals. This allowed us to schedule tasks in the future while avoiding large setInterval or setTimeout calls.

Another service which made use of the Redis queue was the notification service. We send notifications to users via sockets to inform them of events that have occurred involving them, e.g. they are assigned as a reviewer/co-author on a submission, added to a private discussion or their submission has been published. As there often could be a lot of notifications to be dispatched at once, instead of creating, saving and dispatching all of them immediately, we add them to the Redis queue. The queue then processes each notification, saves it to the database and sends it to the user (via the socket service). This improves the performance of our site, speeds up the API calls and takes the load off the REST API.

Another interesting use of the background jobs was our publications service, which resets the featured publications and the publication of the day each day at 2am.

Static Files *(190007422)*

Submissions

We submit in their zip format. A user can either upload a zip or a single file, if they choose to upload a single file, place the file into a zip and store it on the server. We did this as it reduces the overhead of storing a whole directory on the server. Whenever a user wants to retrieve a file we read the zip file, find the path they are referring to and extract it in memory. Although less efficient than storing the directory itself it reduces the overhead of archiving it when exporting the submission. It also somewhat protects us from the user trying to access a path outside of the submission directory as the zip acts as a container for the submission.

Versions

We treat all versions as a submission in its own right. Therefore versions are automatically given an id and stored along with submissions. They are not handled any differently. A submission must have at least one version (the initial submission to upload). Therefore we store versions as a zip and treat them like a submission.

Supporting Documents

Supporting documents are stored on the server separately from the submission that they are related to. They are automatically given a uuid and are stored in the file system of the server. We could have arguably made the decision to use gridfs to store them in the mongoose database itself. This would have the benefit it would be easier to delete and ensure consistency between the submission and the data. However we decided against this as we wanted to prioritise performance and storing it in the file system offers greater performance.

Profile Pictures

We store each profile picture in the file system. We expect that a user uploads either a png or a jpg. A profile picture is named to give the id of the user in the profile picture. Once again this is stored in the file system. There is the argument to store the profile picture in the database to ensure consistency between a profile picture and a user however the performance hit of storing this in the server seemed too great at the time. We had considered using *GridFs.* However, one benefit to using *GridFs* is that it would benefit from redundancy redundancy mechanisms offered by our replica set improving data integrity. If after release the system gets more use this is something we would consider.

Websockets *(180003863)*

While the main component of our backend is a REST API, there are some situations where two way communication is required. We wanted to avoid simply polling our REST API (as it is inefficient and typically provides slower updates), so we utilised websockets, specifically Socket.IO, to provide bidirectional communication between the client and the server.

Examples of this include private messages, notifications and announcements. When a user receives a notification (e.g. if their submission was published) or an announcement is published, the server informs the relevant user(s) and they receive the updates in real time. In both these cases the server is communicating one-way with client(s), the websockets simply bring the benefit of real time updates.

A use case of communication both ways is our private discussions. These discussion ‘rooms’ can contain any number of users (added by the host), who can all communicate with each other in real time. Users are also added to/from the discussion room using websockets so the clients don’t need to refresh their page to see these changes. We implemented this using the rooms feature of Socket.IO, with each room identified by the discussion ID.

Another interesting application of our socket system was to clear all the sessions for a given user. When a major change is made to a user account, such as changing the password or role, or they are banned, our backend server pushes a message to all sessions logged in as that user to log them out immediately. This makes the website more responsive, and is important for security reasons as well (so a banned user or a user signed in with an older password cannot continue to use the site for a short period of time).

Production *(190007422)*

Podman

We used docker to deploy our solution to the CS host servers. We did so as it automated and made deploying the production solution easier. It meant that our config values were not dependent on the URL but they were instead dependent on the virtual machine which meant we could treat the machine as we would our local machine. Hence we can run the latest mongodb solution without need for installation each deployment, and we can write bash scripts to automatically build and deploy the production image. In some cases we had to change values (typically config values such as URLs) depending on whether the solution was running locally or on the host servers. To achieve this we made use of webpack plugins to detect the runtime environment and make the required changes before compilation.

At the end of each sprint we would build the production solution so that other journals could test their system against ours. We had initially planned to include a build number on the home page of our solution so to aid ourselves/others when testing our website. However since our development was of a continuous nature and not working towards a set number of releases (with the MVP and the final product being our only real concrete releases), the build number was somewhat arbitrary and became hard to keep track of, and so was removed.

Documentation (180003863)

All functions and methods of our backend are documented using TypeScript doc comments. This allowed us to utilise TypeDoc to generate documentation (in the form of an interactive webpage). This can be found in the Docs folder (entrypoint is index.html). Note that the tests are not included, as they aren't relevant to the API documentation and each test is also named and described clearly.

We also have Postman API documentation for the important endpoints, this can be found in the Appendices section.

Frontend Design

Pages *(190007422)*

We separated our react project into pages and components. The pages of the solution consist of the different url paths it is possible to reach. These mainly consist of root state that is passed down to the individual components on that given page. This made testing more refined as each component should be, in theory, individually unit tested and thus the only unit tests that were required were those testing that they were in the given page.

The management of pages and their endpoints is handled through the react router dom library. This library allows us to give the appearance of separate pages in a single page application via manipulating the DOM. This was a tradeoff as it resulted in a slight increase in loading time of the website for the first request, but increased the speed of all further page/frame loads. This made navigating the site a lot smoother and improved the user experience.

Components *(190007422)*

Components consisted of units of a page that were not a page in their own right. Through the use of adopting the SOLID methodology we aimed to decompose everything into a component so that they could be reused and tested independently . This made our design much more robust as it meant that to implement a new feature we could reused components previously specified and thus reduce developer time and effort as well as maintaining consistency of the page.

There are many good examples of this across the site, such as the pagination buttons (used on most lists/tables), user metadata components (e.g. profile picture) and components for viewing specific file types (e.g. pdf, markdown etc).

Slices

We stored the global state in slices. The main slice we used was the user slice which stored information about the current session user. This was used in the react component to identify whether or not a user was validated and verified. We had the concept of an authenticated page which wrapped around a react route. This allowed us to redirect a user to a specific page should they try to access an endpoint which is authorised.

Context (180003863)

We made use of React context to share global application data (global as in the same for every component/page, not every user session) across different components. The main example of this is the authentication context, which was used in the root of every page. It provided common authorisation related methods, such as getting user details from the backend, or logging in/registering.

Another context we created was the socket context, which provided all pages/components the ability to listen to and emit messages via websockets to/from the backend. This allowed us to only create one socket connection (per page load) and reuse it across the application, rather than having every individual component connect (e.g. notifications, announcements, private messages) and communicate separately, thus improving the performance of our site.

Validation

We used *Formik* to validate user input from a form and we separated this validation into a set of methods to validate each possible input value a user can input. This means if the validation requirements change we can easily change the validation logic without it affecting any other logic.

Our validation provides instantaneous feedback to user input. This allows a user to be aware of any errors that may have occurred from their input, in some cases informing them of errors before they have submitted the form.

All forms within our website have this validation applied to them, but a specific example is our password fields, which all reject weak passwords (the password requirements are discussed in the Security section) and display an error message informing the user of these requirements.

Services *(190007422)*

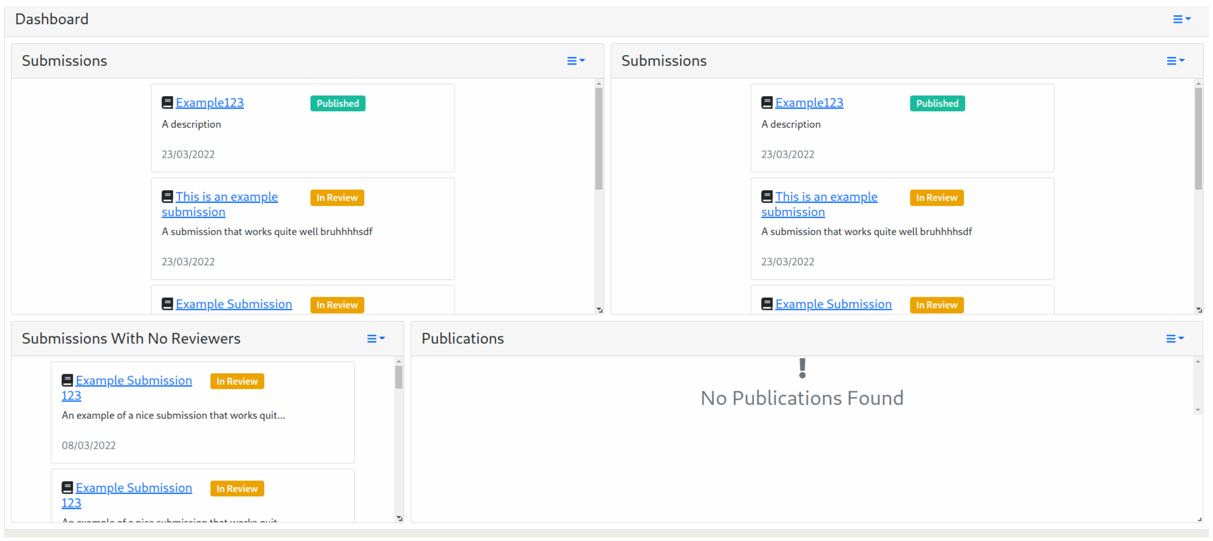
Each API request made from the Frontend was done so through interfacing out each API request into interfaces called *'services'* - a concept inspired by AngularJs (*AngularJS: Developer Guide: Services*, n.d.). This aided the testing of each component but also helped to make the system more maintainable as it meant services could be injected into a component.

With each service it is possible to state the axios object to use when making a request. This meant we were able to make a global error handler that causes an error prompt to appear if an error was returned from an API. It also means that when we extend the API we simply have to create a new method and any component which utilises the service can make use of it.

UI *(190005340)*

Overall, the user interface (UI) of our journal was well thought through and designed with the intention of making the website as easy and intuitive to use as possible. Consistency was kept in the layouts of all pages, through the use of elements such as the top bar for logged in users, containing logo (allows quick access to dashboard), notifications and profile of the user. This layout is among the most widespread for navigating a website, giving users an anchor at any moment. Keeping in line with this is deliberate and important, as it helps users to become familiar with the website more easily, allowing them to explore it freely. Colours were chosen so as to ensure the best possible accessibility of the website, as they were tested for sufficient contrast and on font sizes, using the Stark tool.

