CS3099: Junior Honours Project

Peerbit

peerbit

University of
St Andrews

Artem Rakhmanov Ben Johnson Keanu Eyles Matthew Oates

31 March 2022

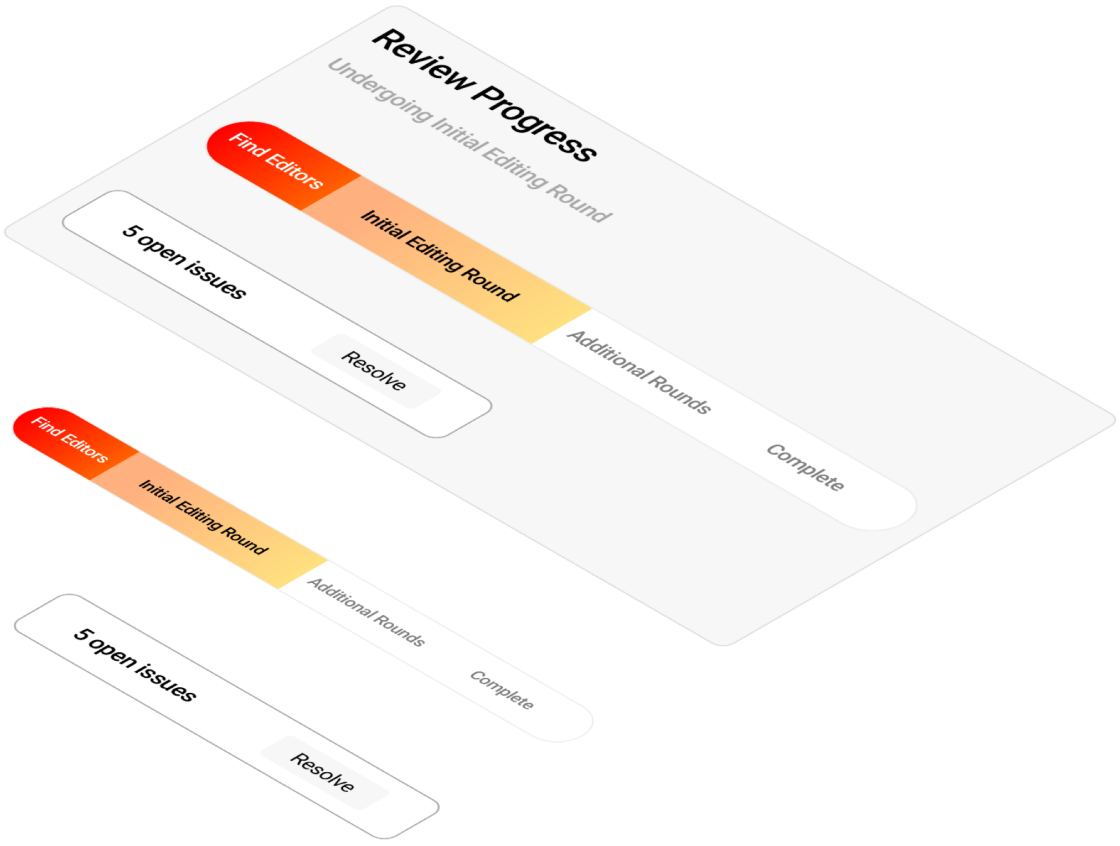
Abstract:

Over the course of a year our group has built a federated platform for publishing and reviewing code. Peerbit is capable of registering secure user profiles and allowing existing users to edit their details using Google Firebase. Its clean and efficient UI created using React allows for the publishing and peer-reviewing of code submissions. These features then extend to the supergroup at large, as users from other code journals are able to log into and export/import code from other journals within the federation.

Declaration:

“We declare that the material submitted for assessment is our own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 11,361 words long, including project specification and plan.
In submitting this project report to the University of St Andrews, we give permission for it to be made available for use in accordance with the regulations of the University Library. We also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. We retain the copyright in this work, and ownership of any resulting intellectual property.”

Contents



Web Application Link:

https://peerbit-6557f.web.app/login

Introduction

Academic journals have long been the standard for publishing peer-reviewed research publications. They provide a hub for academics to share and refine their work with their peers and possibly form partnerships to collaborate on future projects. We set out to engineer a similar platform expressly for coding. Users are able to publish their code and receive feedback from others through a multistage peer-reviewing process or provide feedback of their own to other code submissions via an easily searchable discovery page. Our platform, Peerbit, does not exist in isolation, however, it exists as part of a federation of servers able to communicate by sharing users and having the ability to import or export code submissions. We were successful in implementing all of these features and even extended some of them beyond what was required of us, such as the ability to edit user details, code versioning, and optimization of our user interface and backend structure.

We decided to utilise several third-party libraries and resources to build our web application. React is a robust Javascript library that includes numerous tools and functionality complimentary to frontend development. Google Firebase makes storage of our backend data, like user details and code submissions, secure and simple to access. To accommodate supergroup functionality, we have developed a REST API that follows the supergroup defined protocols for SSO authentication and submission transfer. Throughout the development process, we were guided by the MVVM (Model View ViewModel) software architecture model. The core principle is to keep code that generates the interface (Views), handles data (Models) and logic & orchestration (ViewModel) separately, allowing for cleaner code organisation and collaboration.

We kept our work on-track and organised by utilising the scrum development methodology. We largely managed to keep to a bi-weekly sprint sprint cycle and regularly kept in contact with each other at weekly meetings and the supergroup at large on Teams at agreed upon times. Despite numerous periods of illness and other disruptions, sprint cycles remained smooth by frequently dividing frontend and backend work amongst group members based on experience with each area. In evaluating Peerbit, we feel we have been highly successful in building a clean and approachable web application which achieves everything a coding journal should be capable of, as demonstrated by the results of our functional testing. Overall, this report aims to delve into the design decisions we made while developing Peerbit, highlighting any struggles or successes we encountered along the way.

190018240

Project Management

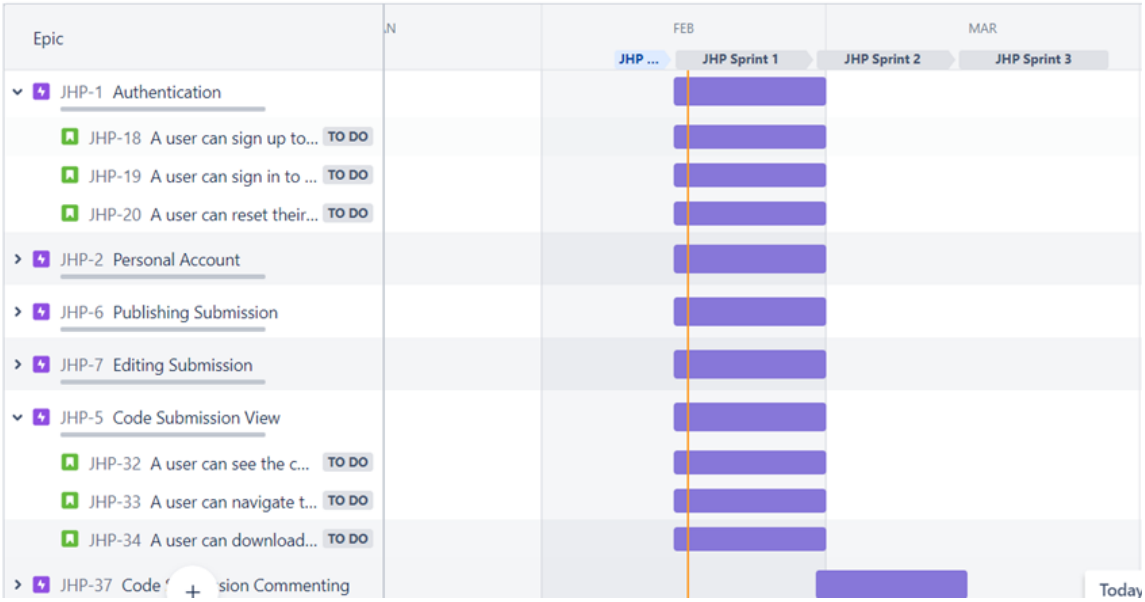
Development Methodology

When developing our final product, we made sure to make good use of the scrum methodology in order to keep ourselves organised and on track to complete our goals. As a team we are all in agreement that we used the scrum well to complete a good final product on time.

Over the course of the year-long project we tweaked how we used scrum as we developed our knowledge in the agile methodology and found new ways to better utilize and gain as many benefits we can from scrum. We were also forced to make some changes due to some unforeseen circumstances that halted work from the group. For the MVP we only had 4 weeks to complete it after starting, therefore we decided to go with much smaller sprints (1 week). At this point we felt that multiple short sprints would work better as we would have more flexibility to adjust to any unexpected bugs and issues and resolve them in a next sprint. This ended up working extremely well for us given the short turnaround, albeit there was a lot of stress towards the end to make the deadline. For the MVP we primarily used a user story approach to assign different tasks, and this led to some stories being completed before another story which it was dependent on was completed.

In the second semester, we all came together as a group and evaluated our usage of scrum. Our findings were that whilst the one-week sprints worked for the MVP, we should try and steer away from them as they caused a lot of stress and a lot of extra planning as we were constantly having sprint review/planning meetings. We therefore decided to go with a two-week long sprint approach, which definitely helped us to plan better and really reduced the stress and pressure to get a full sprint done in a week. We also again identified the issue of assigning tasks based on user stories, due to the sometimes-inefficient completion of tasks. To overcome this, we decided to use epics. Epics are a single feature of our application and can be described by multiple smaller user stories. In our final plan, we split the 34 user stories into 18 epics, helping us identify exactly what needs to be completed for a single feature. To facilitate this change to an epic based approach we decided to transfer to the project management software Jira (more on this coming). We were able to use the project roadmap feature in order to keep track of the development of certain epics and also keep track of progress on sprints (See image below for the project road map showing some of the sprints). In semester 1 we also noticed that there was no way for us to measure how big a user story was and therefore it was very difficult to distribute tasks fairly as we seemed to be guessing on the spot how long it would take to complete a story. In order to overcome this difficulty, we introduced story points, which would help identify how work was needed for a story with respect to other stories. These story points help us to keep track of how much work we have left to do at any given time and allow us to make changes to our pace after each sprint. In the second semester we also found that the use of two specified front-end developers and two back-end developers really helped with the progress. This allowed us to specialise in what we were doing, and we only had to focus on the technologies used in whatever “end” we were working on, making there less time spent learning

new technologies when swapping between front and back end. The final change to our scrum development process between the semesters was the addition of a new stage on the sprint board called acceptance. This means we ended with 5 stages: To do, in progress, review, acceptance and done. We added acceptance due to the plethora of merge issues we had in the first semester. The acceptance stage was for any tasks that had been completed and reviewed and were ready to be merged, however hadn't been merged yet.



In both semesters, the way in which we went around a sprint was mostly the same. For each sprint, we started off with a planning meeting, in which we would discuss which epics/user stories we wanted to tackle and to assign them for the week ahead. During each individual sprint, we would hold scrum meetings twice a week. However, we would still meet at least every 2 days in person or on Microsoft Teams to discuss progress and any potential issues we had with the group. We did find the in-person meetings were much easier and more productive as it was easy for us to look at each other's code and potentially do some peer programming to try and overcome bugs and solve problems. We were also always in contact on the messaging platform Signal talking about anything related to the project. At the end of each sprint, we would hold a sprint review meeting. In this meeting we would take it in turns to discuss our progress during the previous week(s), ensuring we mention which tasks were completed (and to what extent) and which tasks may need to be moved to the next sprint. We would also give a demonstration, if possible, of the new features and walk each other through any code. Each group member would then have a chance to critique this work and suggest any improvements. At the end of the meeting, we would generally attempt to merge the working code together.

Unfortunately, this semester our group suffered from many issues from illnesses including Covid and various other external factors. This led to development at the start being extremely slow, and some sprints not getting completed as some group members were unable to work for weeks at a time through no fault of their own. This did lead to the last few sprints being a lot

busier than we would have liked. We did however persevere through these issues, and we are all in agreement that our final product is of a good standard, especially given the circumstances.

Overall, I believe our use of scrum throughout this project was excellent. I believe we reaped the full benefits from the scrum methodology, and it really helped us to streamline our workload and avoid many of the issues you would expect from developing an application of this size. We are all pleased with how we overcame some of the issues we had with using scrum in the first semester by expanding and improving our implementation of the methodology. Given some of our external circumstances I believe our excellent use of scrum helped us to minimise these impacts as much as possible.

190004593

Development Tools

During the course of our development we used a selection of technology to boost productivity, reduce inefficiency while also ensuring each group member is accessing the most up to date work.

Version Control

As required by the module, the group's version control is hosted off of St Andrew's school of Computer Science's Gitlab server. Of course, the group also used git for collaborative development. This worked well, as all group members have experience with version control and more particularly, were all at least adequate with git.

As our sprints became more structured and better prepared, the group agreed that there were better services available to us, and so we migrated to Jira, so that we could properly structure and execute our ideas and plans.

Cloud storage

We decided to use Google Drive to share assets such as images and PDFs, also to collaboratively write documents together - which was useful for progress reports and project reports. Google Drive made our collaboration very simple.

Communication

The group spoke through formal and informal channels. Formal communication was in the manner of infrequent meetings, either in-person or Microsoft Teams calls (depending on our isolation statuses and what country everyone was in), and then informal communication was in our Signal group chat. This worked really well for us, when we had formal communication, the group would prepare beforehand so that any questions or ideas we had, could be presented at the time of the meeting, whereas in our signal group chat, there were quick messages between each other when something more quick needed sorted, or each member gives a situation update, letting everyone else know what they're working on, when they're expecting to finish etc.

190018018

Supergroup Interaction

Our team's supergroup interaction has remained unchanged since the beginning of the project. We were in regular contact via the Teams chat, attended calls to work on or improve protocols for inter-journal functionality and kept in touch with one of the teams for more rigorous testing and development.

Due to unforeseen circumstances of a group member responsible for supergroup contact and work, we were not able to remain as proactive as we used to be in the first semester, when we were actively suggesting potential solutions to SSO, for example. Our team greatly appreciates the effort and coordination a few most proactive teams showed to design and test the protocol of our supergroup.