Dashboard



Dashboard

Submissions

Example123 Published
A description
23/03/2022

This is an example submission In Review
A submission that works quite well bruhhhhsdf
23/03/2022

Example Submission In Review

Submissions

Example123 Published
A description
23/03/2022

This is an example submission In Review
A submission that works quite well bruhhhhsdf
23/03/2022

Example Submission In Review

Submissions With No Reviewers

Example Submission 123 In Review
An example of a nice submission that works quit...
08/03/2022

Example Submission 123 In Review

Publications

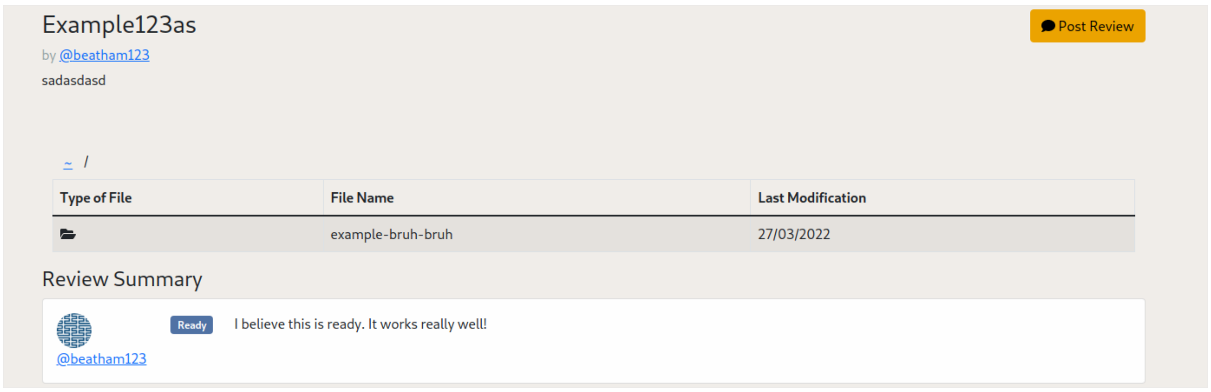
No Publications Found

Figure 5.

The dashboard is the first thing a user sees when they log in, making it essential to create a good experience. It provides basic management allowing users to navigate and administrate the journal all from one place. In our system, it is an adaptive page, meaning it differs meaningfully depending on which type of user sees it. A regular user has quick access to the latest publications and submissions, as well as their own submissions. A reviewer has additional access to their reviews in progress and assigned to them. An editor can see the latest reviewed submissions, and a panel to assign reviewers to new submissions. An administrator has access to reports and can issue bans from the dashboard.

In addition to this, each individual user has the ability to personalise the layout of their dashboard to make it best fit their needs. Namely, the dashboard is organised in widgets that can be dragged and dropped into the desired place, and adapted in size. The chosen layout is saved for future logins. Thus, users have easy access to the features mentioned above depending on their role, but can adapt how they are organised for them. This means that users interact with the website more and “invest” their time to make a layout, which creates more value and attachment on their part. Ultimately, this works to increase user retention for the journal, along with making their experience consistent and most intuitive to them.

Reviews


Example123as

by @beatham123

sadasdasd

~ /

Type of File	File Name	Last Modification
	example-bruh-bruh	27/03/2022

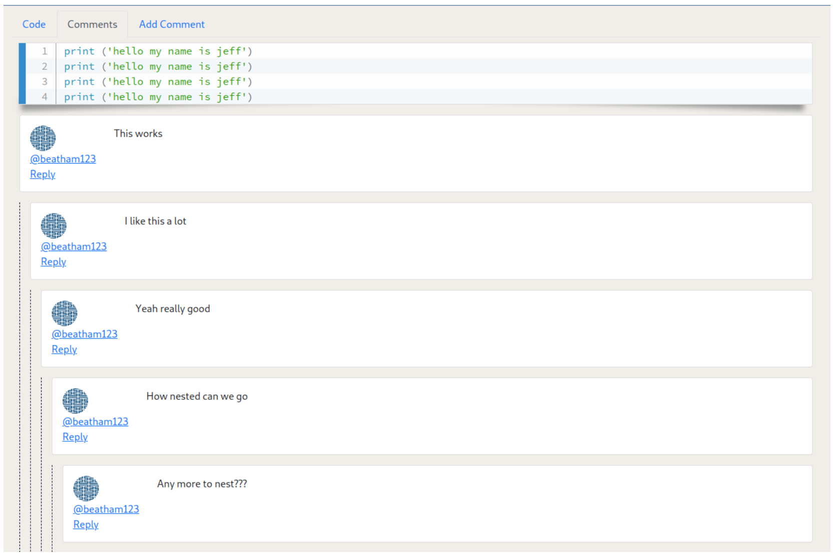
Review Summary

@beatham123 Ready I believe this is ready. It works really well!

Figure 6.

The reviewing process starts with editors assigning reviewers to a newly posted submission. This can be done from their dashboard. Reviewers can then write reviews that show up as comments under the submission, as pictured above. In order to keep objectivity, one’s reviews are hidden from other reviewers until they have all made their decision. Only once all decisions are in, reviewers can view each other’s comments and respond to them. The reviews also include a bright box that indicates at a glance what the verdict was. These features allow for discussion, and ensure the editor can make a coherent and informed decision on publication.

Comment Threads



Code | Comments | Add Comment

```
print ('hello my name is jeff')
print ('hello my name is jeff')
print ('hello my name is jeff')
print ('hello my name is jeff')
```

This works
@beatham123
Reply

I like this a lot
@beatham123
Reply

Yeah really good
@beatham123
Reply

How nested can we go
@beatham123
Reply

Any more to nest???
@beatham123
Reply

Figure 7.

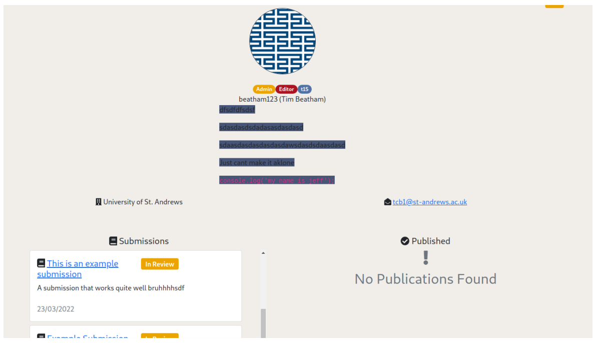
Comments can be seen under a separate tab for quick access. Both comments on the submission in general and comments on specific parts of the code show up here. When code lines are commented on, they appear above the comment that targets them. This makes it easier to replace comments in their context without additional efforts or clicks on the user's part. Replying to a comment creates a thread of replies, making discussion easier and encouraging conversation about a submission. Comments on the same level of the thread are aligned by a dotted line to make it easier for the user to keep track of the conversation. After a certain depth comments are automatically hidden so that readers can concentrate on the subject of the conversation.

Profile

A user profile consists of their avatar, their role(s) and the team in which they belong to. SSO users can create an account on the system and therefore we tell the user which group they are a part of.

User's have a biography which is formatted as HTML. This provides a way of telling other users who they are and what they do.

We identified that it is important for user's to view other user's publications and submissions. Therefore we display this as a list of submissions and publications. These infinitely scroll and fetch each user's publication upon reaching the end of the list.



Admin Editor t15
beatham123 (Tim Beatham)

dfsdfdfsdsf

sdasdasdsdadasasdasdasd

sdaasdasdasdasdasdawsdasdsdaasdasd

Just cant make it aklone

`console.log('my name is jeff');`

University of St. Andrews

tcb1@st-andrews.ac.uk

Submissions

This is an example submission — In Review
A submission that works quite well bruhhhhsdf
23/03/2022

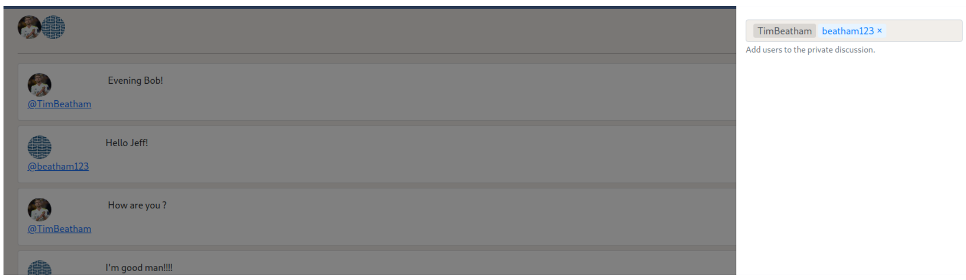
Published

No Publications Found

Figure 8.

We identified that it was important to other user information and not just their email. A user can specify to display their social media such as their twitter, github and linkedin accounts. This provides further ways to user's to verify the identity of another user. Privacy was an important issue in terms of the user interface therefore a user can control fields to be set public and private. User's can decide whether or not to make their email firstname, last name and institution public or private. To us privacy was important as it should be up to the user which information they want other user's to access. The use of a button to toggle public and private information makes it clear which fields are public and clear how to change this immediately.

Private Discussions



@TimBeatham Evening Bob!

@beatham123 Hello Jeff!

@TimBeatham How are you ?

I'm good man!!!

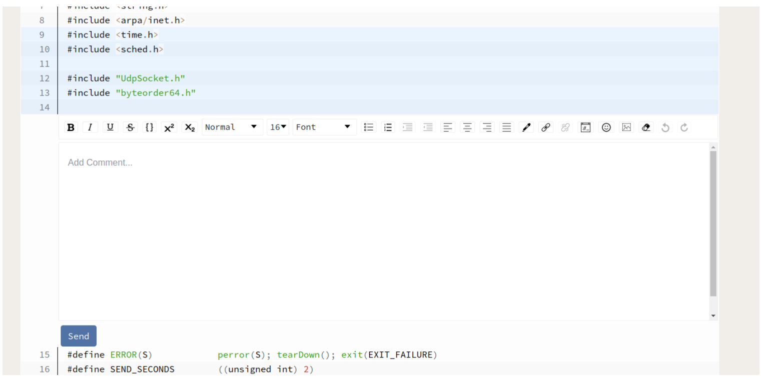
TimBeatham beatham123 ×

Add users to the private discussion.

Figure 9.

Private discussions can consist of one or more users and provide real-time updates of the conversation so that user's get immediate feedback. We decided to make it so that only the host can add or remove users from the private discussion and we use our user selector to do this. We decided that private discussions should not be multithreaded to make it clear exactly when a user has replied.

Commenting On Files



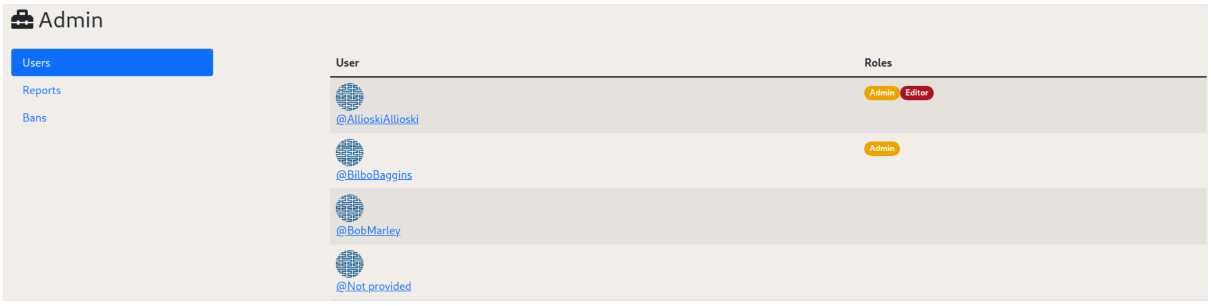
```
#include <arpa/inet.h>
#include <time.h>
#include <sched.h>

#include "UdpSocket.h"
#include "byteorder64.h"

```

```
#define ERROR(S)            perror(S); tearDown(); exit(EXIT_FAILURE)
#define SEND_SECONDS        ((unsigned int) 2)
```

Figure 10.

Our code commenter components provide the ability for users to comment on a file. The commenter is inline so that it is easy to identify the part of the code that is being commented on. When a user makes a comment they are automatically redirected to the new thread corresponding to the code location. We made the design decision to display comments with code and not inline to the code to highlight the specific area that was being commented on and reduce any ambiguities in terms of what is being commented on.

Admin System


Admin

Users
Reports
Bans

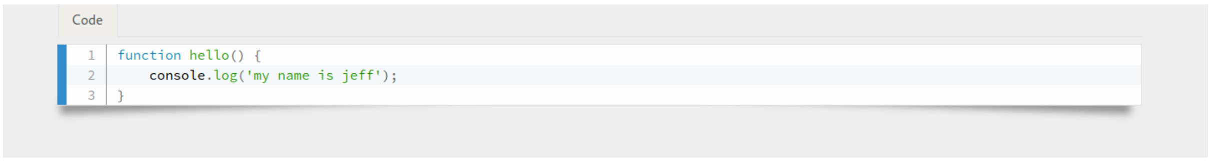
User	Roles
@AllioskiAllioski	Admin Editor
@BilboBaggins	Admin
@BobMarley	
@Not provided	

Figure 11.

The admin system allows user's ban users, respond to reports, assign roles to an individual and view the bans in the system that haven't expired. We present users in a table so that the admin can easily see the information it needs in order to locate the correct user, and perform an action on that individual user. The tabs separate the different functions that are possible in the admin system keeping the information logically separated.

Viewing Submissions (180003863)

Our website supports uploading both zip files and single files as the content of a submission. If a zip file is uploaded then the directories and files within can be traversed from the submission's page. When a single file is viewed, we render it in different ways depending on the file type. Special file types we have custom support for are images, PDFs, Markdown files and Jupyter notebooks. If the file is some other file type then we display the contents with syntax highlighting applied if it is a code file of some language. This way we can support all file types, including raw files with unknown extensions and also have support for common file types.


Code

```
function hello() {
    console.log('my name is jeff');
}
```

Figure 12.

As can be seen, the syntax is highlighted with appropriate colours. We use PrismJS to provide the syntax highlighting, and support a wide variety of programming languages.

When an image file is viewed, we render the image (rather than displaying the raw source of the image like the default approach would).

If the file is a PDF, we render the PDF in the user's browser. They can scroll through and view the document as though it were a normal PDF:

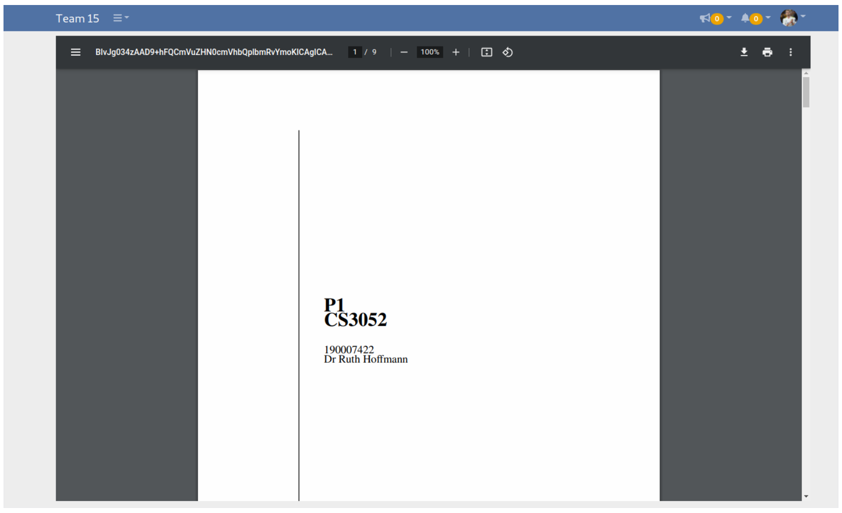
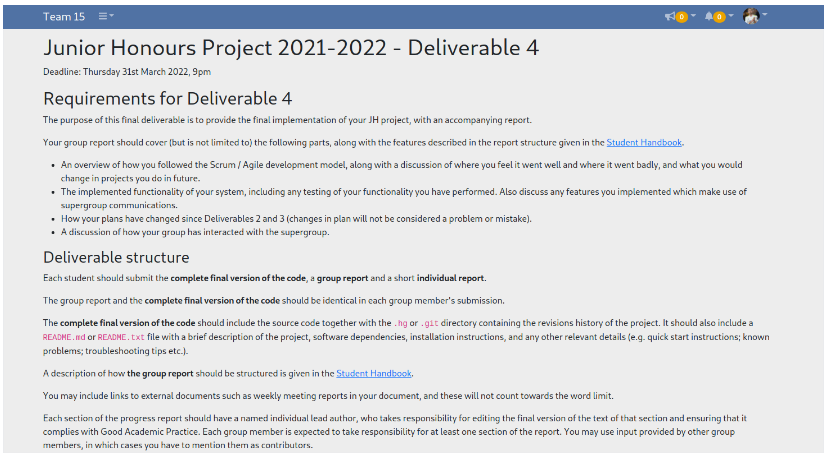


Figure 13.

Similarly if the file is a markdown file, we display it with markdown support (i.e. styled headings, bullet points etc):



Junior Honours Project 2021-2022 - Deliverable 4

Deadline: Thursday 31st March 2022, 9pm

Requirements for Deliverable 4

The purpose of this final deliverable is to provide the final implementation of your JH project, with an accompanying report.

Your group report should cover (but is not limited to) the following parts, along with the features described in the report structure given in the Student Handbook.

- An overview of how you followed the Scrum / Agile development model, along with a discussion of where you feel it went well and where it went badly, and what you would change in projects you do in future.
- The implemented functionality of your system, including any testing of your functionality you have performed. Also discuss any features you implemented which make use of supergroup communications.
- How your plans have changed since Deliverables 2 and 3 (changes in plan will not be considered a problem or mistake).
- A discussion of how your group has interacted with the supergroup.

Deliverable structure

Each student should submit the **complete final version of the code**, a **group report** and a short **individual report**.

The group report and the **complete final version of the code** should be identical in each group member's submission.

The **complete final version of the code** should include the source code together with the `.hg` or `.git` directory containing the revisions history of the project. It should also include a `README.md` or `README.txt` file with a brief description of the project, software dependencies, installation instructions, and any other relevant details (e.g. quick start instructions; known problems; troubleshooting tips etc.).

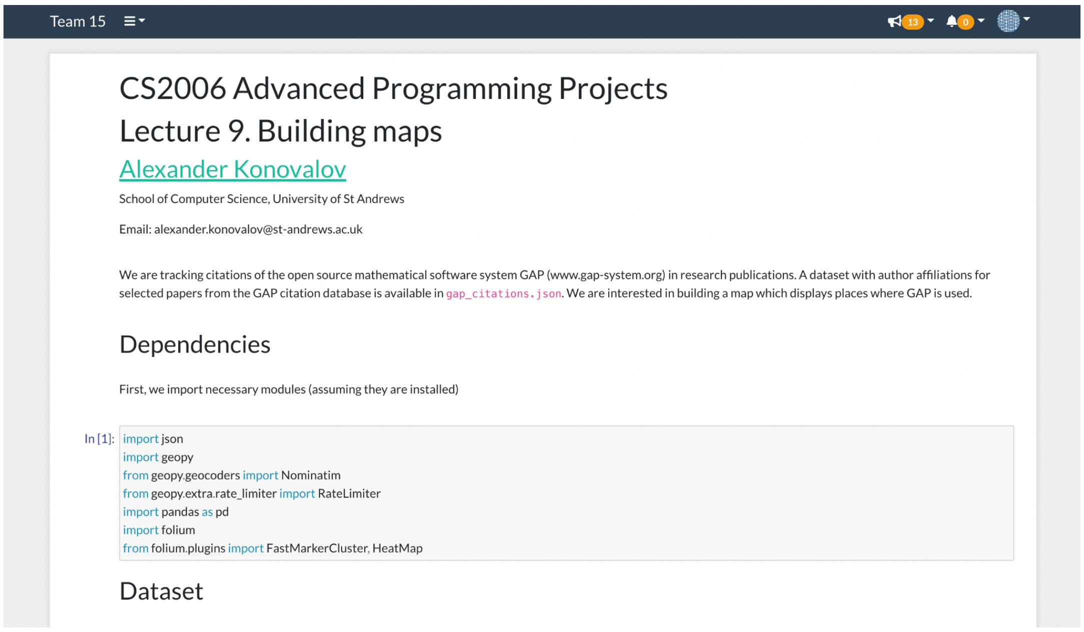
A description of how **the group report** should be structured is given in the Student Handbook.

You may include links to external documents such as weekly meeting reports in your document, and these will not count towards the word limit.

Each section of the progress report should have a named individual lead author, who takes responsibility for editing the final version of the text of that section and ensuring that it complies with Good Academic Practice. Each group member is expected to take responsibility for at least one section of the report. You may use input provided by other group members, in which cases you have to mention them as contributors.

Figure 14.

We also support viewing Jupyter Documents. When we planned on supporting them we hoped to make use of an existing library for displaying them in the browser, however we found all the existing libraries we tried had issues or didn't look like a real Jupyter document. As a result we parsed the Jupyter document file ourselves, and built a document viewer ourselves, aiming to make it appear as close to a normal Jupyter document as possible.


Team 15

CS2006 Advanced Programming Projects
Lecture 9. Building maps
Alexander Konovalov

School of Computer Science, University of St Andrews

Email: alexander.konovalov@st-andrews.ac.uk

We are tracking citations of the open source mathematical software system GAP (www.gap-system.org) in research publications. A dataset with author affiliations for selected papers from the GAP citation database is available in `gap_citations.json`. We are interested in building a map which displays places where GAP is used.