180023970

Project Details

System Functionality

Utilising the Scrum development model, we were able to implement all of the functionality described in the project specifications. Additionally, beyond these core requirements, we were able to complete many of the more advanced user stories we were unable to implement in the MVP, as well as some extra functionality we found necessary during the second half of development. Peerbit's core functionality includes the ability to:

- Access a clean, efficient web-based user interface.
- Register, log in and log out of a user account.
- Log in and out of other journals within the supergroup.
- Post source code files onto the platform, so that others can see them and perform code review.
- Remove / resubmit code to another journal.
- See comments, reply to them, or post new ones.
- Change / recover a password.
- Edit / provide supplementary data to a submission during the review process.
- Find research codes on the Discover page.
- See a user's own code submissions on the Account page.

The overarching principle that guided us throughout the development of our MVP was to provide an excellent solution in the domains of our focus and only afterwards expand towards other areas and add features. In other words, our efforts were primarily spent on ensuring the core functionality of our code journal was working as intended. However, once these features were completed to a sufficient standard, we were able to extend our functionality to include important quality-of-life changes and useful additions. These include:

- The ability to change user details.
- The ability to view metadata about code submissions such as the number of views.
- A review page allowing for the implementation of our peer-review protocol which includes:
    - U.I. elements such as a progress bar to visualise the progress of a code submission's multistage review process.
    - Code versioning so that reviewers can see code changes over time.
    - The ability to suggest additional changes or approve a code submission's current iteration.
    - Resubmit edited code after considering peer feedback.
- A refactoring of our backend structure to reduce the number of read requests made to Firebase as well as increase efficiency by the use of teasers.
- A redesign of our user interface to increase its visual appeal, readability, and efficiency.
- An expansion of comments into a dedicated discussion section on a code submission's page.

Product Design

The project specification had intentionally given a lot of freedom in product and feature decision making. Our team decided to address this challenge with a sufficiently comprehensive product design procedure. The main goal of this stage of project development was to identify product requirements and provide viable solutions that are likely to satisfy the user's end goals.

Firstly, we had carefully analysed the explicit requirements and given background information. We were able to outline core principles expected from the product, the industry, potential user groups and existing product alternatives.

Secondly, we conducted an efficient high-level industry research into the aforementioned key components of our product. Among others, our primary focus was to research peer review practice in great detail, learn about academic publishing and evaluate potential real-world use cases. This research allowed us to build a common understanding of the environment around our product and made feature design more straightforward and the output of better quality.

As the result of a few stages of preparation and product design, we arrived at a mutual understanding of the importance of focusing on providing excellent review functionality. Our vision was an application that would have a user friendly and straightforward yet powerful review process with additional features that would let users stay within the app for any review purposes.

The next stage after product definition and user story design was UI design. We have focused on desktop web design due to assumptions about potential use cases with responsiveness in mind. Our UI design process took two stages, with one being the MVP design and the final submission design. The user experience is valued on an equal level if not more important in our project than pure functionality or efficiency. We decided to be conservative with the number of features our product offers and focus instead on their usability and quality.

Throughout our development, instead of increasing complexity, we attempted to do the contrary. We had successfully identified over engineered features and user interactions and replaced them with simple and elegant solutions. Through iterative development, we were able to swiftly respond to feedback from anyone in the team and make changes that would help us bring our product closer to our vision for it.

Overall, we believe that we achieved a thoroughly designed solution that is on par with the highest standards users demand nowadays. It is functional yet simple, with features that integrate seamlessly together and offers a great user experience.

180023970

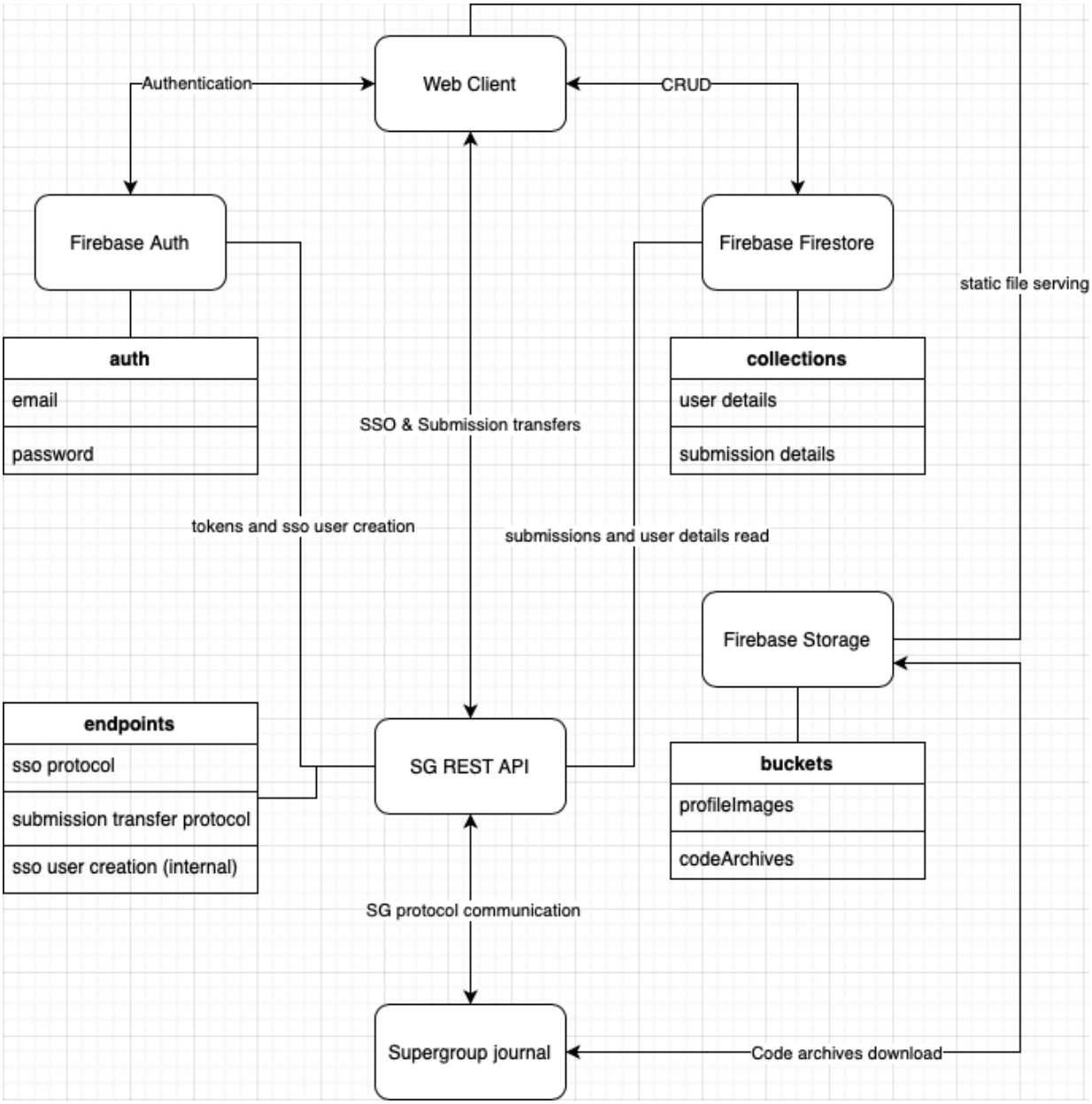
Software Architecture

Overview

Peerbit is built as a serverless web application that relies on a theoretically infinitely scalable infrastructure. It has a heavy client that performs all of the internal CRUD logic on the front-end via the serverless API. We have extended our application with our own REST server in order to achieve the supergroup inter-journal communication functionality.

Our choice of technological stack and the design of our software architecture has been made with scrupulous attention to detail. We have collected input from everyone in the team in terms of their former experience and thoughts on solutions, evaluated initial MVP requirements and extensively researched the market of different technologies that could help us develop our product. We have trialed our technological stack during our MVP development. It has proven to exceed our expectations in terms of flexibility, performance and ease of development which led to our final decision to move forward without changes towards the final submission.

Below is the scheme of our software architecture and infrastructure, exploring our system from schematic point of view to help understand our solution:



This section will explore Peerbit's software architecture and technological stack in great detail and will augment formal descriptions with schemes starting with the Front-end module.

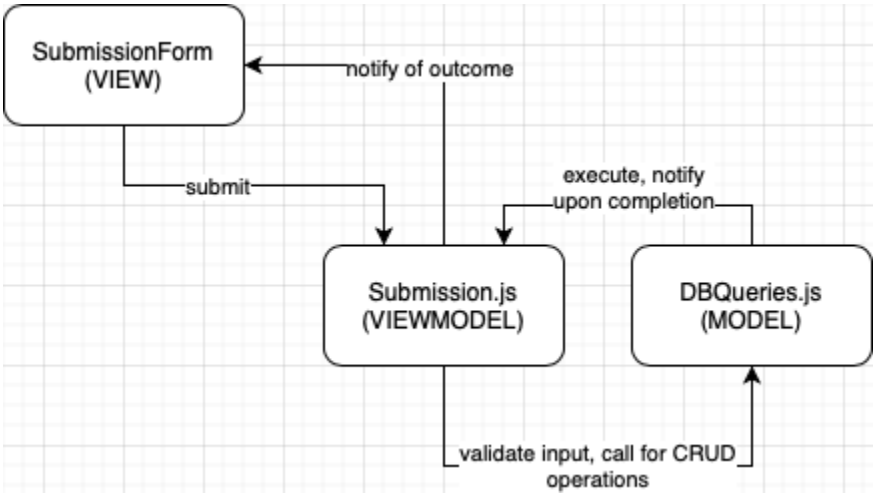
Front-end

Our front-end is built with the React.js framework. The main reasons are the component structure that allows for great reusability in an app-like website and a shallow learning curve in comparison to comparable frameworks.

Due to our decision to experiment with a serverless architecture and the need to accommodate CRUD API calls code along the UI code, the requirement for a robust front-end architecture was evident. Among other requirements was the ability to be able to work asynchronously on UI code and API code, proper file management due to the size of the codebase and a shallow learning curve due to time constraints of this project.

Our choice of front-end architecture is Model-View-ViewModel (MVVM). Although it is primarily a mobile application paradigm that is particularly widespread in iOS development, it is a very powerful abstraction architecture that is intuitive, flexible and scalable. Model in MVVM stands for files that are responsible for data storage in memory, ViewModel are files that perform business logic / API operations and View files are responsible for UI code. This clear structure allows for a well organized codebase, with clear separation of concerns and has proven to be intuitive and powerful since we started our MVP development.

The diagram below represents the MVVM architecture for publishing code submissions in relation to the code files employed in this feature and their communication:



Serverless Backend

Our team had initially decided to proceed with a serverless backend in mind for the MVP version. The reasons behind this was the input from one of the team members in relation to a steep learning curve and complexities of building our own database, api and other services infrastructure ourselves. In addition, time constraints did not allow for such work without extensive experience that none of the team members possessed. Therefore, it was agreed that our team would seek to discover a solution that would outsource infrastructure development to a product or framework to then focus on feature and functionality development instead.

We decided to trial the Firebase PaaS suite of services for our MVP submission. Firebase platform offers numerous services ranging from high performing NoSQL databases to authentication and analytics services. The idea behind it is that the product offers a complete API to all of their databases and services for all major platforms and allows developers to focus only on functionality development rather than preparation of web servers and self-hosted or managed databases. The PaaS such as Firebase is an opinionated framework, which was clearly identified in our research for technological stack solutions. Our team acknowledged this and began implementing the MVP functionality to identify whether the flexibility this platform offers would be sufficient for our product. Upon MVP completion every team member was impressed by the performance of this PaaS and agreed that it had exceeded every expectation, consequently resulting in our decision to move forward without changes for the serverless Firebase backend plan. Here is our team's usage of Firebase PaaS services in our product as of the final submission time:

Firestore Database	Powerful NoSQL Database with strong query API, realtime web-socket capabilities and robust table structure	Main choice of database for user details, submission data and collections of data items for feeds
Realtime Database	Cutting-edge JSON Database with a NoSQL infrastructure implementation that offers unparalleled performance and concurrency	Choice of database for anonymous realtime private conversations
Firebase Storage	Object storage service	File storage for images, code files and any attachments
Firebase Authentication	Secure interface to Google Cloud Authentication services, that offers an extensive range of user management features	Authentication and Token management service, both for internal users and adapted for SSO
Firebase Hosting	CDN web hosting	Front-end app hosting

Our team relied on the external Firebase Authentication service for user management. It was mutually agreed that in the timeframe given for this project our team could not implement a secure solution of good quality for authentication. The use of this service has greatly reduced development time and allowed more focus on our product design and feature development, while allowing our product to possess an industry standard authentication that Google's products use. Our team has adapted the usage of it for external journals via our implemented SSO, which will be covered in more detail in the "Other implementation details" section.

The initial choice of our database resulted in the need to communicate with the database directly via the CRUD API on the front-end. Although we had concerns as to the flexibility of such an approach, we soon discovered that it was sufficient. Our use of prepared infrastructure allowed our team to complete first features in relation to user details and submission posting in record time with robust performance and stability. Underneath the implementation of this product is a NoSQL database. Our team has been acquiring knowledge as to the best practices of NoSQL data management and query which resulted in efficient optimisations that will be discussed in the "Optimisations" section.

For functionality that could not be performed client-side, we made use of the Firebase server environment SDK and successfully implemented complex features such as SSO and inter-group submission transfer.

As it was mentioned, our approach entails CRUD logic to be present client-side. Code that is responsible for Firebase communication neatly relates to our MVVM front-end architecture as the ViewModel files. This resulted in a very structured and intuitive codebase that can scale effortlessly for collaboration and further development.

Supergroup Server

In order to implement the supergroup functionality and adhere to the agreed protocol our team developed a REST server using a python framework called FASTApi. This server is hosted on school servers via uvicorn python environment.

Our service interacts with our database and authentication services via a server environment Firebase SDK, to facilitate token management for our SSO functionality and query our database for submissions export / import.

Infrastructure Summary

Our choice of technological stack and infrastructure has proven to be of great quality. This section will explore the main benefits of our choice which were among the most important aspects used in our decision making.

Firstly, our serverless backend powered by Firebase is theoretically infinitely scalable. Underneath the API we are using is Google Cloud infrastructure that dynamically responds to any load and concurrency, therefore making our product capable of handling any traffic. Although this is not a realistic use case scenario nor was is among the requirements from our project, we believe that having a system capable of handling any load is beneficial to us as developers and potential users by making our project scale-ready from the beginning.

Secondly, the underlying Google Cloud infrastructure that hosts our web application and Firebase Resources guarantees a 99.9% industry standard uptime. In addition, all services are distributed on global CDN, making the use of our application blazingly fast worldwide.