Dependencies

First, we import necessary modules (assuming they are installed)

```
In [1]: import json
        import geopy
        from geopy.geocoders import Nominatim
        from geopy.extra.rate_limiter import RateLimiter
        import pandas as pd
        import folium
        from folium.plugins import FastMarkerCluster, HeatMap
```

Dataset

Figure 15.

As can be seen, the Python code has syntax highlighting applied. We also support markdown syntax within Jupyter documents. Any output or tracebacks from the result of cells in the document (before uploading the submission) are also displayed.

Security *(180003863)*

As our project is a web application that deals with sensitive user details (such as emails and passwords), security is of the utmost importance. This section will discuss the information we store, how it is kept safe and the further measures we have taken to protect our users' details in the event of an attack/data breach (worst case scenario).

User information we store

To login to our journal, a user account is required. Users can sign up for an account via the register page, where they are asked to provide their email, a strong password, a username and their first and last names. The criteria for a strong password is discussed in the *Steps we have taken* subsection.

Users also have a profile where they can provide additional information: a profile picture, the institution they belong to and a biography. Their profile also contains their email and first and last names. They can control which information is publicly visible, this is discussed more in the *Steps we have taken* subsection.

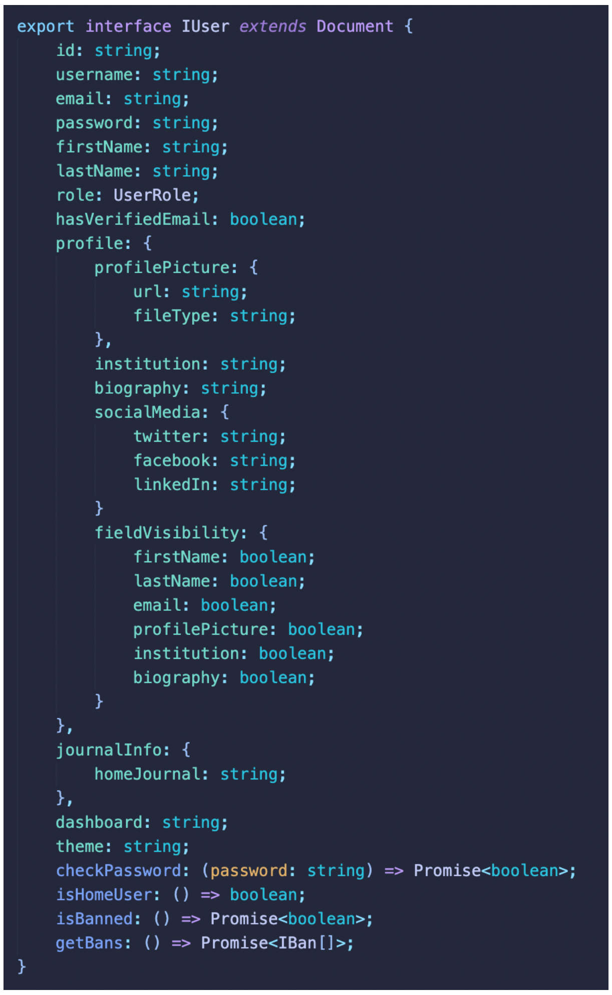
In addition to the aforementioned information, we also store some additional data on users: their role within the website (i.e. user, editor, admin), whether they are banned, their dashboard widget preferences and the URL of their home journal (if their home journal is not our website).

When a user makes a publicly visible interaction with our site, such as leaving a comment, reviewing a submission or sending a private message/announcement, a reference to their user account (their user id) is stored to acknowledge this. Their username and profile picture will then be visible on their comment/review/message.

Our website has the functionality for users to report other users. When a report is submitted we store the reason/content of the report, and the ID of both the reporter (user who submitted the report) and the subject (user who is being reported).
Administrator users have the ability to take action upon reports and ban users (or ban them for other reasons not related to a report). Similar to a report, a stored ban contains the ID of the issuer (the admin user who issued the ban) and the subject of the ban.

How we store user information

User accounts are stored in the database, MongoDB. This database is of course private, and only connected to by our backend servers. Users are stored according to the user “model”, which is essentially an interface that describes how the data should be stored. This can be seen:



```typescript
export interface IUser extends Document {
    id: string;
    username: string;
    email: string;
    password: string;
    firstName: string;
    lastName: string;
    role: UserRole;
    hasVerifiedEmail: boolean;
    profile: {
        profilePicture: {
            url: string;
            fileType: string;
        },
        institution: string;
        biography: string;
        socialMedia: {
            twitter: string;
            facebook: string;
            linkedIn: string;
        }
        fieldVisibility: {
            firstName: boolean;
            lastName: boolean;
            email: boolean;
            profilePicture: boolean;
            institution: boolean;
            biography: boolean;
        }
    },
    journalInfo: {
        homeJournal: string;
    },
    dashboard: string;
    theme: string;
    checkPassword: (password: string) => Promise<boolean>;
    isHomeUser: () => boolean;
    isBanned: () => Promise<boolean>;
    getBans: () => Promise<IBan[]>;
}
```

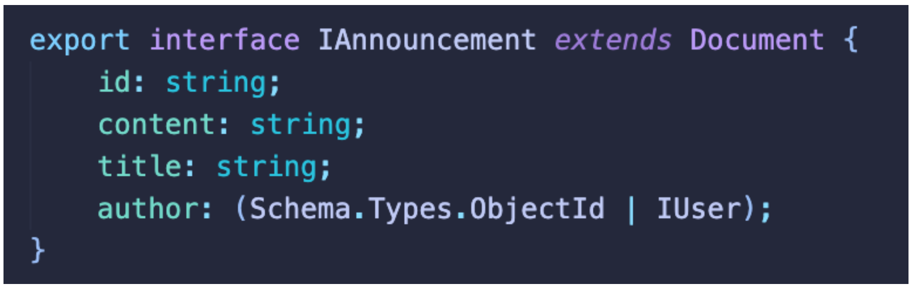
Figure 16.

Before being stored in the database, passwords are hashed using bcrypt, which is a password hashing algorithm known to be secure and is currently unbroken. Hashing passwords rather than storing them plaintext ensures that they are protected even in the event of a data breach (or at least as protected as they can be). We perform some post-processing on the password hash before saving it in the database, this is discussed in the *Steps we have taken* subsection.

When information regarding a user account is required from a backend API endpoint (for displaying some data on the frontend for example), it is pulled from the database and returned. The endpoints that return user information (such as those related to user details, their profile etc) take care to only return information that the user has marked as private (via the profile options mentioned before) are only returned when the user making the request is themself.

Some information is stored locally on the user's browser, in the form of a session cookie. This information is all non-sensitive (such as their user id) and allows us to persist users across different pages. To ensure the integrity of this information, it is stored in a JWT and the claims are verified before being trusted.
The session cookie itself is marked as secure and HTTP only. Secure means that it is only sent over secure communication protocols (i.e. HTTPS rather than HTTP). HTTP only means the cookie is not accessible to client-side scripts (but still sent to the server when making requests). This helps to mitigate the risk of XSS attacks and prevent session hijacking, as even if someone were to inject a malicious script into our website they wouldn't be able to steal session cookies.

As mentioned before, when a user leaves a comment/review/message, their user ID is stored within the appropriate database model (e.g. the comment model). This allows us to cross-reference who created the post and display some information regarding them, such as their username and profile picture. This information is not stored within the secondary database model, instead the user id is used to retrieve their user model and access the information from there.
An example of such a model, the announcement model:



```
export interface IAnnouncement extends Document {
    id: string;
    content: string;
    title: string;
    author: (Schema.Types.ObjectId | IUser);
}
```

Figure 17.

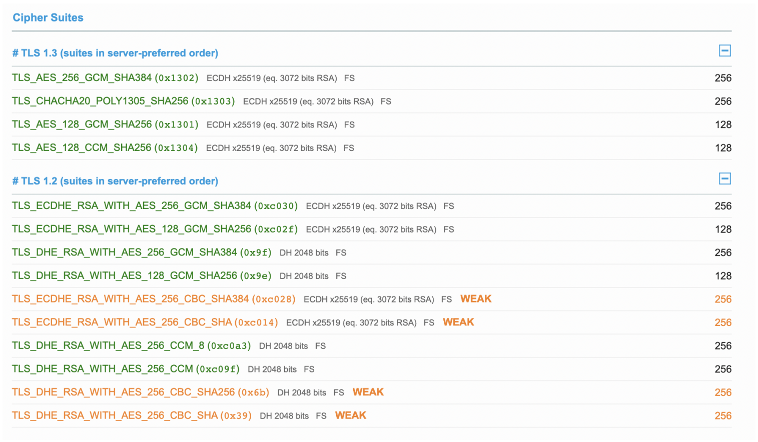
Similarly, the report and ban models contain the IDs of the reporter/issuer and the subject of the report/ban.

Steps we have taken

Since our website is a publicly accessible web app, there are a variety of different potential security threats to be aware of. As a result we have taken many important steps to prevent any potential attackers from exploiting our website, and to protect our users' information in the worst case scenario where an attack was successful.

Our website is accessible via HTTPS, which is a secure communication protocol that uses the Transport Layer Security (TLS) encryption protocol to ensure secure communication of data. TLS uses asymmetric encryption where the client (in this case a web browser) encrypts the data using the server's public key and the server decrypts it using their private key. This prevents any naive man in the middle (MITM) attacks, as an outside attacker who has managed to intercept traffic is unable to decrypt and steal any sensitive details.

A common concern when using HTTPS is the use of insecure cipher suites (a cipher suite is simply a set of algorithms/protocols that indicate how the communication should be encrypted/transmitted), that open a website up to potential exploits. As our website is served on the school’s host servers it is out of our control what cipher suites our website supports, to check this I ran a test using SSLLabs (*SSL Server Test (Qualys SSL Labs*, n.d.). The supported cipher suites can be seen in *Figure 18.*



```
Cipher Suites

# TLS 1.3 (suites in server-preferred order)
TLS_AES_256_GCM_SHA384 (0x1302)   ECDH x25519 (eq. 3072 bits RSA)   FS                      256
TLS_CHACHA20_POLY1305_SHA256 (0x1303)   ECDH x25519 (eq. 3072 bits RSA)   FS                256
TLS_AES_128_GCM_SHA256 (0x1301)   ECDH x25519 (eq. 3072 bits RSA)   FS                      128
TLS_AES_128_CCM_SHA256 (0x1304)   ECDH x25519 (eq. 3072 bits RSA)   FS                      128

# TLS 1.2 (suites in server-preferred order)
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)   ECDH x25519 (eq. 3072 bits RSA)   FS       256
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)   ECDH x25519 (eq. 3072 bits RSA)   FS       128
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x9f)   DH 2048 bits   FS                              256
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x9e)   DH 2048 bits   FS                              128
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)   ECDH x25519 (eq. 3072 bits RSA)   FS   WEAK 256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)   ECDH x25519 (eq. 3072 bits RSA)   FS   WEAK    256
TLS_DHE_RSA_WITH_AES_256_CCM_8 (0xc0a3)   DH 2048 bits   FS                                 256
TLS_DHE_RSA_WITH_AES_256_CCM (0xc09f)   DH 2048 bits   FS                                   256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x6b)   DH 2048 bits   FS   WEAK                       256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x39)   DH 2048 bits   FS   WEAK                          256
```

Figure 18.

As can be seen, several weak cipher suites for TLS 1.2 are supported. However these are typically supported for backwards compatibility with older browsers, and since the SSL report ranked the website good overall, they shouldn’t be a major cause for concern.
(Muscat, 2019)

One of the main things a user of a website may be concerned about is their privacy. As a result we have designed our website to allow privacy customisation as much as possible. For every field on the profile users can select whether it is private or public. If it is private, only they can view it and it will not be displayed on their public profile (which is visible to other users of the site) or accessible via API endpoints to anyone except their user account. As mentioned before, every API endpoint that handles user details takes care to respect the user’s private/public decision for every piece of information.

Another major concern a user may have is if their sensitive data is stored securely. As mentioned before, passwords are hashed using the bcrypt hashing algorithm. In the event of a data breach where an attacker has got access to all data stored, this prevents them from viewing the passwords in plain text, and then using them for nefarious purposes (hence protecting the user).

Password hashing is only effective when performed on a strong password, as brute forcing a weak password is still a possibility. Therefore our website requires users to provide a strong password, and won’t let them sign up without one. A password is deemed to be strong if it is at least 8 characters long and contains at least one each of the following: lowercase letter, uppercase letter, number. These criteria ensure brute forcing a password hash within a reasonable time period isn’t possible.

However hashing passwords isn’t complete protection, as although an attacker isn’t able to decode or brute force a strong password hash within a reasonable time period, what is still a possibility is comparing a hash with the hash of another known password. If they match then the attacker knows what the original password was. As a result we made sure to salt our hashes (adding random data to the password before hashing) so attackers cannot compare with pre-computed hashes.

Another possible method of attack is actually hashing another existing password and checking if the two hashes match (and hence they know what your password was). Typically attackers will leverage huge lists of known passwords in order to “crack” hashed passwords (which is why using common passwords is a bad idea, and hence why our website advises users against this).

There isn’t much that can be done to fully protect against this (aside from giving users password advice), however we have taken some steps to make it harder for an attacker to do so. In order for an attacker to be able to compare with an existing password, they need to know which hashing algorithm was used. After hashing a password, we perform a simple XOR cipher on the string then pad it with some random characters, before then saving it in the database. This means that if an attacker gets access to our data they will not be able to tell which hashing algorithm was used by inspection of the hashes (as the transformed strings will not look like typical hashes of any algorithm) or be able to hash known passwords and compare without knowledge of the additional cipher and padding added. Of course this is only an extra step if the attacker also has access to our codebase in addition to the data, but still worth implementing. Note that while similar to hash salting, this is different as it is a post-hashing transformation.

One of the most relevant forms of attacks to a web application nowadays is cross site scripting (XSS). This is where an attacker manages to inject their script into your website and can hence steal/exploit sensitive information. Typically there are three forms: reflected, stored and DOM-based.

Reflected XSS occurs when an attacker includes a malicious script in a URL (typically as a query parameter) that is then executed by the website. To address this we reviewed our use of query parameters and request body fields, and ensured that none of them can be treated as HTML elements or executed in any other way.

Stored XSS occurs when an attacker manages to include a script in data that is stored in the database, which will then be served back to other users of the application. Our backend API already performs strict validation on all of the data it accepts (i.e. data types, length etc), but we also took extra steps to sanitise any values that are stored in the database.

DOM-based XSS occurs when an attacker injects a script into the HTML DOM of a webpage that is then executed. Since once of the most common ways of this occurring is the use of document.write (and similar DOM methods) we made sure we don’t utilise them, and also ensured that we don’t unsafely render any untrusted HTML. In the cases where we do render HTML, such as any field that makes use of our rich text component (user biography, comments, private messages etc), we take care to sanitise the HTML content before displaying it.
(*What Is Cross-Site Scripting (XSS) and How to Prevent It? | Web Security Academy*, n.d.)

Another relevant threat is man in the middle attacks (MITM). These occur when a third party manages to intercept (and possibly modify) communications between two computers that should only be communicating directly with each other. In our case this would typically be the website traffic that would be intercepted. As mentioned before, our website uses HTTPS so even if an attacker gained access to a user’s raw outgoing network traffic, they wouldn’t be able to decrypt it.

However decrypting the traffic is possible if the attacker is able to pretend to be the certificate authority (CA), which can be achieved by getting the user to trust a certificate on their device. The attacker can then provide the user with their own public key for encrypting traffic, intercept and decrypt it (with their own private key) then encrypt it with the website's real public key and send it off to them (so the communications still function as normal).

To address this threat and protect users, we use asymmetric RSA encryption on all of the endpoints that include passwords (the login, register, change and reset password endpoints). The client (our frontend web page) encrypts the request payload with the website's public key (which is included with the frontend scripts of the website as it is public), and sends the encrypted payload with the request rather than the raw payload containing the password. Our server then decrypts the payload using our private key, which is stored securely.

The user's password is then protected even if an attacker is able to decrypt the HTTPS traffic. The password is now only ever stored in plain text in the DOM and the memory of the browser, so the only avenue for an attacker to gain access to it (at least those in the scope of our website) would be via exploiting the HTML page (i.e. XSS which was discussed before).

Another benefit of the additional asymmetric encryption is that passwords won't be stored in plain text in log files or dumps from network debuggers that are decrypting the HTTPS traffic, such as Fiddler or Wireshark.
(*Man in the Middle Attack: Tutorial & Examples*, n.d.)

I already discussed injection attacks in the context of XSS, however it's worth mentioning that injection attacks can also be used to pull data from databases or even delete the whole database. Our website uses MongoDB and although NoSQL injection attacks are not as common as SQL injection, they are still possible.

Typically an attacker provides a value in a request that is then queried in the database, a user ID for example. If the value is not sanitised (and is not queried securely) they can run their own queries and control what is returned from that endpoint (or perform other malicious actions).

In the context of our backend, an Express server with MongoDB, the area for concern is dealing with query objects, as an attacker would aim to provide their own object or add extra queries to an existing one. We ensured that we validate and sanitise all user input that is queried (particularly making sure that no input can be passed as an object), and ensured that we create/build the query objects yourself (and only include input from the client as strings/numbers within the query).
(Belmer's & Belmer, 2019)

Evaluation *(190007422)*

Evaluation Against The User Requirements

At the start of the year when undertaking the project we outlined an initial specification of criteria that we had hoped to achieve. Whilst we have accomplished the majority of the points in the specification there are a few points that need outlining.

User Registration And Login

Initially we set out the goal of user's being able to register an account and login to the system including authentication processes to verify the user. Our solution in the first release covers these points. User's can register to the system using their email address, the system asks the user to verify their email which helps to identify the individual.

Despite this there are a few issues which our first release does not cover. We set out the goal of creating an OAuth mechanism so that user's can conveniently log in through GitHub / Google. We did not end up implementing this as we did not identify it as a necessity. However, it would have added massive business value and is something that should be considered should development continue as it seems to be an industry standard. Additionally there is no way for a user to specify whether or not they would like to stay logged into the site, it defaults so that they are automatically logged into the site which may lead to privacy issues if they forget to sign out.

Roles

Our initial goal was to create a role based system where we could restrict specific functionality to a given role, where the role system would represent that of an academic journal. We believe that we have accomplished this basic requirement. There are two roles in the system: an administrator role and an editor role. Only administrators can assign and revoke roles.

However one could argue that this could be significantly improved. Having two roles of admin and editor seems minimal and it may have been beneficial to implement more roles or preferably allow administrators to specify new roles and specify the permissions user's of this given role have.

SSO

We specified that users within our SuperGroup should be able to log into our system using their journal credentials. We have arguably achieved this goal. SSO users can log into our journal securely, their details are imported into our journal (including their profile picture) and they are treated as any other user in our journal. Roles can be assigned to them, they can create and review submissions. They can even change their details in our journal should the imported information be accurate.

If there was one thing to be improved with regards to this goal it would be the ability to import more data from each journal. It may be useful to display submissions that users have opened in other journals which would further highlight the importance of the supergroup.

Viewing Code
The goal of viewing code related to viewing formatted code within a submission. We set out the goal that code should have syntax highlighting. Using the library PrismJS we believe we have met this goal. We offer support for syntax highlighting various different languages.

One improvement that could be made to this would be that we could have provided a mechanism for viewing the text differences between versions within a submission. This would allow editors and reviewers to see the improvements made to a given submission.

Reviewing Submissions
We set the goal of allowing assigned reviewers to review submissions and that each review should be independent of one another. We have accomplished these outlined goals. Reviewers can add threads of comments to a given submission and they can give a status of a submission and each review is hidden until a decision has been made. At the start we identified that this was an important feature to consider as it reflects the review process in real life.

There are a few areas where the review process could be improved. There is no support for editorial boards meaning that a single editor can decide whether or not submission should be published, there is no support for an editor to decline publishing a submission, they simply don't publish / delete it which is arguably restraining.

Export / Importing Submissions
The initial specification required that journals within the supergroup should be able to export and import submissions to one another. We accomplished this: we are able to export a submission to other groups in the supergroup and also import submissions. Reviews are retained when doing so and their corresponding threads as well as their corresponding author.

There are a few issues however which in future releases may need to be addressed: there is no endpoint specified for updating a submission if it is updated on another journal meaning it is possible for submissions to be inconsistent. On our end we could have utilised a separate event queue for exporting submissions to improve the performance of exporting large submissions, there is no way to export the corresponding supporting documents and status' of a review meaning that a reviewer has to re-make a decision on the other journal.

Social Networking
Initially we specified there should be basic social networking implemented. User's should be able to communicate with each other through private messages and there should be the ability to follow individuals so that they users can keep up to date with one another.

Currently our journal provides basic the ability for direct messaging including direct messaging through private discussions. User's can directly message other users in real time through the use of websockets. Additionally, there more than one user can be assigned to private discussions meaning that a group of users in real time can communicate with each other.