Thirdly, our reliance on a trusted external system for authentication and database services resulted in an industry level security of our system. The user details and data is protected on systems built by experienced developers. Considering the privacy and security was among the requirements of this project, we believe we achieved it and went beyond in this respect with our choice of infrastructure.

Finally, we are hosting our supergroup server on CS school machines. Although this does not offer the aforementioned benefits for in-app user activity, we believe that it was rational to opt for using the CS lab services due to modest load expected from the supergroup functionality within the federation, which would render this option adequately sufficient for all purposes and needs.

(Architecture section: 180023970)

Peer Review

Overview

The purpose for Peerbit is to offer a centralised service that hosts, exhibits academic source code whilst also aiding the review process of academic source code. The review process has been a major topic of discourse in the group as we determine what is the best way for Peerbit to follow a peer review procedure.

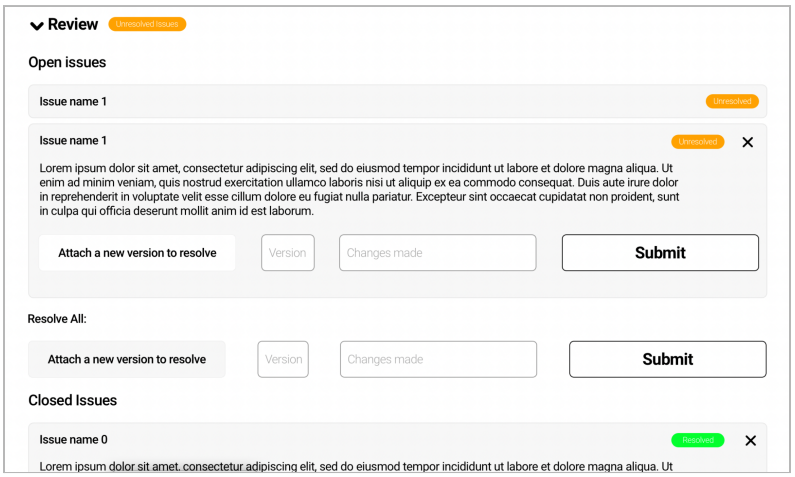
Once a submission is uploaded, a code submission must be reviewed by two distinct peerbit accounts. Each reviewer has the ability to add issues they find with the code submission. Each submission holds a review status: not reviewed, needs reviewers, under review, and reviewed. This provides the user with a simple flow of the review process, which is made clear with the app's UI.

Code submission's also have a general view page, with a description section - to present an extensive description of the source code, a code browser - to view the source code, a discussion section - that hosts comments under the code, an attachments section - which offers some other attaching documents to the source code such as readmes, pdfs, and images.

Submission and Version Control

The user can upload as many files as they like, give the submission a name, a brief description, a large description and attachments. Peerbit also gives the option if the user wants to make their code open for review or not. This can all be configured at the upload page. This was a design decision to allow all details about a submission to be set at upload for simplicity. The group agreed that given time to develop, we could allow the author to edit these fields after upload. However for the moment in the current time restrictions we decided it was best to leave it as is.

Peerbit encourages authors to upload newer versions of their submissions after issue's have been raised by reviewers. This pushes for the user to upload their best, most improved work to the platform.



In-code Comments

As a group we agreed that in-code comments would be a very useful part of the peer review process. When viewing a specific revision of a submission, users can make comments in a file on a certain line. This allows reviewers (or any user) to point out any mistakes in code or start discussions about how a section of code could be improved. To make a comment, users

need to just click on a line and then add a comment on the comment prod that comes up.When a comment is made on a line in a file, it permanently has a red mark next to the line showing there is a comment here allowing users to know which lines have comments and which don't.

We wanted to encourage discussion between users about code, and therefore we also implemented the ability for users to reply to comments. Each comment can have as many replies as possible, allowing for long discussions between users about specific issues. We did however limit the depth of replies to one, meaning that a comment can have replies, but a reply can't be replied to. We limited this as we felt the code was more important and didn't want users to get carried away in the depths of comments.

Discussions

Peerbit has a general discussion section, where any thoughts from any user who has access to the submission can be posted. Although the in-code comments is a valuable functionality for specific lines of code, the group agreed having a general discussion section as well is suited to the Peerbit's purpose.

As well as comments to the submission, user's can also 'reply' to these comments. This encourages discourse between users about the content of the submission. We decided that for this use, it would be best to give each comment a maximum reply depth of one - meaning primary comments can have child comments 'replies' however these child comments cannot have replies as well. The reasoning behind this is, although discourse of the submission is a crucial part of Peerbit, the main content is the actual submission, and not so much the comments. Limiting the child depth to one allows users to make comments, yet doesn't distract any attention away from the actual content. This follows a similar comment style as Youtube, and Facebook.

Reviewer Issues

Reviewing code is probably the most important part of the whole PeerBit application. The way we decided to approach the task of review was through the use of issues. When reviewing a submission, if a reviewer has some problems with code, they can add an issue. When posting an issue, a reviewer will add some text explaining any problems with the code. When an author of the submission amends any problems raised by the reviewer, they can close the issue by attaching a new version of their code. This new version will now become the main version for the code. Reviewers can then accept this new version or add new issues. Once it has been accepted it can be finally marked as reviewed.

Transparency

At the start of the project before development, we went through hefty market research to try and make our system better than others. One of the issues identified with other peer review systems was a lack of transparency. When someone reviews a piece of academic work in most systems, it remains hidden from everyone except the author and the reviewer themselves, almost creating a sense of secrecy. We all felt as if it was better if everyone could see any previous

reviews or issues as it might stimulate them to think of other issues or they could even help the author solve these problems. This is why in PeerBit we have a transparent approach to reviewing, meaning everyone can see any review and nothing is hidden.

Incentive Model

An incentive program is a formal scheme used to promote or encourage specific actions or behavior by a specific group of people during a defined period of time.[1]
One of Peerbit's main problems is understanding that not many people are going to willingly review someone else's work, with no good reason to. It is probably reasonable to assume that supervisor's of a postgraduate will review their work as it is part of the supervisor's job, however Peerbit requires two reviewers to review the work before it can be classed as 'Reviewed'.
To combat this, our plan is to create an incentive model for user's to review work, encouraging other people to sign up for reviews, in exchange for their personal code submissions to be shown further up on the discover page.

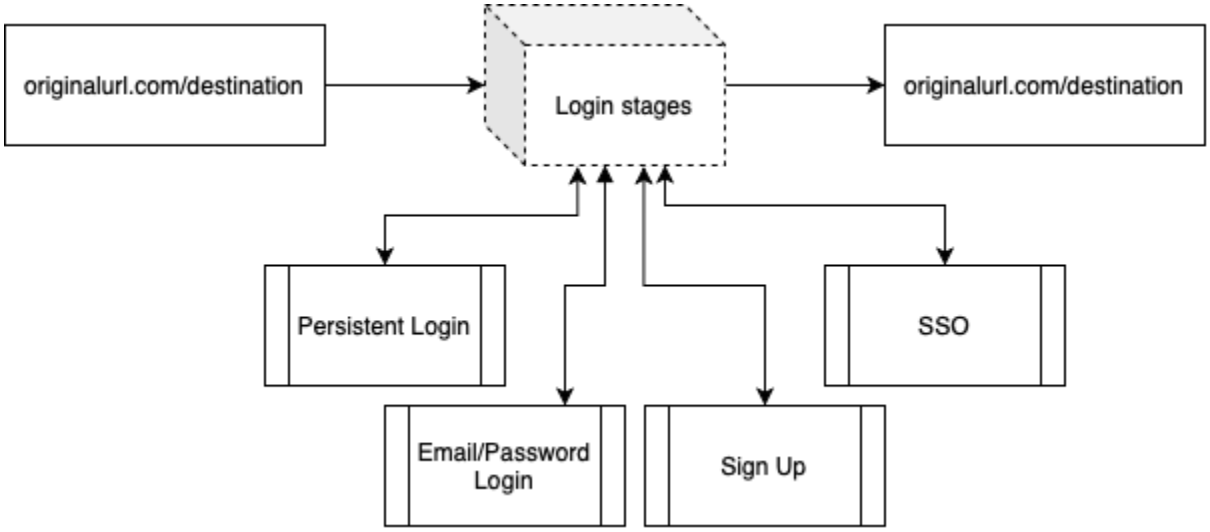
190004593 & 190018018

Other Implementation Details

Link Routing and Persistent Login

Our system makes use of persistent authentication. This means that a user that has signed in will remain signed in throughout all subsequent sessions, without needing to re-login. This and our security approach requiring users to be authenticated to access any page results in a need to handle page refreshes or subsequent logins gracefully, prioritizing user experience and code reusability. Furthermore, we want to let a user get through to their final url regardless of whether they are logged in or not, in case a link is shared with someone who might not have an account or be authenticated. To overcome this challenge our group created an elegant solution that is powered by state management library Redux. This section will go through implementation details and explain how our solution functions.

The main goal of this reusable system is to let the user through to their final destination through any login pages / processes. Intuitively, our decision was to store the original queried URL, let the user authenticate / be automatically authenticated and then once they are logged in - redirect to their destination. This would allow our solution to be agnostic of the destinations, origins and authentication state, therefore making it flexible to any situation and designed for reusability. The process is demonstrated in the diagram below:



The main powering mechanism that would allow for clean code and access to this functionality from any page is a Redux state store. Redux allows to store values that are synced with the UI for changes and make them accessible from anywhere in the app. This is a global equivalent of React state variables. Our choice of Redux allowed us to escape dependency injection that comes with managing local state in favor of React hooks that query the “store”. As a result, our code is much more readable.

Although the benefit of abandoning dependency injection was discussed as a positive aspect of this solution, it does not apply to our team's implementation of other functionality. We are using global state only for this solution because it would be inherently difficult to pass and manage state locally. For all other solutions we followed good practice of local controlled dependency injection, as it is generally argued that global state may lead to difficult code to debug should state change issues arise.

180023970

Database Optimisation

For our MVP deliverable, the database worked perfectly fine and loaded submissions and users in good time. We were happy with this at the time, however for the final submission we realised we could have benefited from some database optimisations, so we overhauled how we stored all our data.

Firebase's Firestore database stores data using the NoSQL format, which is highly different to the relational format that we are used to. NoSQL stores data in a JSON like format with objects containing properties. Unfortunately, during the first semester we structured our database in a format which would suit a relational database, however this was not appropriate for NoSQL. We initially were storing each submission/user in an individual document within a single collection. This all worked well for reading the details for an individual submission/user, involving just one read. However, when it came to loading all submissions in the discover page it was highly inefficient as it would have to read as many submissions (and their authors) as there were in the database. Firebase also is a pay as you go model and has a free quota of 50,000 reads, 20,000 deletes and writes per day. This seems like a lot but in our old model, there were potentially upwards of 100 reads (and this number could get much larger if more submissions and users were added) every time the discover page was loaded, so this limit could easily be reached in normal usage. Our old model used O(2n) reads every time the discover page was loaded, where n is the number of submissions in the database, and this is not sustainable.

To overcome these issues, we used what are called teasers. All the teasers were located in a different collection to the usual submission data and had its own document. In this document there was a single array of objects, of which each were linked to one submission. In these objects, only the details of the submission relevant to the discover page was stored: name, shortDesc, posterUserID. The object also kept a reference of the objects id in the actual submission database, for when we wanted to access the full submissions data. This meant that when the discover page was loaded, we only needed to read this teaser array, which means only 2 reads, a drastic improvement from the MVP database model. When a new submission/user was added to the database we add the document as was before, however we also add an entry to the teaser array.

This drastic reduction from 2n reads to just 1 read has helped fully optimise our database and means we won't be at risk of hitting that free quota of 50,000 reads. It is however worth noting that Firestore databases only store 4,000 entries, so when we get to this many users or submissions, we will have to create a new teaser array.

190004593

Security

We decided to use Google Firebase as a NoSQL database to store Peerbit's account data, source code files, and their metadata. All of this data is therefore stored on an external server. Furthermore, we used Firebase's hosting service to host our web application itself on their servers. However, our primary concern with security was with regards to user registration and login, and, once again, we utilised Firebase via their authentication service. To log in to the app, a user enters his or her authentication credentials, which for Peerbit is an email and a password, that are passed to the Firebase Authentication SDK. Firebase's secure backend services will verify the credentials before returning a response to the client. Passwords are never stored in any of the services we wrote the code for, and login persistence and tokenization is implemented by the service, by people who have a much greater expertise in such. Additionally, we make use of a user's unique identification token assigned by Firebase to verify the identity of a user and whether they can make changes to our backend data by submitting code or updating their user details, and that those changes are implemented in the correct location within the backend database. The implementation of these services and processes is handled by experts with far greater experience in web-application security than any of us.

190018240

Scalability

At the beginning of our development, although it wasn't necessarily a specification assigned to our project, the group agreed that it was crucial that Peerbit was scalable. How easy it is for Peerbit to host thousands of users with hundreds of submissions each, is very important for any modern day application.
In an extreme scenario of scaling required, we could look at how well Zoom scaled when the entire world turned to work at home overnight. Zoom has said they had a 2000% increase in daily users between December 2019 and March 2020[2].
Although it is unlikely Peerbit will ever require that level of scalability, it is still worth creating an app that can scale well for its potential users. We decided for this level of scalability to be possible, we would need to use cloud services, which enables startup companies to have this ability to increase their capacity.
Following in this manner, the group decided to use Google's firebase, to allow us to scale with the amount of users we have. Firebase offers a tiered price plan that is dependent on read/write counts etc. This works well for Peerbit because for the moment we won't be reaching the upper limit of the free tier, and so, all our hosting, functions and API's are run through firebase for free. That said, if we ever did reach the limit, our app is capable of increasing the capacity to allow for any traffic hike.

190018018

Evaluation and Critical Appraisal:

Overview

Our academic code publishing platform, Peerbit, achieves all of the core requirements set out by practical specifications and most of our initial user stories. We built a clean and efficient web-based user interface which allows for secure account creation and management, discovery of peer-reviewed code submissions, and a protocol for sharing users and code between journals in the supergroup. We also managed to expand upon these features with additional functionality such as a Review page to implement our multistage peer-reviewing process, refactoring of both our backend structure and user interface to increase efficiency, and an expansion of our comments into a dedicated discussion section. There were some other features which we had planned which never materialised but we feel the final product is to an exceedingly high standard.