However we could improve the social networking functionality of our submission by providing the ability to follow / unfollow other user's and displaying a feed of updates from users an individual is associated with. This would mean that academics within the journal can work together better as a team by keeping up to date with exactly what they are doing.

Privacy
We specified initially that users should be able to control the information which they set as public and private and that our system should conform to GDPR so that user's know exactly what cookies the site uses.

In this release we have implemented the ability for users to control information that they set as public and private. A user can prevent other users from seeing their email, name and institution. However one could argue that there should be more options other than public and private. Users should have the option to share this information with associated users only so that they can communicate better with trusted individuals.

However, in the first release we have not implemented a way for users to display and control the cookies that are enabled on the site. Which means if the site was to become public we would have to implement this.

Comparison With Other Products

Our software hits a niche market which is not supported by similar products. However, there are certainly features from similar products that can be used to evaluate our solution.

Reviewable
Reviewable is an extension built on top of GitHub for reviewing code. This is something that we ourselves could have undertaken initially. We could have built our solution on top of GitHub / GitLab which could have certainly prevented reinventing the wheel. Additionally, they have taken the approach of a reviewer to mark a review as resolved whenever they have addressed an issue. We have no concept of this, instead require authors to create a separate review each time. This is arguably harder to keep track of. As reviewable is an extension for GitHub it automatically incorporates version control. We ourselves could have added support for uploading Git repositories in order to visualise changes between different versions of a publication. The extension additionally has a mechanism for viewing what parts of the submission have been and have not been reviewed, this addresses the reproducibility aspect of each review better.

Evaluation Against A.M Vable (Vable et al., 2021)
From the recommendations suggested by (Vable et al., 2021) our solution facilitates the majority of recommendations outlined. The article suggests that in the peer review process test results should be published along with the code (Vable et al., 2021). Our solution incorporates this feature through the use of supporting documents. User's can publish their test results this way. Alternatively a user can publish the test result in the description of the submission. However, if there is the requirement that every submission should have test results provided with it then it may have been worthwhile to include this functionality.

(Vable et al., 2021) states that the academic peer review process is done on the entire submission and not incrementally. This provides verification that our versioning system reflects that of academia as if the submission needs to be updated all reviews become invalidated and the process needs to happen again.

(Vable et al., 2021) suggests that all lines of code need to be peer reviewed. Our solution could have provided a feature to mark a file as reviewed so that a reviewer coming back to the code knows which files they have and haven't reviewed.

As stated by the article it would have been beneficial to incorporate a way to provide a journal-wide code policy which would address the naming convention of variable names and so on so that reviewers can refer to it when deciding whether or not to approve the submission.

The purpose of the code review process is to provide reproducible results and better communication (Vable et al., 2021) . Our submission addresses each of these aspects. Reproducibility of results are supported by our supporting document system and comment threading system, communication amongst is supported through private discussions and the notification system.

Conclusions *(190007422)*

By the end of the project we had created a system which allows users of a journal to peer review and publish code. We had designed an API for the project and developed a single page application for it in ReactJS. We had provided unit and integration tests for each major component of the site and manually tested each feature as part of the agile process. We believe that our project makes the following achievements:

- The project considers a broad range of aspects with regards to journal management and the peer review process. It considers not just the peer review process of code but also managing users, networking with users, administering the system and the actual publication of submissions.
- The dashboard really allows users to customise the information related to them. It allows them to get an overview of the tasks that they need to carry out (for example, an editor assigning reviewers to a submission) in one place.
- The project provides real-time updates of events in the system. Users are notified in real-time whenever an editor has published a submission, the user needs to verify their email and so on.
- The private discussion system allows users to network with other user's in the system in real-time which ultimately helps to administer the journal by allowing editors to coordinate privately and authors to discuss their submission privately.
- The supporting document system allows authors to publish their results along with the submission allowing others to reproduce their results.
- Users of other journals within the supergroup can log into the journal where basic user information is shared. Each SSO user is treated as a user of the journal in their own right. They can be assigned a role, banned from the system and upload their submission.
- The export system provides the ability to export both the journal and its corresponding reviews directly to another journal, along with the corresponding user information itself.
- Our backend makes use of design patterns that help to keep the backend maintainable and testable. We make use of dependency injection to automatically inject services into controllers and api requests into our end points.

However there are a few downfalls which our project fails to address:

- The versioning system is quite lacking and there is no way to review which have been made on previous versions. It would be beneficial to keep track of reviews on previous versions so that editors and authors can track the submission as it progresses. Additionally, there is no way to view the changes between versions which means reviewers have to completely read the entire code again.
- For most pages and components there is a lag time when making an API request. A lot of components do not have a mechanism to show that the data is loading which may confuse or frustrate the user as it makes it hard to distinguish between an error and the request loading.
- When reviewing a file the comments aren't displayed inline with the actual code. From a HCI aspect this may cause the author to navigate to the code and back to the review to find out which section a snippet is from.

- Searching and querying submissions in the system is limiting. One can only search by name of a submission. There is no way to provide a tag for a submission allowing users to search by tag or for users to search by dates a submission was uploaded.
- There is no way for users to tag other users in comments. Which could result in users forgetting to view and respond to a comment.
- As the project has got bigger, using ReactJS becomes harder and harder to maintain. It would be better to use a framework that utilises MVC such as NextJS which provides server-side rendering of react applications. Going forward if were to develop the solution further for another release we would aim to address the following issues:
- Provide better versioning so that users can see the changes between each revision of a submission.
- Provide to display the number of comments of a given file within the directory viewer so that users know where comments are in a given file.
- Enhance the published submissions system so that users can rate publications and bookmark publications.
- Provide a way for authors to explicitly include their findings for a submission so that other authors can verify and check the findings.
- Provide the ability to add tags to submissions to make it easier to search for submissions.
- Provide more widgets for the dashboard so that users can tailor the dashboard to their needs even further.

Appendices (180003863)

The overall objective of the project was to develop a single journal within a federated platform for publishing peer-reviewed code, targeted towards publishing research software.

Initial Goals

At the start of the project we first decided upon a list of general goals we aimed to achieve throughout the project:

- **User registration and login system,** so that a user can create an account and login. This includes authentication processes to verify a user.
- **Ability to assign roles to specific users**, so they get access to the specific functionalities they need. For example, a reviewer can add users to the pool of reviewers. This hierarchy will reflect the hierarchy of an academic journal.
- **Allowing users from other journals of the federation** to log into and access our journal, and vice-versa.
- **Ability to upload and submit code** to the journal's server, and have it associated with one or more user profiles (the authors) as well as other relevant metadata such as title, tags and description
- **Ability to visualise submitted code,** this means providing an online text viewer which displays the content of each file in the submission.
- **Ability to comment on submissions**, both overall and on specific lines of the submission, as well as to reply to comments so as to create threads of comments.
- **Ability to mark submissions as approved**, and publicise them. One or more reviewers from the pool of reviewers can mark a piece of software as ready, ready with changes required or not ready.
- **A User Interface (UI)** that makes it easy to navigate without prior experience of the platform.
- **Ability to transfer submissions** to and from other journals in the federation. A submitter / reviewer can move their submission to another academic journal should this be appropriate.

As well as the required functionality we aimed to implement the following extensions:

- Ability to connect with or privately contact another user.
- Ability to personalise user profiles by linking CV, contact details, LinkedIn profiles, their institution and so on.
- Support for reviewing Jupyter documents and displaying them on the web application in a human readable format.
- Notifications so that a user is notified when the status of their submission has been updated, when a reviewer has been assigned to a submission or when an editorial board needs to vote on a submission. (which includes emailing users if they opt in)
- Embedding media into comments to make communication easier.
- Implement an API that clients can make requests to. For example if an institution wants to track the articles that their staff are requesting to be published.

- Versioning of submissions so that if a reviewer marks something as needing more work the submitter can upload a modified submission with a modified version number.

Initial Requirements Specification

In addition to general goals, we also put together a detailed specification list of requirements for our journal:

User Requirements

1. Login
    1.1. The login system should allow the user to login with an email and password or alternatively an account from a third party such as Google, GitHub and so on.
    1.2. The login system should allow the user to login with their email and password.
    1.3. There should be a system in place to reset the user's password if a user has forgotten their password.
    1.4. The password should be stored securely.
    1.5. The login system allows users to login using an account from another journal within the federation.
2. Registration
    2.1. Given a valid email address, forename surname, date of birth and password the user should be able to create an account with the journal.
    2.2. Given a valid third party account such as a Google account I should be able to register via OAuth.
3. User Profiles
    3.1. When a user sets up an account they should be able to add the institution that they belong to.
    3.2. When a user sets up an account they should be able to add a description about themselves.
    3.3. When a user sets up an account they should be able to add a research speciality
    3.4. When a user sets up an account they should be able to link their CV / Resume
    3.5. When a user sets up an account they should be able to add their contact details
    3.6. When a user sets up an account they should be able to toggle which information they want to set as public.
    3.7. A user should be able to search for other users within the journal.
    3.8. A user should be able to edit their information.
    3.9. A user should be able to contact other users via direct messaging.

4. Roles
    4.1. Administrator
        4.1.1. If a user is an administrator they should be able to ban other users and delete accounts.

        - 4.1.2. If a user is an administrator they should be able to remove submissions.
        - 4.1.3. If a user is an administrator they should be able to look at the logs.
        - 4.1.4. If a user is an administrator they should be able to assign roles to users.
        - 4.1.5. If a user is an administrator they should be able to view a list of all reports.
    - 4.2. Editor
        - 4.2.1. An editor should be able to comment on submissions.
        - 4.2.2. An editor should be able to assign reviewers to open submissions.
        - 4.2.3. When all reviewers have reviewed the work the editor should be able to see all comments and a summary of whether the reviewers have marked the work as ‘ready’.
        - 4.2.4. An should be able to see a list of all ‘approved’ submissions and mark it ready for submission.
    - 4.3. Reviewer
        - 4.3.1. A reviewer should be able to download the code.
        - 4.3.2. A reviewer should be able to read the code and the code changes.
        - 4.3.3. A reviewer should be able to comment on specific lines of code.
        - 4.3.4. A reviewer should be able to comment on general lines of code.
        - 4.3.5. A reviewer should be able to give a verdict on the submission.
    - 4.4. Author
        - 4.4.1. An author should be able to upload a submission
        - 4.4.2. An author should be able to update a submission
        - 4.4.3. An author should be able to assign co-authors to the submission
        - 4.4.4. An author should be able to provide a description of the submission, tag and title
        - 4.4.5. An author should be able to migrate my submission to another journal.
    - 4.5. Global
        - 4.5.1. Everyone should be able to leave public comments at specific lines of code as well as the submission as a whole.
        - 4.5.2. Everyone should be able to view a list of published submissions
        - 4.5.3. Everyone should be able to view a list of closed submissions
        - 4.5.4. Everyone should be able to view a list of submissions that are in progress.
    - 4.6. Submission
        - 4.6.1. As a user I want my profile to be linked with the submission.
        - 4.6.2. I want to be able to filter submissions.
        - 4.6.3. I want to be able to search for specific submissions.
        - 4.6.4. I want to be able to filter by a tag for a submission.
        - 4.6.5. I want to be able to upload Jupyter documents.
        - 4.6.6. I want to be able to mark something as a draft review.
- 5. Federation
    - 5.1. When I create an account for the journal I want to be able to log in using that account in other journals.
    - 5.2. I should be able to export/import submissions from other federations.