During our development of Peerbit, we aimed to build a platform with code review aspects comparable to that of similar work available in the public domain, such as Github. Github is the standard for code hosting and review platforms and includes many of the features we implemented. Users are able to easily search for and filter code submissions, add comments and reply to others in a discussion section, make suggestions for and approve revisions of code during a review process, and create a user account which can be edited at any point. While not as fleshed out, many of Peerbit's design decisions drew inspiration from websites like Github, such as the expansion of our line-specific commenting system into a dedicated discussion section.

Finally, during development, we made good use of the scrum methodology to keep ourselves organised and on track in meeting our goals and deadlines. We encountered difficulty when several group members fell ill with coronavirus and other illnesses at different points throughout development. Additionally, another group member experienced difficulties at home which required his attention. However, we largely managed to keep to our week long sprint cycles, which, coupled with regular group and supergroup meetings, allowed us to create a good final product. We reached this conclusion after evaluating Peerbit using integration testing and walkthroughs of the various features of our site. We created example users, some with profile pictures, workplaces, and a position, and some without. We edited those users and verified changes were made in our Firestore database. We uploaded code files and found them on both the Discovery and Account pages. We simulated a peer-review of those sample code submissions to ensure code versioning, commenting, and suggesting/approving edits all works correctly. These tests are detailed in full in the appendices of this report.

190018240

Original Objectives

The set of core requirements set by the project specifications included:

- The ability to register new users
- The ability to post new source code files
- The ability to post comments on source code files
- A simple, web-based interface
- An implementation of a protocol for sharing user information and migrating content between coding journals

As part of the planning for our application, we incorporated these features, in addition to several of our own, in a set of user stories detailing what we envisioned Peerbit being capable of, these include:

- A user can sign up to PeerBit
- A user can sign in to PeerBit
- A user can reset their password
- A user can see their personal account
- A user can edit their personal account
- A user can publish code submissions
- A user can save a draft of a submission
- A user can see a list of their submissions
- A user can view a code submission page (minimum for editing, no code rendering display)
- A user can edit details of their code submission
- A user can delete their code submission
- A user can update their code with a new version
- A user can see the code submission view with a code and folder browser
- A user can navigate through versions of code in the submission
- A user can download code submissions as zip
- A user can see and attach comments to code locations
- A user can see, start, and participate in general discussions
- A user can attach images or files to comments in code and general discussions
- A user can sign into other journals via our journal
- A user can sign into our journal via other journals
- A user can export a submission to another journal
- Our journal can accept an incoming submission from another journal to an existing user
- Our journal can accept an incoming submission from another journal to a not yet registered user
- A user can submit a request for their code to be peer reviewed with parameters selected
- An editor can select a submission to review
- An editor can create issues to be resolved
- A user can see the review progress item, with current stage, issues by each of the reviewers
- A user can submit a new version of code and select which of the issues were addressed, as well as attach a comments

- Editor(s) can review newly submitted versions and approve/decline + comment
- User gets notified via email on review progression
- A submission that is undergoing a review has a respective public badge
- A submission that has been approved by reviewers and journal admin gets a “peer reviewed” badge
- A user gets the feed of all code submissions
- A user can filter for submissions that are peer reviewed
- A user can search for submissions or authors via indexed search bar

190018240

Achieved Outcome

As with any lengthy development cycle, modifications, additions, and some cuts to an initial plan are almost inevitable. We completed most of our user stories with some minor changes, these include:

- A user can sign up to PeerBit
- A user can sign in to PeerBit
- A user can reset their password
- A user can see their personal account
- A user can edit their personal account
- A user can publish code submissions
- A user can see a list of their submissions
- A user can view a code submission page (minimum for editing, no code rendering display)
- A user can update their code with a new version
- A user can see the code submission view with a code browser
- A user can navigate through versions of code in the submission
- A user can download code submissions as zip
- A user can see and attach comments to code locations
- A user can see, start, and participate in general discussions
- A user can attach images or files to comments in code and general discussions
- A user can sign into other journals via our journal
- A user can sign into our journal via other journals
- A user can export a submission to another journal
- Our journal can accept an incoming submission from another journal to an existing user
- Our journal can accept an incoming submission from another journal to a not yet registered user
- A user can submit a request for their code to be peer reviewed with parameters selected
- An editor can select a submission to review
- An editor can create issues to be resolved
- A user can see the review progress item, with current stage, issues by each of the reviewers
- A user can submit a new version of code and select which of the issues were addressed, as well as attach a comments
- Editor(s) can review newly submitted versions and approve/decline + comment
- A submission that has been approved by reviewers and journal admin gets a “peer reviewed” badge
- A user gets the feed of all code submissions

Due to time constraints, the majority of our focus was dedicated to our core functionality, so we decided to cut features we felt were superfluous such as saving and filtering code on the Discover page or user control over when to release a code submission to the public after uploading it. These incomplete user stories include:

- A user can save a draft of a submission
- A user can edit details of their code submission

- A user can delete their code submission
- User gets notified via email on review progression
- A submission that is undergoing a review has a respective public badge
- A user can filter for submissions that are peer reviewed
- A user can search for submissions or authors via indexed search bar

Positive changes made to our initial plans and MVP were the refactoring of our code for backend storage access and management, a rework of our user interface to increase efficiency, the expansion of features like comments, and the creation of a framework for peer-reviewing code submissions.

190018240

Conclusions

In our interconnected, digital age, the ability to share information and collaborate is crucial to the betterment of society and advancement of technology. Our goal for this project was to create a platform that contributes to this ability to share reliable and pioneering information. Fundamentally, users of Peerbit are able to publish their academic code and receive feedback from their peers during a multistage review process. This is accomplished within a clean and efficient web-based application built using third-party resources like ReactJS and Google Firebase. Our functionality includes the ability to register new users, change user details, find new code submissions, see, add, and reply to comments in a submission's discussion section, either give or receive feedback on code during peer-review, and communicate with other journals within the supergroup federation. Using functional testing, we ensured the final product is of high quality by performing walkthroughs of the core features listed above and found no significant issues. Some of the user stories we had initially created were cut or modified during development, but this was because we prioritised the core requirements set by the project specifications. For example, we did not include the ability to save a draft of or edit an uploaded code file, or search/filter submissions on the Discover page. While these would be beneficial quality-of-life additions, they are not essential to producing a great final product.

We would not have been able to build Peerbit to the standard we did without utilising the scrum development methodology. Our weekly and bi-weekly sprint cycles significantly assisted in keeping us on track to complete our goals by set deadlines. Furthermore, constant communication via messenger applications like Signal and Teams and weekly group and supergroup meetings made sure everyone was able to solve any issues that arose in a timely manner. Scrum also helped alleviate the disruptions caused by repeated periods of illness or other problems throughout the year, which were numerous.

Looking to the future, we envision several possible improvements and directions which we could take Peerbit. An obvious step would be to add the features which we cut due to time constraints to better reflect similar platforms like Github. A search bar, filtering submissions, uploading zip files, and code visibility options would be a good start. In the long term, however, we think Peerbit could be tailored to act as a platform for interdisciplinary academic code. There would be multiple sections dedicated to peer-reviewed work for different fields like mathematics, physics, chemistry, or computer science. In this situation, our platform would act as a Github for academia. We feel a concentrated hub of academic work would be hugely beneficial and further our overarching goal of facilitating the sharing and advancement of research.

190018240

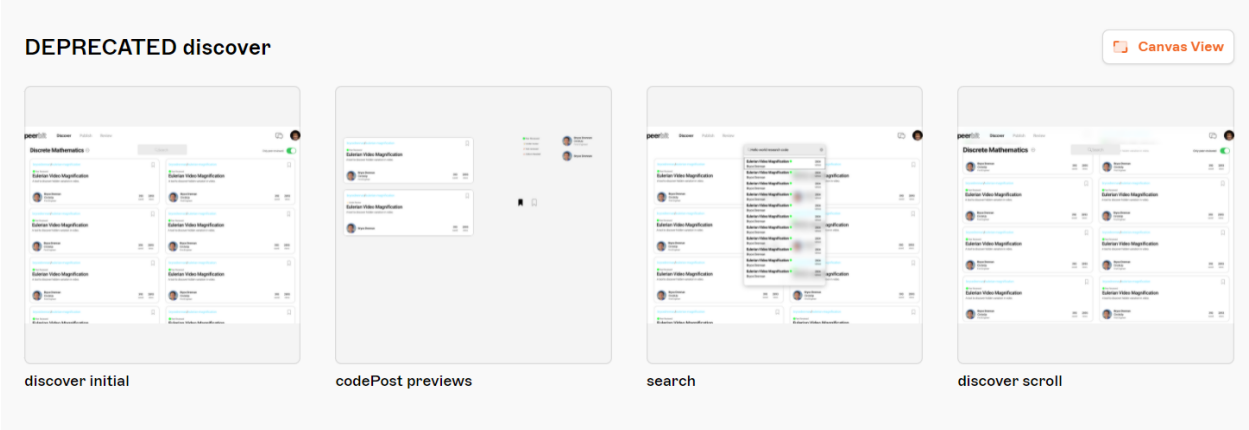
Appendices

UI Design

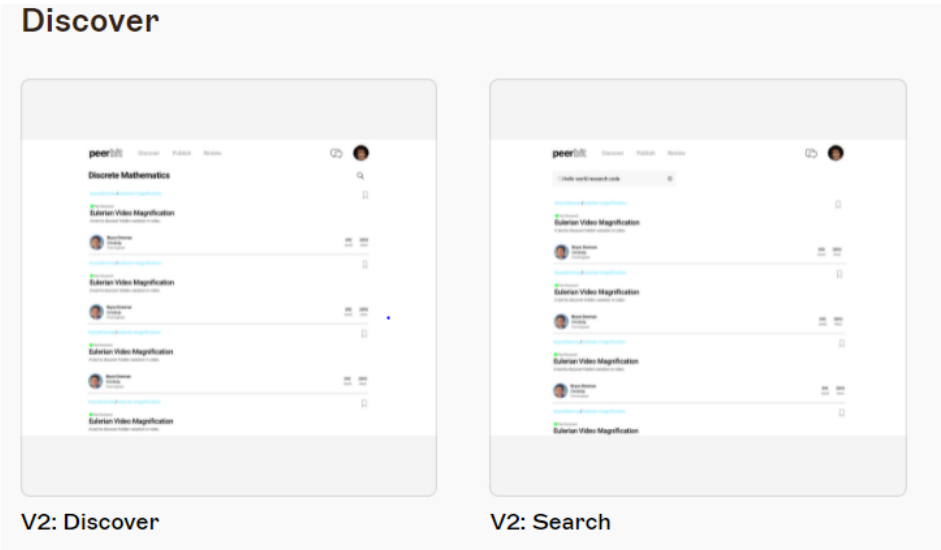
Our team has greatly prioritized UI and UX design in our project. We believe that usability and the experience of using the app or website is equally as important as the raw functionality it provides, as without intuitive UX the user would simply not be able to find the feature to use.

To design our UI we predominantly used the sketch application. In Sketch we were able reuse components and plan out pages via a very user friendly user interface. You can access our sketch UI designs via this link:
https://www.sketch.com/s/7f6644ff-b62e-4d77-9ec6-5c718d9987da

After the MVP we realised a few flaws in our UI design and therefore, for the final submission we decided an overhaul of the UI was necessary. In order to account for this we changed all the old UI designs on sketch to have Deprecated in front of the name, for example the old discover UI in sketch is shown below:



Any pages for the new format will have a V2 before the name, for example with discover again:



I will now discuss the design decisions in the new UI design vs the old UI design (Please refer to the sketch page provided by the link to see the comparisons).

Discover page

Initially we showed the discover page as 2 columns containing all of the relevant submissions. However, we felt as if this was a little bit too cluttered for our liking, as there were so many submissions on your screen at one time. We therefore decided to change it to just a single column of code submissions. The page is also reactive to changes in window size, so the components are never too wide, which would make it difficult to read. This new format makes it a lot easier to process the data and find the submission you want to.

Codeview page

Initially the code view page put the comments to the right of the submission in a separate tab. This however got extremely messy when we had lots of comments with lots of replies. This also didn't work with how we wanted to implement the review functionality. In order to overcome this we decided to put comments below the line in which they are being added on. This works nicely with our review functionality and also looks a lot less messy with lots of comments

Account page

The account originally looked all nice and worked fine, however we all agreed that when we added more fields it looked a little too long as everything was in one column. In order to overcome this issue, we separated the profile picture to the left of the user details. This not only made better use of the page, but it also allowed us to make the profile picture bigger.

Publish page

We all agreed the publish page had a similar problem to the account page with the idea it was too long being in one column. To overcome this we moved the submission file to the left and made collapsable fields in order to make better use of the space on the page. The submission form is now a lot more user friendly and it is easier to work out what fields to fill out

Others

All of the other pages were new to the application, therefore did not involve any changes. You can see the UI design for these pages via the sketch link. The only pages which we kept the same as MVP were the login and register pages as they already were user friendly and worked fine.

180023970

Testing

Functional testing is the process by which software developers determine if a piece of software is acting in accordance with predetermined requirements. Ideally, it uses black-box testing techniques, in which the tester has no knowledge of internal system logic.
Process for functional testing:

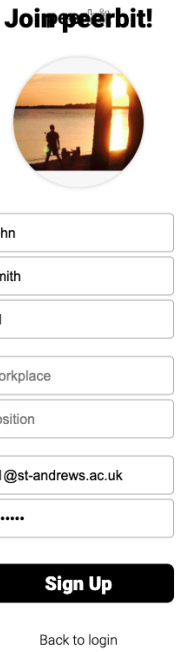
1. Determine which functionality of the product needs to be tested. This can vary from testing main functions, messages, error conditions and/or product usability.
2. Create input data for functionalities to be tested according to specified requirements.
3. Determine acceptable output parameters according to specified requirements.
4. Execute test cases.
5. Compare actual output from the test with the predetermined output values. This reveals if the system is working as expected.