System Requirements Specification

Functional Requirements

1. Login
    1.1. The password should be hashed with the SHA-256 protocol.
    1.2. The user should specify their home journal on login.
    1.3. To authenticate a user from another journal we should make the following request https://t01.example.com/api/sg/sso/login?from=https://t02.example.com&state=abcd123. The home journal asks the user to login and asks whether they want to continue to the secondary journal. (See the SSO protocol for more details).
    1.4. On redirection the secondary journal should verify with the home journal that the token is valid (see the SSO protocol for more details).
    1.5. If the user requests to change their password given an email address an email should be sent to the user with a link to reset their password.
2. Registering an Account
    2.1. If we register via a third party account OAuth 2.0 should be used to verify the user's account using the service providers API. As much information as possible should be retrieved and the user should be required to fill out additional information.
    2.2. To register an account we request an email, password, forename, surname and date of birth as a minimum.
    2.3. A verification email should be sent to the user when they create an account and they will only be able to access their account upon verification.
3. Submission
    3.1. A submission should be submitted in zip format. A HTTP POST to the appropriate endpoint should be made to upload the file where we specify the zip filename, description and title to upload the submission. Upon retrieval the server processes the blob unzips it and stores it locally in a folder indexed by the submission ID. The metadata about the submission is stored in the Database indexed by a submission ID.
    3.2. When viewing a submission a HTTP GET to the appropriate endpoint is made. Where we specify the unique ID of the submission. The server sends a response with the name of each file in the submission.
    3.3. When viewing source code we make a HTTP GET request to the appropriate endpoint, specifying the submission ID and path of the file in the submission. The server then sends the content of the file in plain text.
    3.4. A submitter can add co-authors by making a HTTP POST request specifying the co-authors IDs. The user is verified to ensure they have permission to do this via their JWT token.
4. Authentication
    4.1. When the user logs into their account an API call is made to the appropriate endpoint. Upon successful verification a token is created for the user. This is then used in the body of all requests to verify the user. This token should be a JSON Web Token. We specify a period of time where this token is valid. When invalidated re authentication is required.

    4.2. All requests are made under the HTTPS protocol.

5. Roles
    5.1. An initial administrator should be set up so that roles can be given to other users.
    5.2. Stored with user data should be a role attribute specifying the user's role within the system.
    5.3. A roles tree should be implemented specifying the permissions each role has. For each role there is a discrete set of values such COMMENT, CAN_ASSIGN_REVIEWER…
    5.4. For each endpoint a list of permissions are specified specifying what permissions are required to process the data. From the access token the user is identified and their role is inspected.
6. Commenting
    6.1. To comment on a file a HTTP POST should be made specifying the comment, the path of the file within the submission, the submission ID and the line number. The message is included in the payload.
    6.2. A general comment is made via a HTTP POST request to a separate endpoint specifying the submission ID. The message is included in the payload.
    6.3. To get a list of all comments a HTTP GET request is made specifying the submission ID. This returns a list of general and line specific comments.
7. Assigning Reviewers
    7.1. To assign a reviewer a HTTP POST request is made specifying the user ID. The user is authenticated to make sure they have permission to assign the given user to a given role. The user role is then updated in the database table.
8. User Profiles
    8.1. Users can make a request to edit their details via a HTTP PATCH request specifying the fields they want to change and the new value in the payload. If they want to change highly sensitive information via their password they may be accessed to confirm their identity via an email. The user is authorised via their JWT access token.
    8.2. A user can make a HTTP PATCH request specifying details they want private and public in the payload.

Non Functional Requirements

Privacy

1. The system should comply with GDPR
2. All personal data must be encrypted.
3. Users can specify data to keep as private.
4. Users must be able to delete their accounts.

Performance

1. A page should load within one second of making the request.
2. The system should allow 100 concurrent users without performance noticeably degrading.

3. Each file upload should take no longer than one minute to process 100 MB of data. If it takes longer it should be because of the clients upload speed.
4. The system should allow a submission to store up to 2GB of data initially.

HCI

1. Users should be able to use the system within one hour of training.
2. The system should be accessible to the colour blind.
3. Certain services should be available to the blind.
4. The application should be able to be translated to another language.
5. The application should work in major browsers (Google Chrome, Mozilla FireFox, Internet Explorer 10, Safari, Opera, Edge).

Reliability

1. Clients should be able to access the system 98% of the time.

Security

1. The system must verify a user's details should they make an account.
2. The system must validate all user input.
3. The system must restrict access to features based on their roles.
4. The system must encrypt sensitive data using a suitable encryption protocol such as SHA-256.
5. The system must encrypt any data exchanged in the super group.
6. The system must not allow anyone to comment / view submissions unless they have created an account.
7. The system must authenticate each request.
8. The system must ensure integrity of client account information.
9. The system should provide reliability, such as backing up user accounts.

Writing Code

1. Developers should stick to the SOLID[1] principle.
2. A line of code should not exceed 80 characters.
3. A function should not exceed more than five arguments.
4. Local variables should be written in camelCase.
5. Functions should be written in camelCase.
6. Constants should be written in SNAKE_CASE in capitals.
7. Properties should be written in camelCase.
8. Code should be written in either Visual Studio Code or the WebStorm IDE.
9. Avoid using the ‘any’ type in TypeScript whenever possible.
10. Favour an abstraction of a concrete implementation.

Writing Unit Tests

1. Jest unit tests should make sense as a sentence for example it(‘logs the time to complete’)
2. TDD should be followed. This means we should be writing to an interface.
3. Unit tests should be arranged in the following way: arrange, act and assert.
4. A unit test should only test one specific thing.
5. Unit tests for JavaScript and TypeScript should be written in Jest for both the front end and the backend.

Pull Requests

1. A branch should only be merged into the main branch if another developer has approved it.
2. Merge requests that affect fundamental logic of the system should be approved by two developers.
3. A branch should be titled issuenumber-title-of-issue
4. All unit tests should pass before merging a pull request.
5. Magic stringsThe above standards when considering to approve a pull request.
6. and numbers should be avoided

Interim Reports

Throughout the project we wrote weekly Sprint reports detailing the progress that had been made during that week. Since our Sprint cycles lasted 2 weeks, the first report of each Sprint listed the goals at the start of the Sprint and the second evaluated whether we achieved these goals. The second report also evaluated the effectiveness of our work during this time, and included a list of things which we should start, stop and continue to do.
These reports greatly helped us to track and evaluate our progress, and refine our workflow as we progressed through the project.
These reports have not been included inline (as there are many of them), instead they can be found in the Reports/Weekly directory of the submission.

We also wrote progress reports at three points throughout the project (the previous deadlines). These were larger reports that detailed and provided evidence of our work up until that point. Similarly these reports can be found in the Reports/Progress directory of the submission.

Changes During the Project

Throughout the project we made use of our meetings and regular reviews to identify areas of weakness, such as technical debt or design problems, or features that were missing or overlooked previously.
The major changes were:

- Reports
    - Initially during the planning phase we did not write explicit reports summarising progress and evaluating our performance. However once we began the first Sprint and wrote our first Sprint report we recognised the importance and benefit of these, and as a result decided to focus on frequent summaries/reviews to allow us to streamline the development process.
- Backend testing library
    - Initially we chose Jest as the library to test the backend with. However we soon discovered that the alternative library Mocha was more configurable and better suited to testing both our Express server and our MongoDB models. As a result we switched to using Mocha as our testing library, Chai for assertions and Sinon for mocking. This was detailed in Sprint 1 Report.
- Unit tests

    - Initially we planned to use an in-memory database for unit testing the backend. However we realised it was better to mock the unimportant methods (in the sense that they were not the focus of the test) using Sinon instead. This allowed us to truly focus each unit test on a single feature and avoid testing multiple features. This was discussed in Sprint 1 Report.
- Supported files
    - In our initial plan for the project we had intended on only supporting zip files to be uploaded as the contents of a submission, as this allowed us to support both files and directories and we believed most projects would consist of several files and folders. However after discussion with the rest of the groups in our supergroup and further review, we decided to also support uploads of single files as well. This required changes to how we stored and represented files in the backend, however improved the functionality of our website.
- Comments
    - Initially we had planned to keep the notion of reviews and comments separate (in the frontend and the backend). However it occurred to us this would have been a design flaw and also contrasted with other groups in our supergroup, so we changed our approach so that each comment was associated with a parent review (in addition to having general and specific file comments, nested comments etc). This was mentioned in Sprint 2(i) Report and Sprint 2(ii) Report.
- Backend API framework
    - During the first semester we built our backend using ExpressJS and TypeScript using the MVC pattern, a choice we were very happy with. We had been implementing the pattern ourselves, writing nested controllers, specifying the endpoints for each and performing explicit data validation. It came to our attention that there was a series of TypeScript libraries called TypeStack that could hugely simplify this process, and make our life a lot easier going forward. We made a proof of concept server first to confirm this, then converted our backend to utilising the TypeStack packages. This was a significant structural change to our project, however it was incredibly useful and greatly improved our backend structure and saved us a lot of time for the rest of the project. This was described in Sprint 3 Report.
- Code comments
    - Since we aimed to provide the functionality for a user to write a comment on a range of lines within a source code file (as opposed to a single line), our commenter component consisted of two sliders to specify this range. However we found this was primitive and not very user friendly, so we altered it to specifying directly which range of lines to comment on, which functioned well. This was detailed in Sprint 4(i) Report.
- Supported files
    - Initially we hadn’t planned to support viewing of PDF or markdown files (i.e. have specific components to handle them, they would have been treated as raw files instead). However we found nice ways to handle both of these file types and so added them. This was mentioned in Sprint 4(i) Report.
- Jupyter Documents
    - Similar to PDF and markdown files we previously had no explicit support for viewing Jupyter documents. However we decided to develop a custom

    - Jupyter document parser and viewer to allow users to view the formatted code, text with markdown support and execution results of cells within a Jupyter document.
- Real time updates
    - When implementing announcements, notifications and private discussions we used only our backend REST API, and would check for updates (i.e. new messages or notifications) on a timer. However we realised this did not provide as good a user experience as it could, and was also wasteful/inefficient. As a result we developed a websocket system, implemented using Socket.IO, for bidirectional communication between the client and server. This allowed us to automatically load in new notifications or messages for events as soon as they occurred, and also saved resources as we only pushed the new data required to be displayed rather than reloading the entire resource on a small update. This was discussed in Sprint 8 Report.
- Themes
    - Initially we had aimed to develop a single website-wide theme, with the colours selected carefully for accessibility and ease of use. While this was successful, we realised we could also provide a range of different themes, and allow the user to choose their preference. This was very successful and added significantly to the customizability and user experience of our website. This was described in the Sprint 8 Report.

Future Changes

In the light of the experience gained across the duration of the project, we would make the following changes in the future:

- Allow users to create new versions via modifying existing files inline on the online code editor (rather than having to upload an entire new version), similar to the GitHub web console
- Add support for execution of Jupyter documents to our custom viewer (would likely require a background job communicating with a Python virtual execution environment)
- Allow comments to specify a portion of a PDF or Jupyter document to comment on (would be tricky and require segmenting the documents)
- Allow shorthand referencing of submissions and comments by their ID across the site (similar to how you can reference GitHub/GitLab commit hashes)
- Improve our frontend testing
- Allow creating a private message with another user directly via clicking on their icon/profile
- Make further use of notifications throughout the site
- Conduct a usability survey to review/improve the user experience
- Extend pulling submissions from GitHub to pulling submissions from anywhere (wouldn't require many changes)
- Add visually pleasing loading icons when a page hasn't fully loaded (i.e. loading a user's details etc)

- Protect our RSA private key for decryption further via storing with a hardware security module
- Store IP addresses and add functionality to ban or time out users
- Implement DDOS protection via implementing a basic WAF
- Implement a cloud based logging solution and an admin dashboard to view live logs and events
- Make use of serverless technologies to run background jobs (such as those provided by AWS, Google Cloud etc)

Postman API Documentation

The important endpoints of our backend REST API are documented here:
https://documenter.getpostman.com/view/18243967/UVyrUGFf#4c54cf7b-be93-4424-9184-acf9efcdaa33

The generated documentation of the backend can also be found in the Docs directory.

Testing summary *(190007422)*

Backend

Unit Tests

In the backend of our solution we used the mocha, sinon and chai testing stack for our unit and integration tests. Mocha was used as the testing framework, sinon was used as a library to stub and mock dependencies and chai was used to make assertions.

Controller

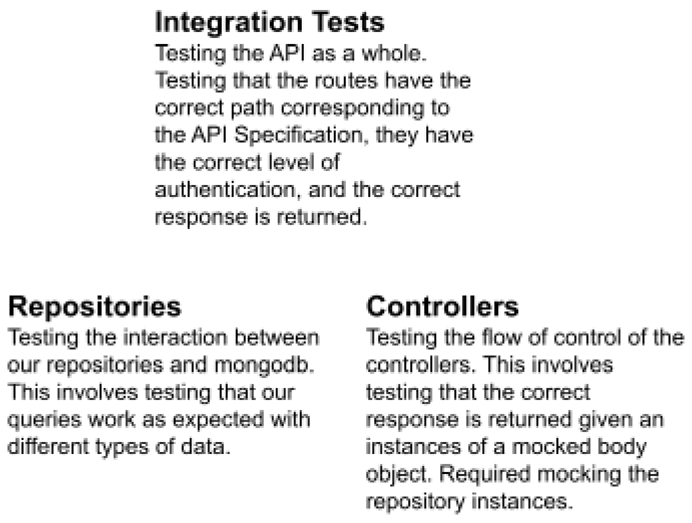
We designed our API so we could provide unit tests for the flow of control of our controllers. This was done via creating mock “repositories”, which are objects designed to mock a dependency, such as database models or services such as the email service, that a controller relied upon with sinon stub methods. We used these mock repositories to control what was returned from the dependent methods (that weren’t the objective of the test). They also allowed us to test that the correct object or correct error was thrown based on the return value of a model or the data injected into the parameters. We could only achieve this because of our use of dependency injection and separating everything into an interface so that we could mock dependencies.

Mongoose Models

We tested our mongoose models and their associated queries through the use of the mongodb in memory server. This allowed us to create and test queries and validations during our tests without it affecting the database associated with our application. This essentially allowed us to verify that our interface that abstracted the retrieval of data from the database worked correctly. Overall we had a solid code coverage with regards to testing the mongoose models as each model had associated tests with them.

Integration Tests

Our integration tests for the backend API were made using the supertest testing framework. The purpose of these tests were to test that our API as a whole provided the expected response and response time given an actual request. For example, they allowed us to test that if an attribute is not provided then a 400 error is returned and that an endpoint is only accessible through the correct role. We could have used postman to write this, however we wanted the tests associated with the unit tests so that we could run them via a test runner in GitLab.



Integration Tests
Testing the API as a whole. Testing that the routes have the correct path corresponding to the API Specification, they have the correct level of authentication, and the correct response is returned.

Repositories
Testing the interaction between our repositories and mongodb. This involves testing that our queries work as expected with different types of data.

Controllers
Testing the flow of control of the controllers. This involves testing that the correct response is returned given an instances of a mocked body object. Required mocking the repository instances.

Figure 19.

Frontend

Unit Tests

To automate testing of the frontend we use react testing utils to test the frontend. This consisted of testing that an error occurred if a user entered invalid data, testing that a page could not be accessed if the user did not have the correct role. Our tests mainly covered testing individual components manipulated the DOM correctly and that pages had the correct components and gave the correct response to errors.

The main challenge we faced when writing unit tests for the frontend was how we would stub and mock calls to our API. We found the library *'msw'* incredibly useful. The library intercepted our API and allowed us to return mock data. This allowed us to fake the data that was returned and test that the component responded correctly to the data.

Unfortunately our frontend tests are somewhat out of date and could definitely be improved.

Manual Tests

Built into our agile process is the feature of an acceptance criteria associated with each card. After an issue was merged another developer on the team would be responsible for verifying that each aspect of the acceptance criteria was met. If met, the tester would be responsible for closing the card. This gave instantaneous feedback to ensure that a developer had met all points of the acceptance criteria.

Coverage

Overall we have good coverage in terms of unit tests and front end tests of the controllers. However, we have very little automated tests for the supergroup protocol. If any aspect of the supergroup protocol broke during development there would be no way of identifying this. This is mainly because good testing of the protocol requires connection to another mock journal and the overhead of setting this up seems great and the reliability of this would be questionable.

There is no automated coverage of the frontend - or at least very little. This is something that would have to be addressed going forward. However, one can be confident that the frontend is unit tested in a sustainable way as we have decomposed every component where possible. However despite all of this there are a few smaller components where there is no coverage; the *announcementController, announcementsController, privateDiscussionController.ts* have no integration or unit tests around them. We simply did not have enough time to implement them.

All of our models however do have unit tests surrounding them. We prioritised this as both controllers and services rely on these and thus we made it a priority to provide tests wrapping these queries.

Improvements

Although our testing stack is large there are a few improvements we could make to it. Initially we set out to incorporate performance testing into our automated tests. This would provide analytics into the performance of each page and endpoint in terms of time of response. However we found that there simply was not enough time to achieve this. Furthermore we have minimal frontend unit tests going forward this would be something that would need addressing as currently at the time of release there is no way of knowing if a controller has been broken other than going through the entire acceptance criteria.

Bibliography

(n.d.). Redis. Retrieved March 31, 2022, from https://redis.io/

(n.d.). Podman.io. Retrieved March 31, 2022, from https://podman.io/

(n.d.). Formik: Build forms in React, without the tears. Retrieved March 31, 2022, from https://formik.org/

(n.d.). Reviewable - GitHub Code Reviews Done Right. Retrieved March 31, 2022, from https://reviewable.io/

AngularJS: Developer Guide: Services. (n.d.). AngularJS: API. Retrieved March 31, 2022, from https://docs.angularjs.org/guide/services

Belmer's, C., & Belmer, C. (2019, August 6). *A NoSQL Injection Primer (With MongoDB)*. Null Sweep. Retrieved March 20, 2022, from https://nullsweep.com/a-nosql-injection-primer-with-mongo/

C# Best Practices - Dangers of Violating SOLID Principles in C#. (n.d.). Microsoft Docs. Retrieved March 31, 2022, from https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/may/csharp-best-practices-dangers-of-violating-solid-principles-in-csharp

IETF, Jones, M., Microsoft, Bradley, J., Sakimura, N., NRI, & Jones, M. (2015, 05). *JSON Web Token (JWT)*. Retrieved 03 30, 2022, from https://datatracker.ietf.org/doc/html/rfc7519

Man in the Middle Attack: Tutorial & Examples. (n.d.). Veracode. Retrieved March 20, 2022, from https://www.veracode.com/security/man-middle-attack

Mongodb, Inc. (n.d.). *GridFS*. Retrieved 03 30, 2022, from https://www.mongodb.com/docs/manual/core/gridfs/

MongoDB, Inc. (n.d.). *MongoDB*. MongoDB. Retrieved 03 31, 2022, from https://www.mongodb.com/

Morris, D. (2017). *Scrum in Easy Steps: An Ideal Framework for Agile Projects*. In Easy Steps Limited.

Muscat, I. (2019, April 10). *Recommendations for TLS/SSL Cipher Hardening*. Acunetix. Retrieved March 20, 2022, from https://www.acunetix.com/blog/articles/tls-ssl-cipher-hardening/

Smith, S. (2022, 03 26). *Overview of ASP.NET Core MVC*. Microsoft Docs. Retrieved 03 30, 2022, from https://docs.microsoft.com/en-gb/aspnet/core/mvc/overview?WT.mc_id=dotnet-35129-website&view=aspnetcore-6.0

Socket.IO. (n.d.). Socket.IO. Retrieved March 31, 2022, from https://socket.io/

SSL Server Test (Powered by Qualys SSL Labs). (n.d.). Qualys SSL Labs. Retrieved March 31, 2022, from https://www.ssllabs.com/ssltest/

Three Five Two. (n.d.).

TypeStack. (n.d.). *TypeStack*. TypeStack. Retrieved March 31, 2022, from https://github.com/typestack

Vable, A. M., Diehl, S. F., Glymour, M. M., & Diehl, S. F. (2021). *Code Review as a Simple Trick to Enhance Reproducibility, Accelerate Learning, and Improve the Quality of Your Team's Research* (Vol. 190). American Journal of Epidemiology. https://doi.org/10.1093/aje/kwab092

What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy. (n.d.). PortSwigger. Retrieved March 20, 2022, from https://portswigger.net/web-security/cross-site-scripting