Functional testing techniques include:

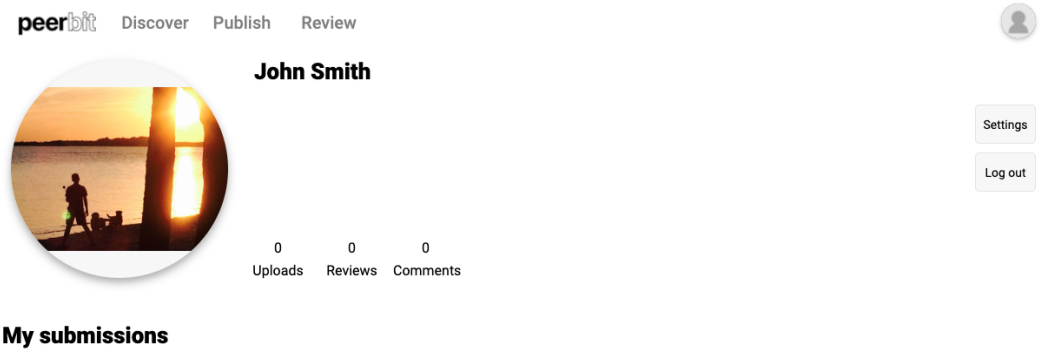
- End-user based tests: Test the system to see if components are working correctly in combination. An example could entail the code journal loading, the user entering credentials, being directed to the home page, reviewing code entries, and logging out.
- Equivalence tests: Test data is segregated into partitions called equivalence data cases which must respond in the same way. Example: Files outside the approved file types should all be rejected in the code submission form
- Boundary value tests: Checks system behaviour when data limits are implemented. Example: Passwords less than the required number of characters should be rejected during account creation
- Decision-based tests: Check for system outcomes when a particular condition is met. Example: Changing user details should return them to their updated account page, incorrect login credentials should reload the login page.
- Ad-hoc tests: Tests targeted toward breaking the system and checking its response. Example: Code reviewer submits comment to a code file that is deleted by an administrator.

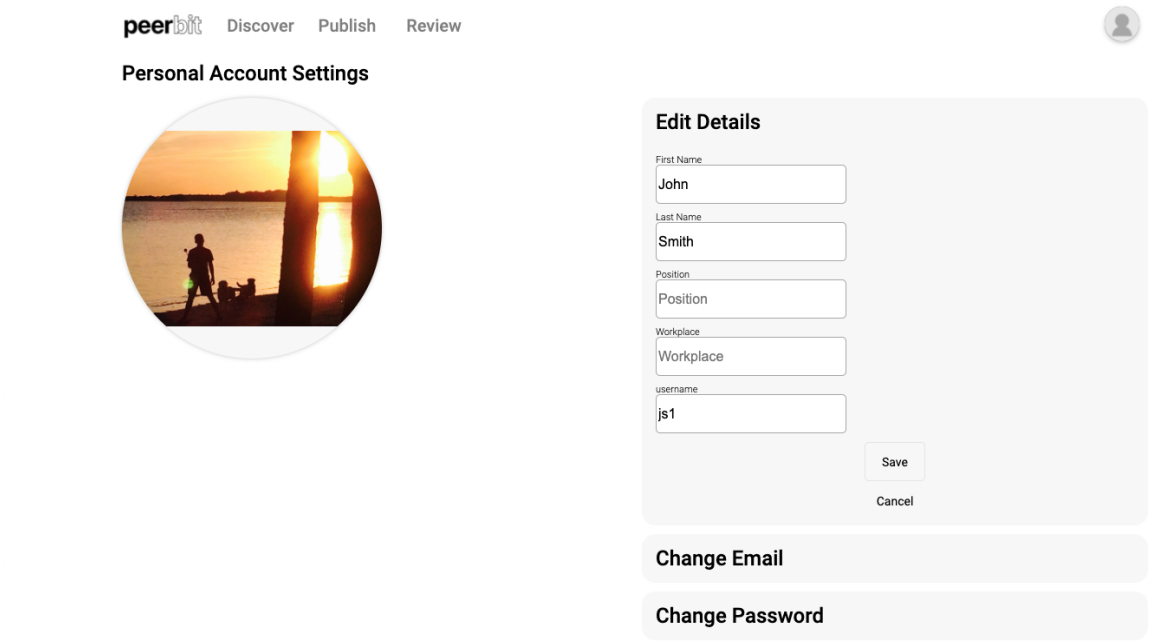
<u>User Registration:</u>

For walkthrough testing of Peerbit's user registration, I created an example user called John Smith, he is a student at university so he does not have a workplace or position as these are optional fields.

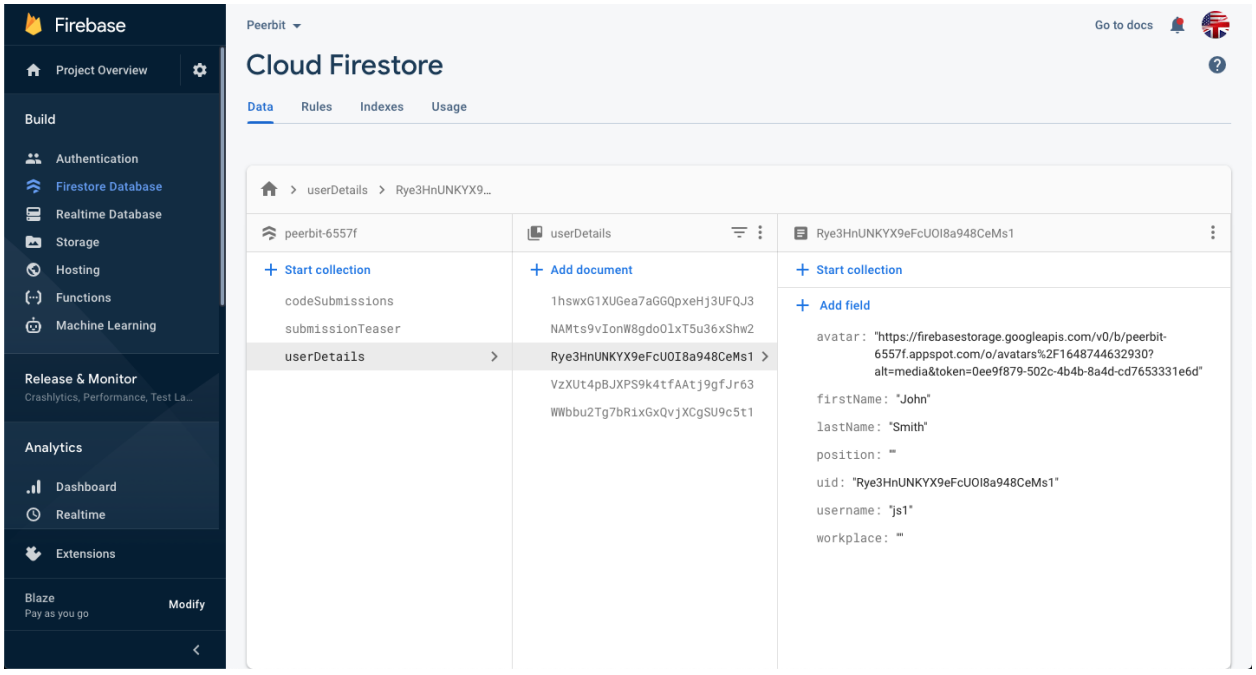


After clicking the sign up button, the user has been successfully registered and their details will appear on the Account page and Account Settings page if they wish to change them.

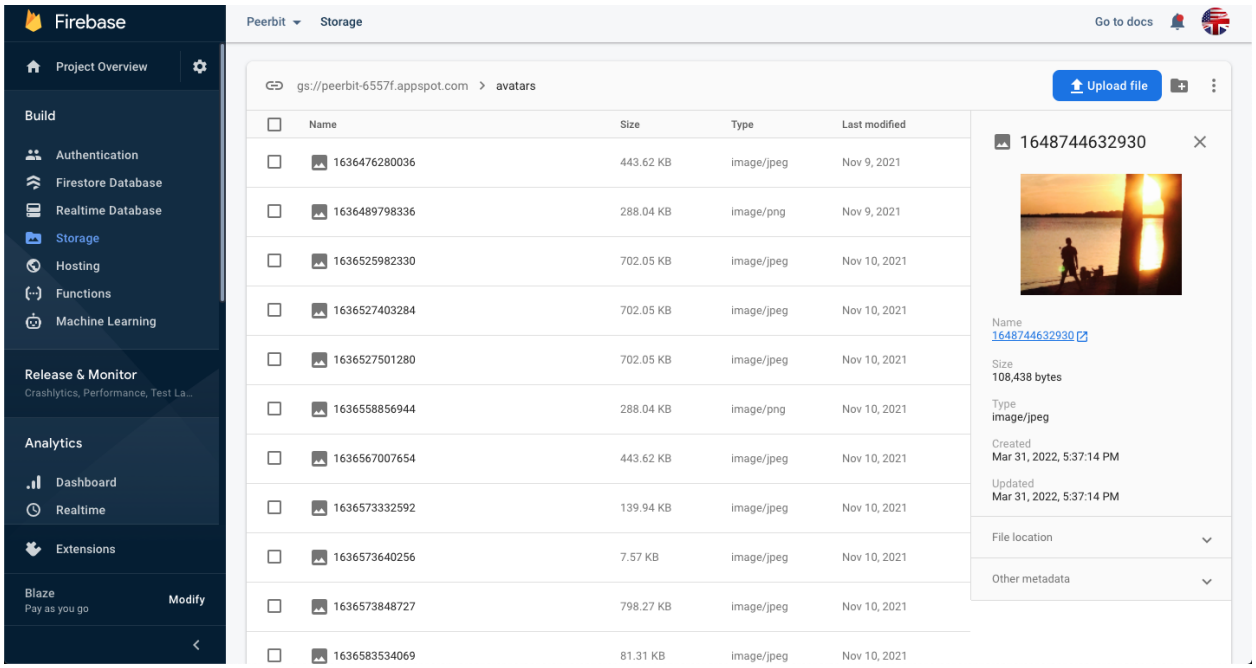




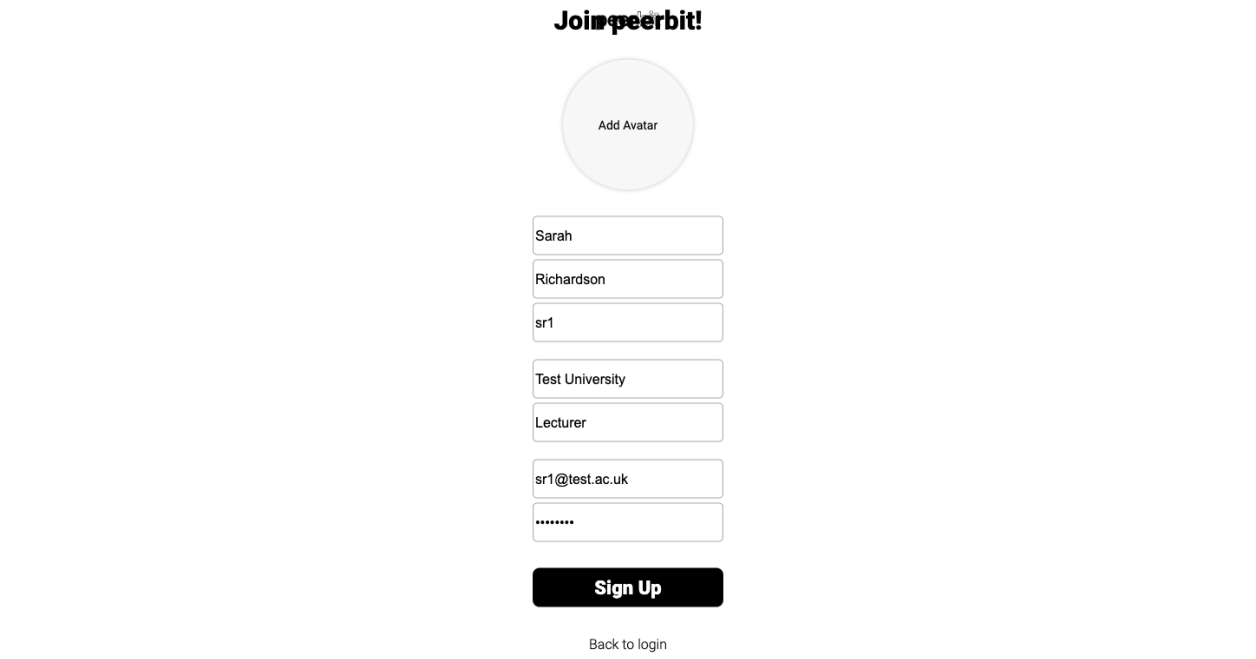
Additionally, to demonstrate how this information is stored and retrieved when the page is rendered, we have included the corresponding user document from Firebase's Firestore Database where we store user details and code submissions.



The uploaded profile picture itself is located in Firebase's Storage, the url is stored in Firestore.

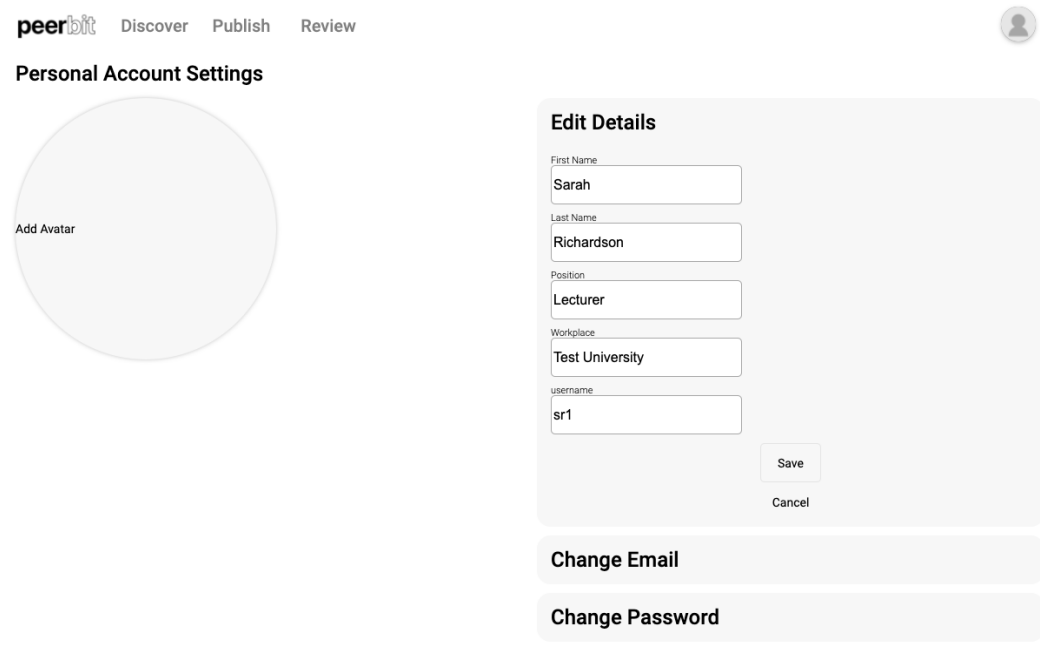


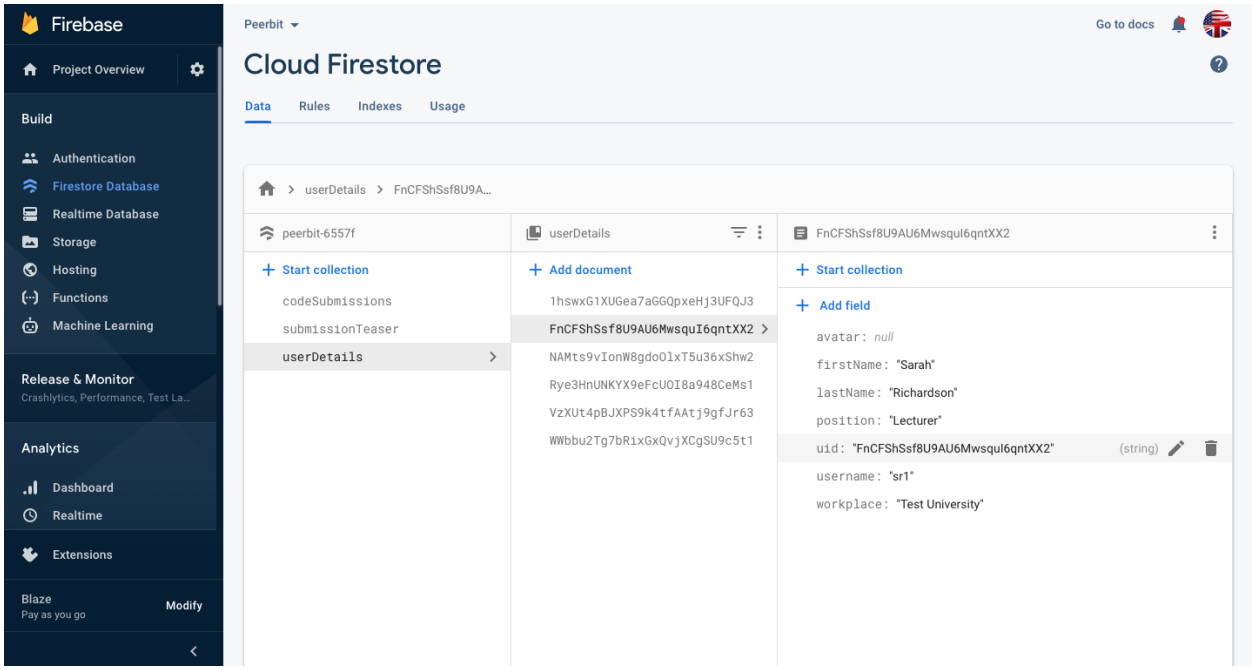
Next, to further demonstrate the optional user fields, I created a second user called Sarah Richardson who does have a workplace and position since she is a lecturer at Test University. Also note Sarah does not have a profile picture as that is also optional and can default to a placeholder image.



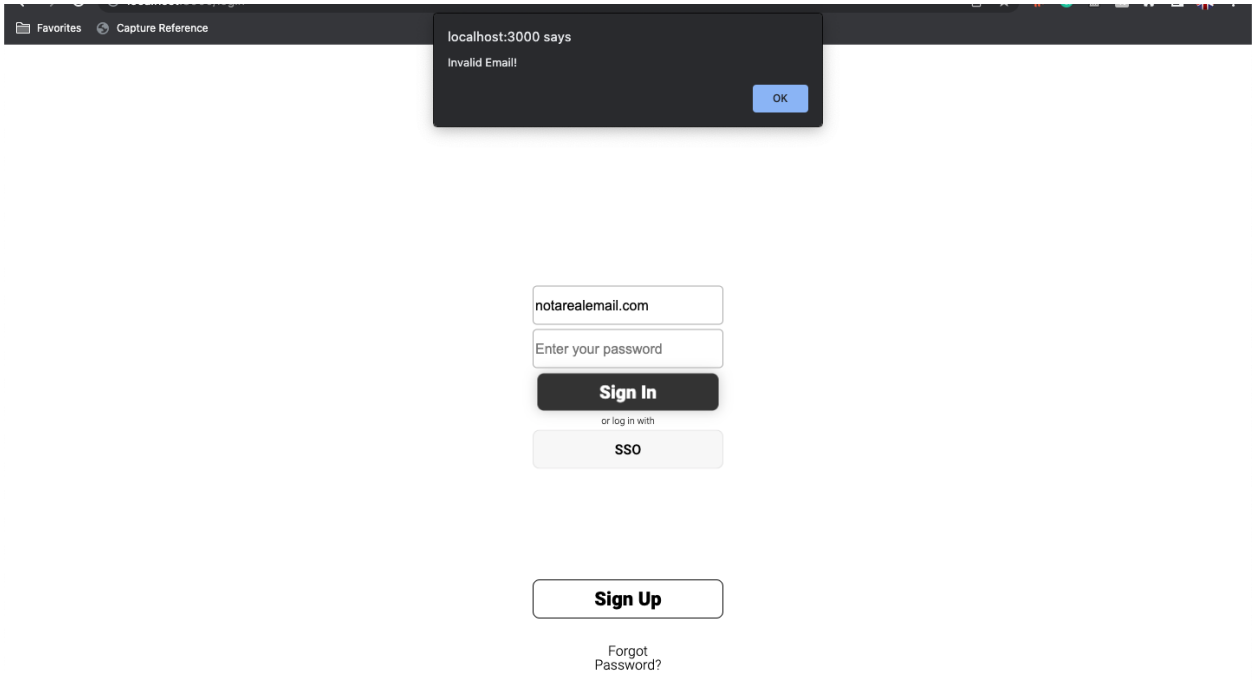
Once again, these details are then displayed on the Account and Account Settings pages and are added to the Firestore Database.

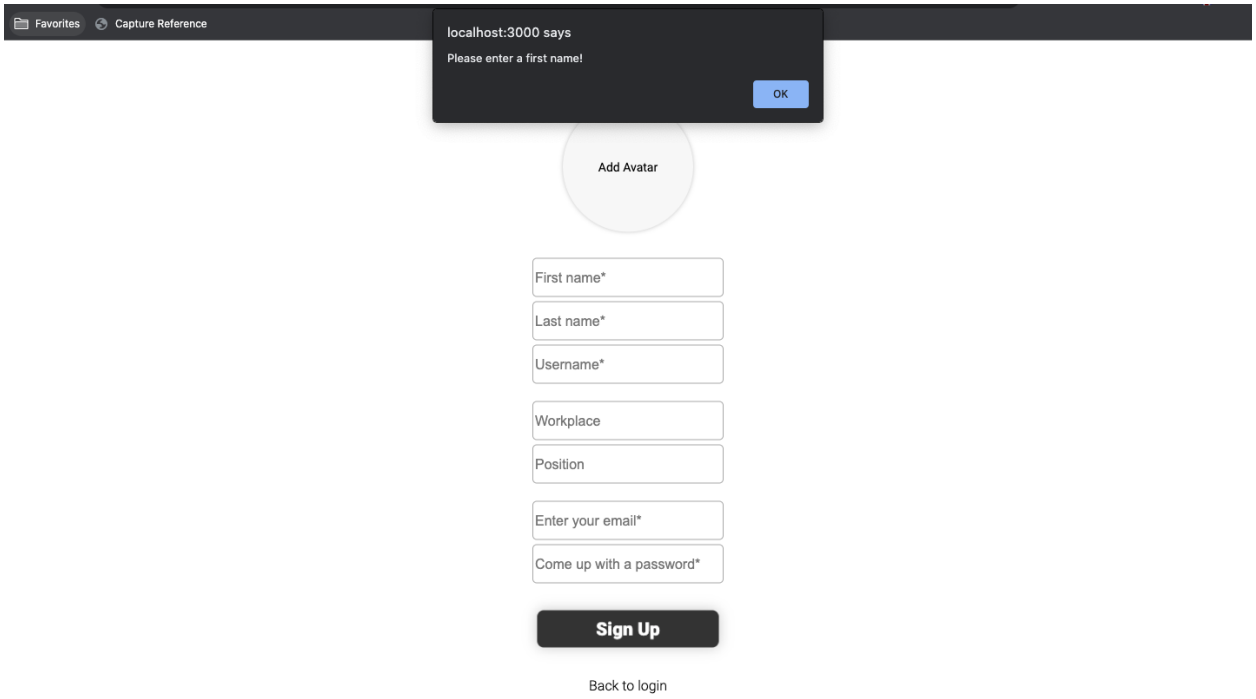

peerbit
Discover
Publish
Review
Sarah Richardson
Test University
Lecturer
Settings
Log out
0
Uploads
0
Reviews
0
Comments
My submissions


peerbit
Discover
Publish
Review
Personal Account Settings
Add Avatar
Edit Details
First Name
Sarah
Last Name
Richardson
Position
Lecturer
Workplace
Test University
username
sr1
Save
Cancel
Change Email
Change Password



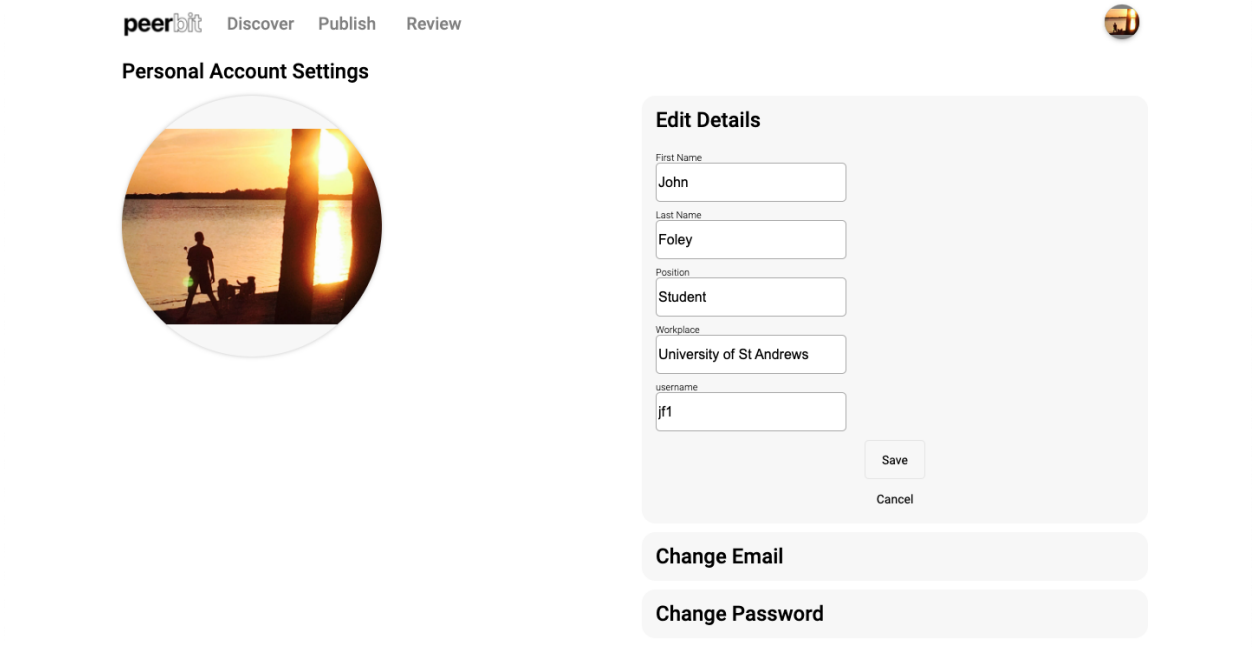
We implemented pop-up error messages when an invalid email or password is entered during log in or registration, as well as when a required field like first name, last name, or username is missing.



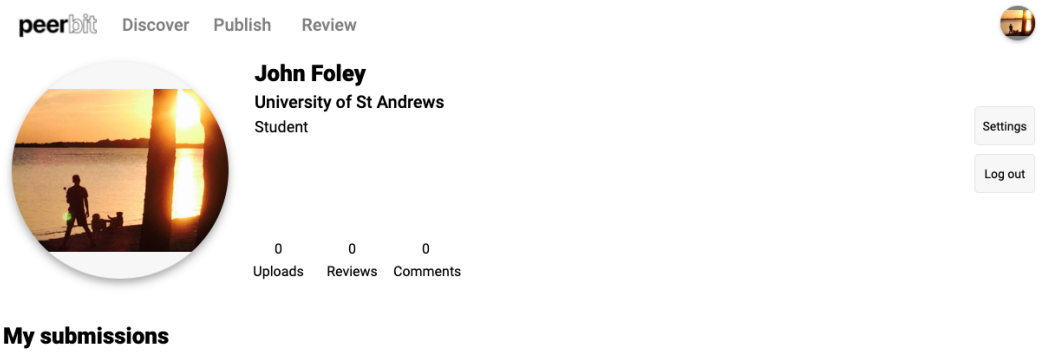


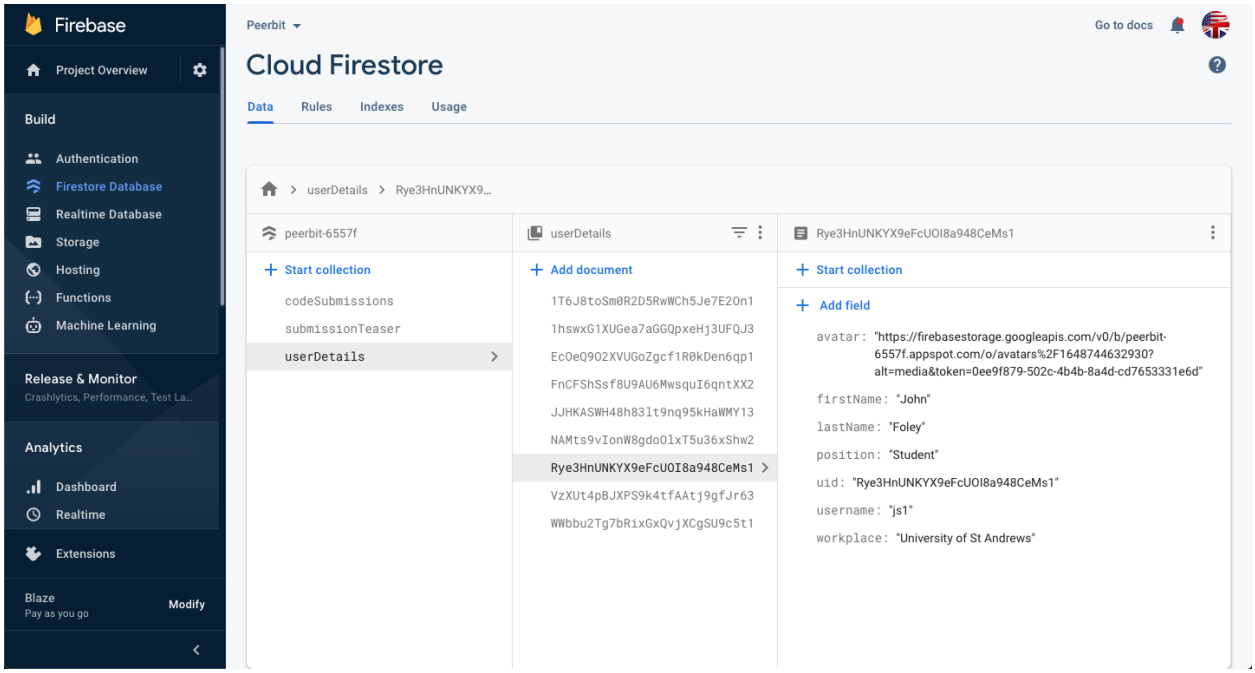
Updating User Details:

Using our previous example user, John Smith, we can change his last name, username, and add entries for his previously null position and workplace.

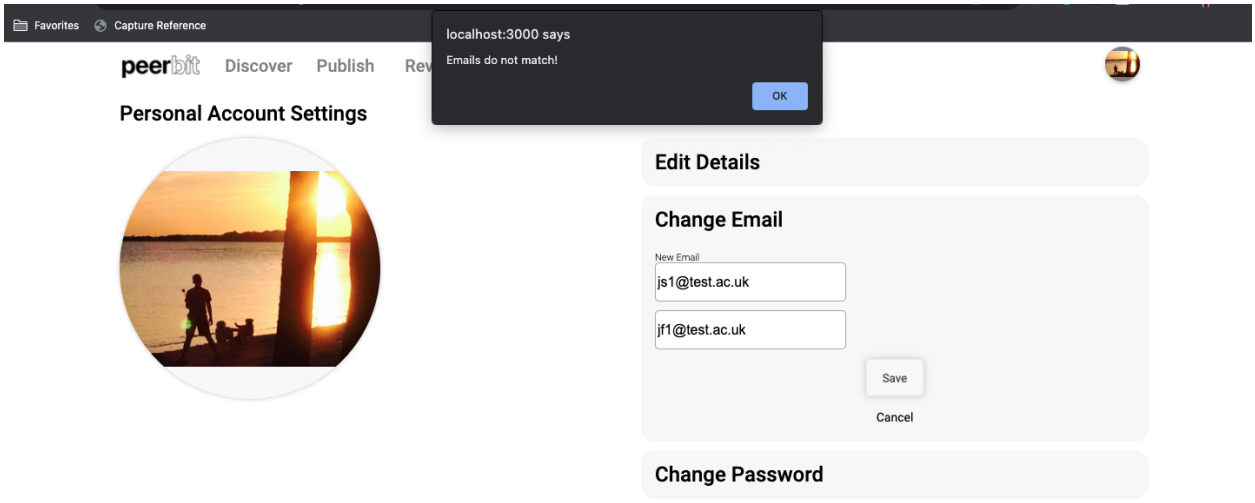


These changes are then reflected on the Account page and in the Firestore Database.



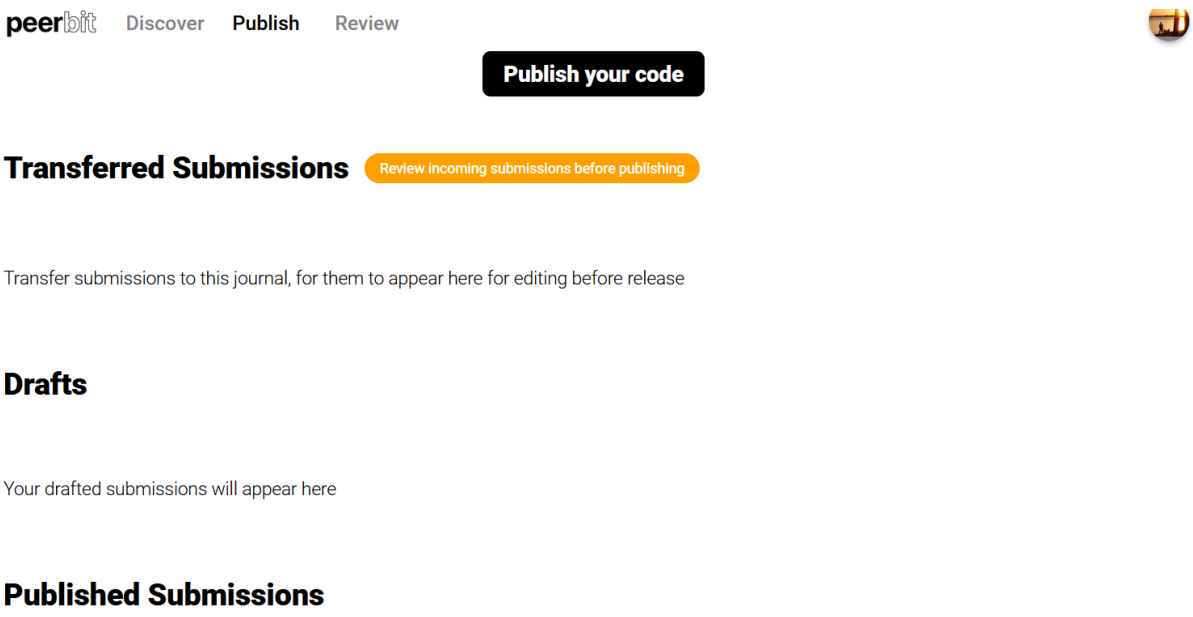


We also included error messages when the emails does not match while attempting to change a user email.

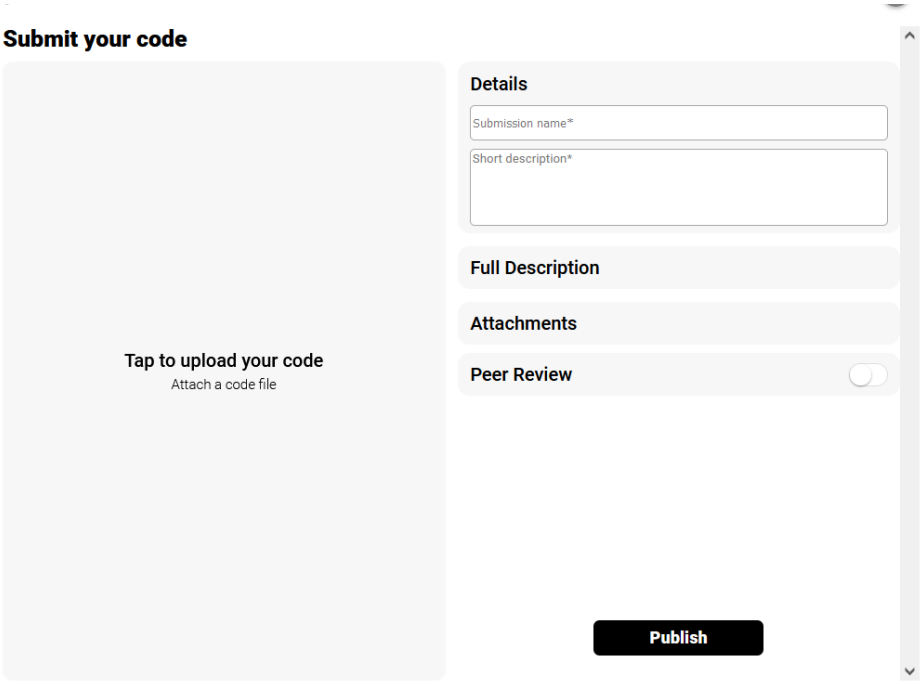


Posting a Submission:

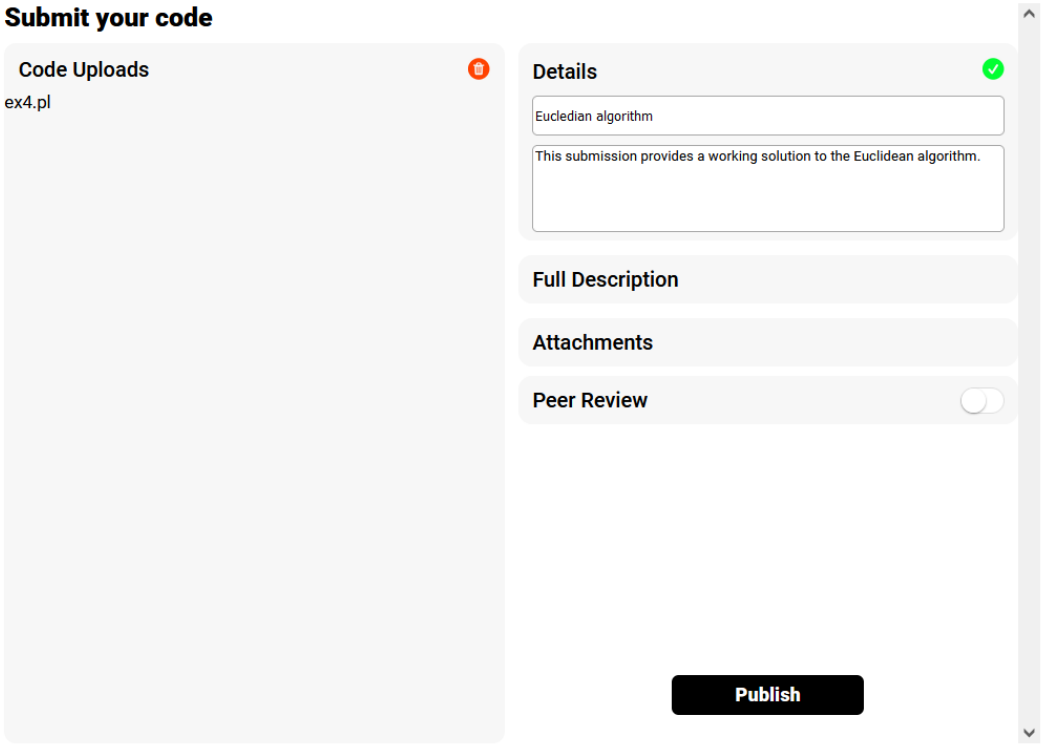
We will continue our example with the John Foley account, and to start we will navigate to the publish page.



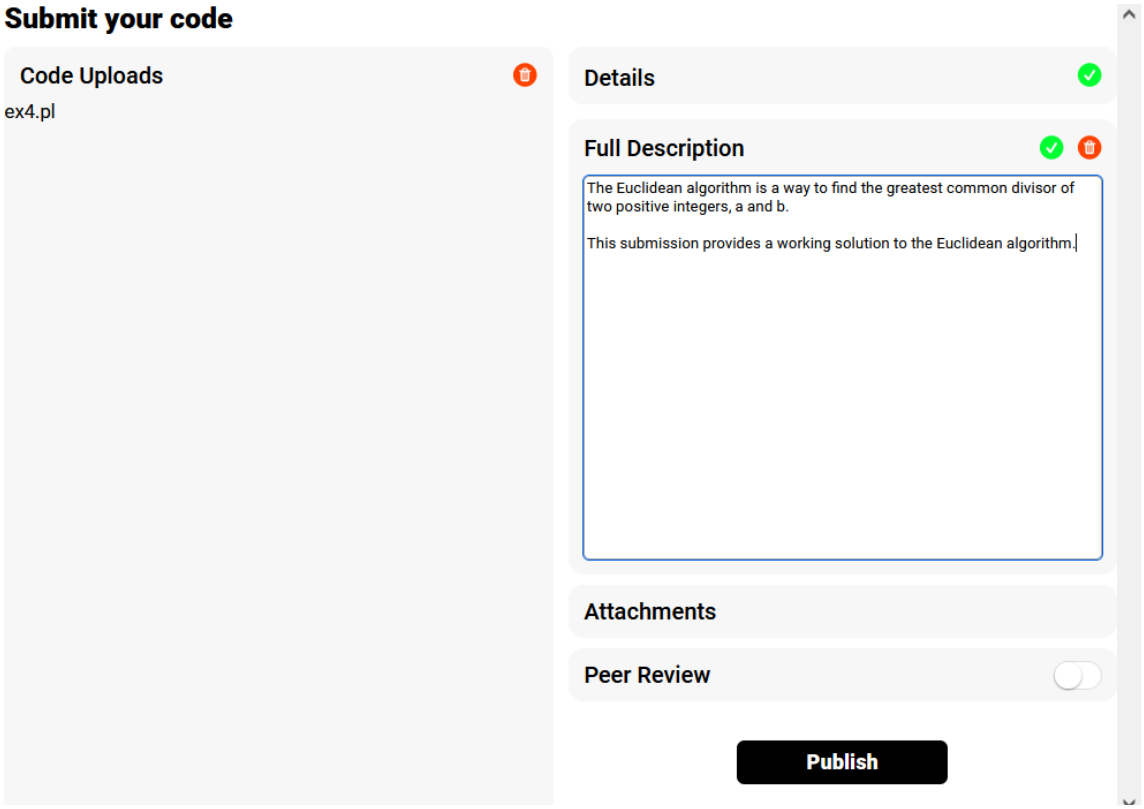
As you can see there are no submissions under the published submissions title.
If we click the Publish your code button, we get to the publish form page.

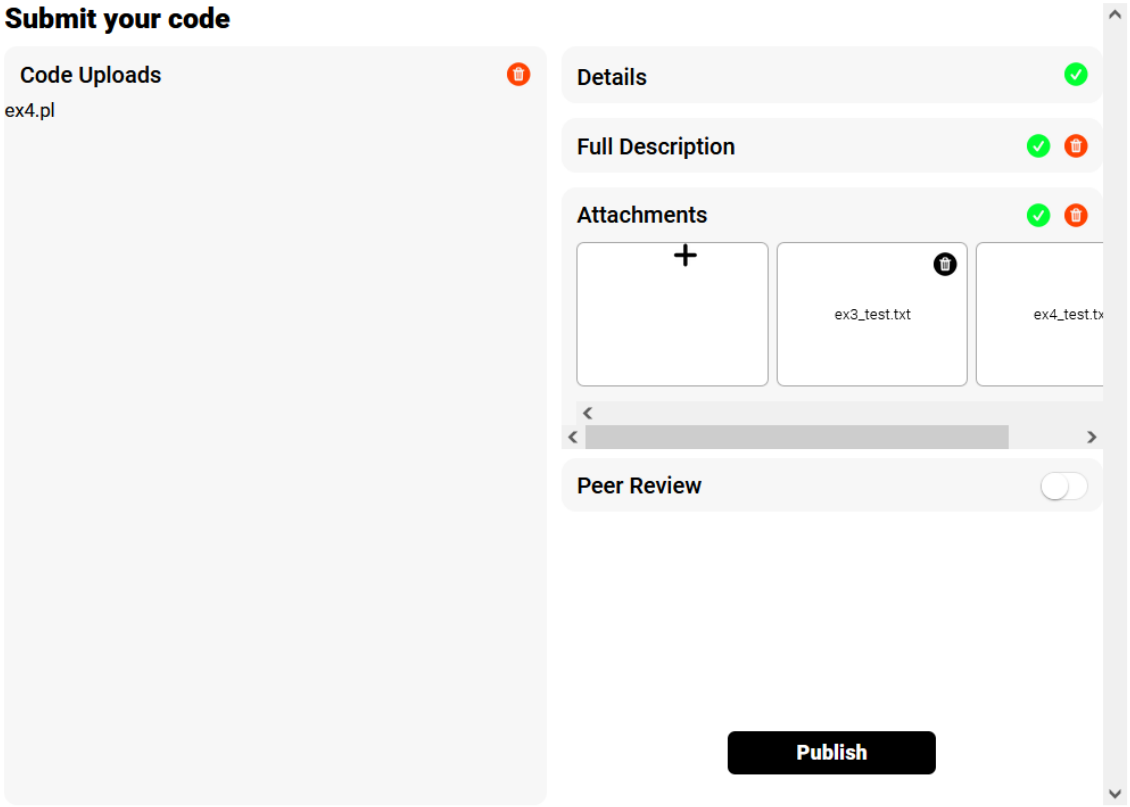


You can now fill out the form with the submission details and attach your code file from your directory.

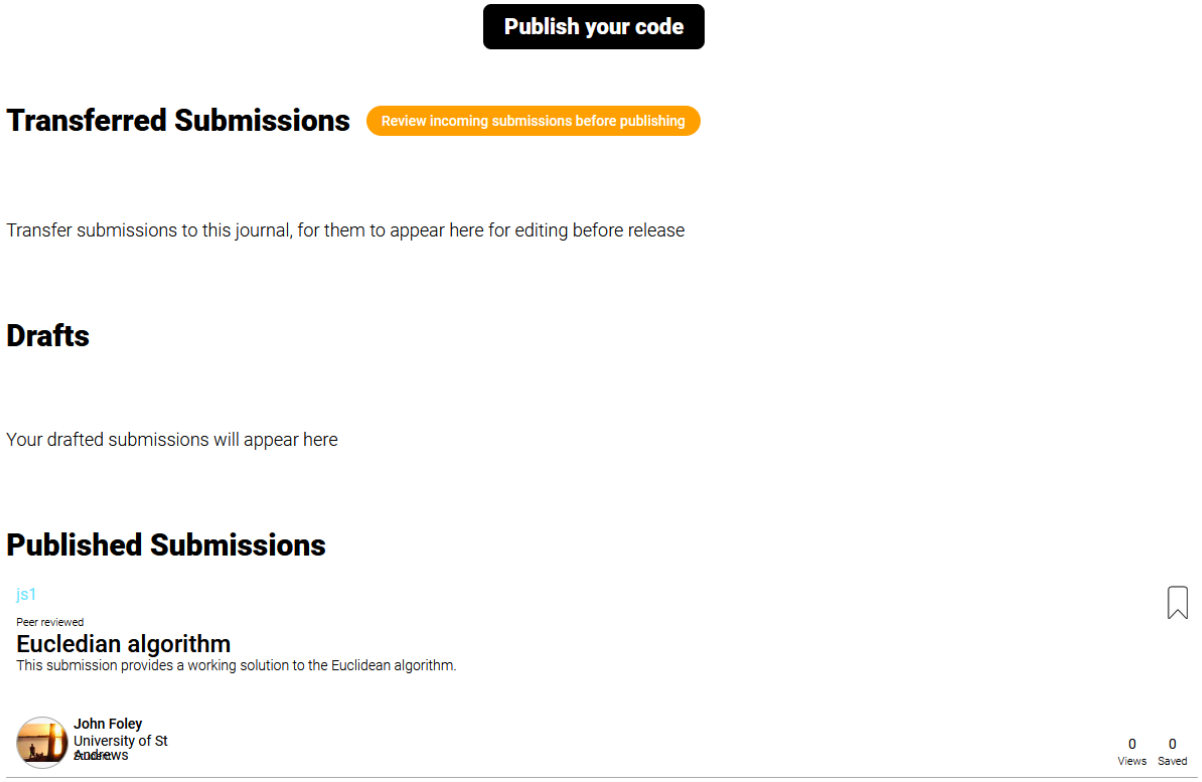


You can also add a Full description and/or an attachment

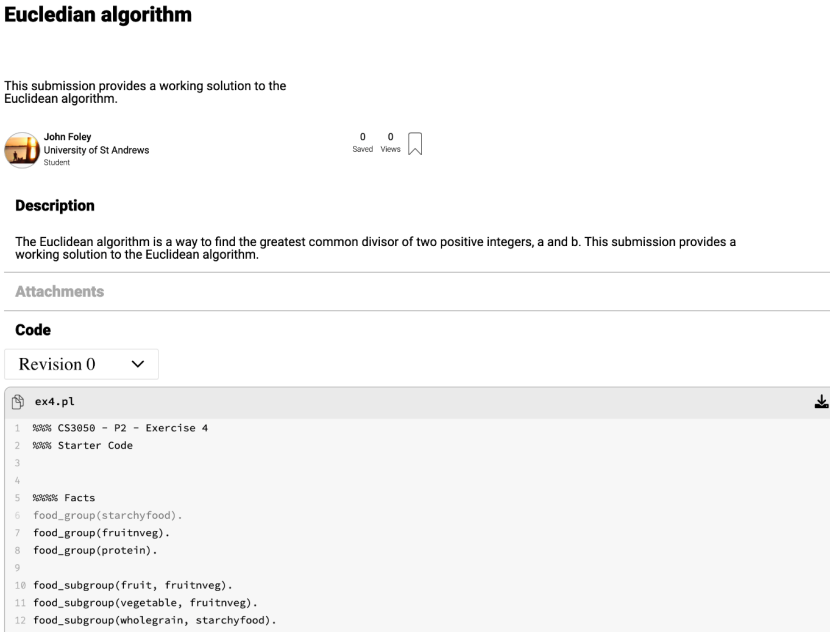




When we click publish we are sent back to the publish page and you can see your submission has been added under your submission

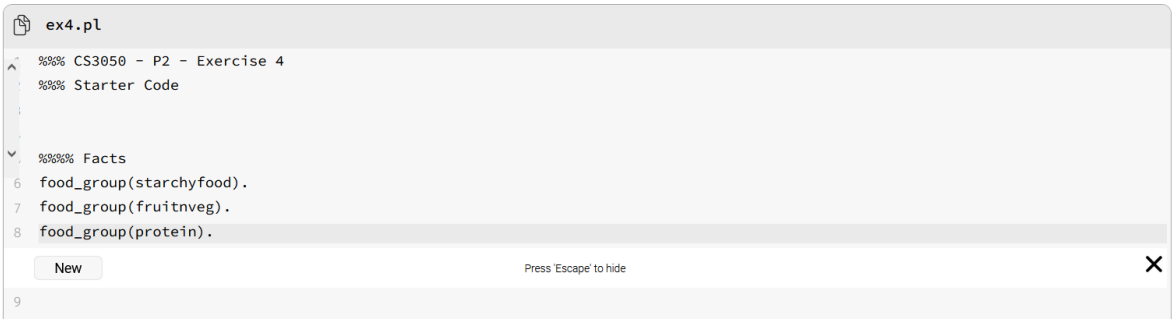


This submission can now be found in your profile tab and the discover page, and if you click it you can see all of the details were uploaded correctly

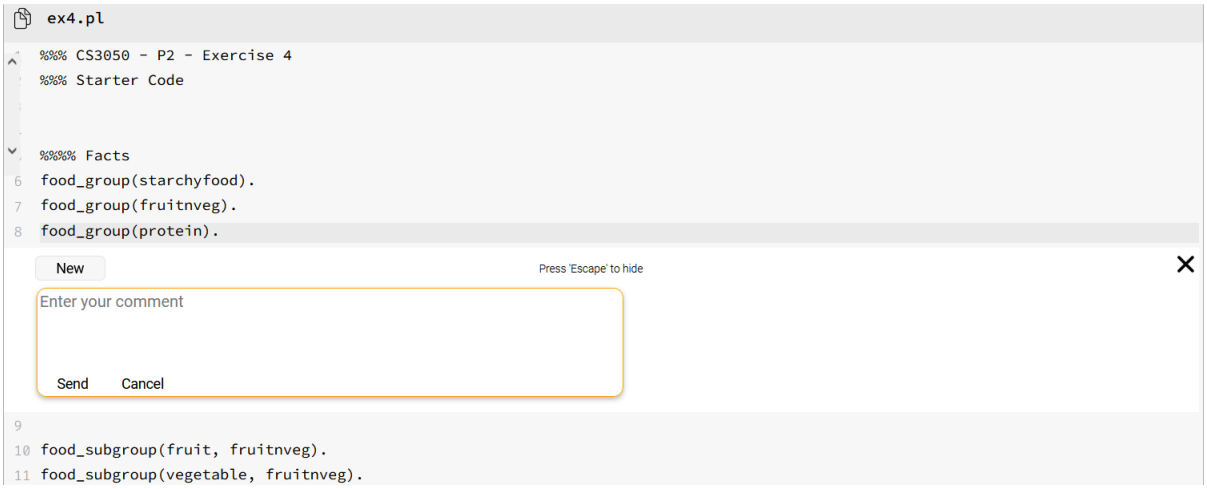


Adding a comment on a line:

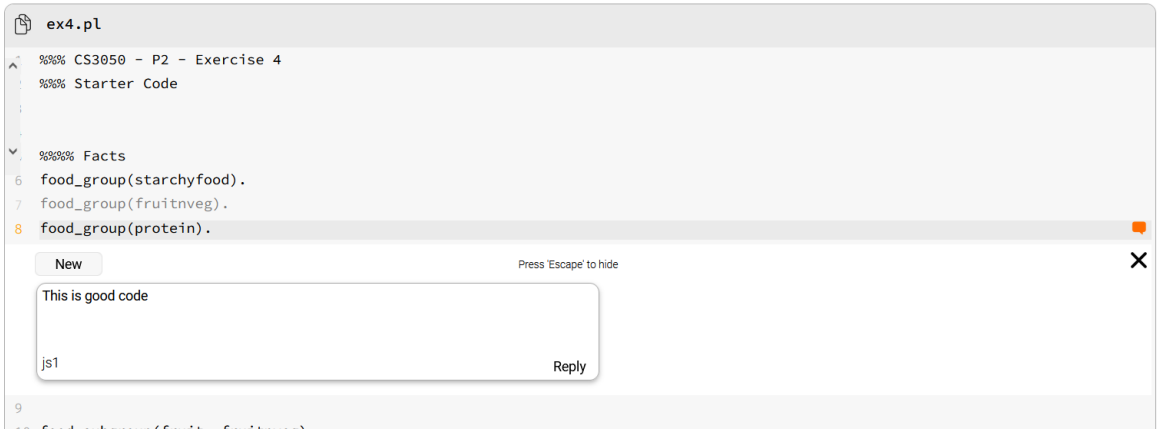
We are going to continue with the same submission as before and adding a comment on a line. So to start we will click on the line we want to add a comment to.



Now that we have selected we can press new to add a comment



If we fill out the text box and press send we can add the comment to the file



If we close this comment we can now see there is a comment maker next to the line indicating a comment has been added.

Code

ex4.pl

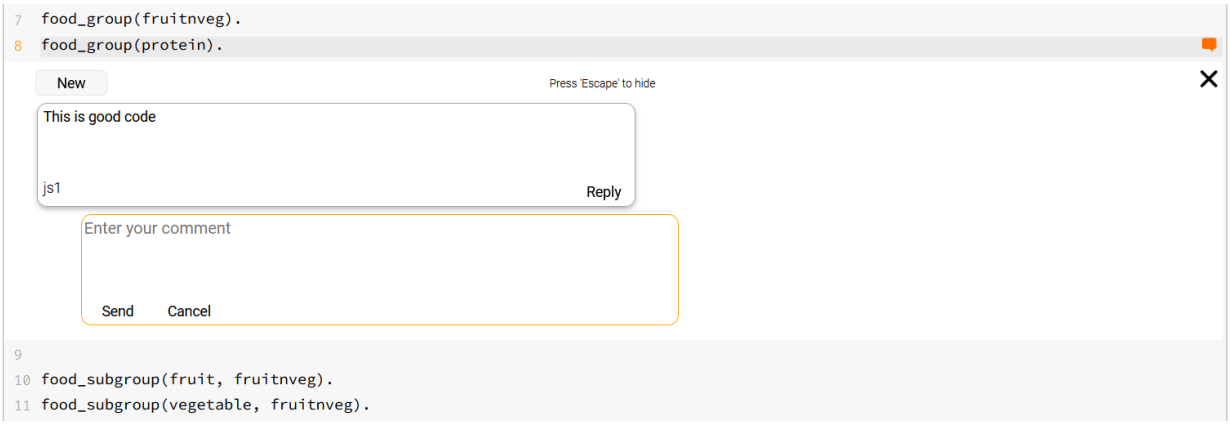
```
%%% CS3050 - P2 - Exercise 4
%%% Starter Code


%%%% Facts
food_group(starchyfood).
food_group(fruitnveg).
food_group(protein).

food_subgroup(fruit, fruitnveg).
```

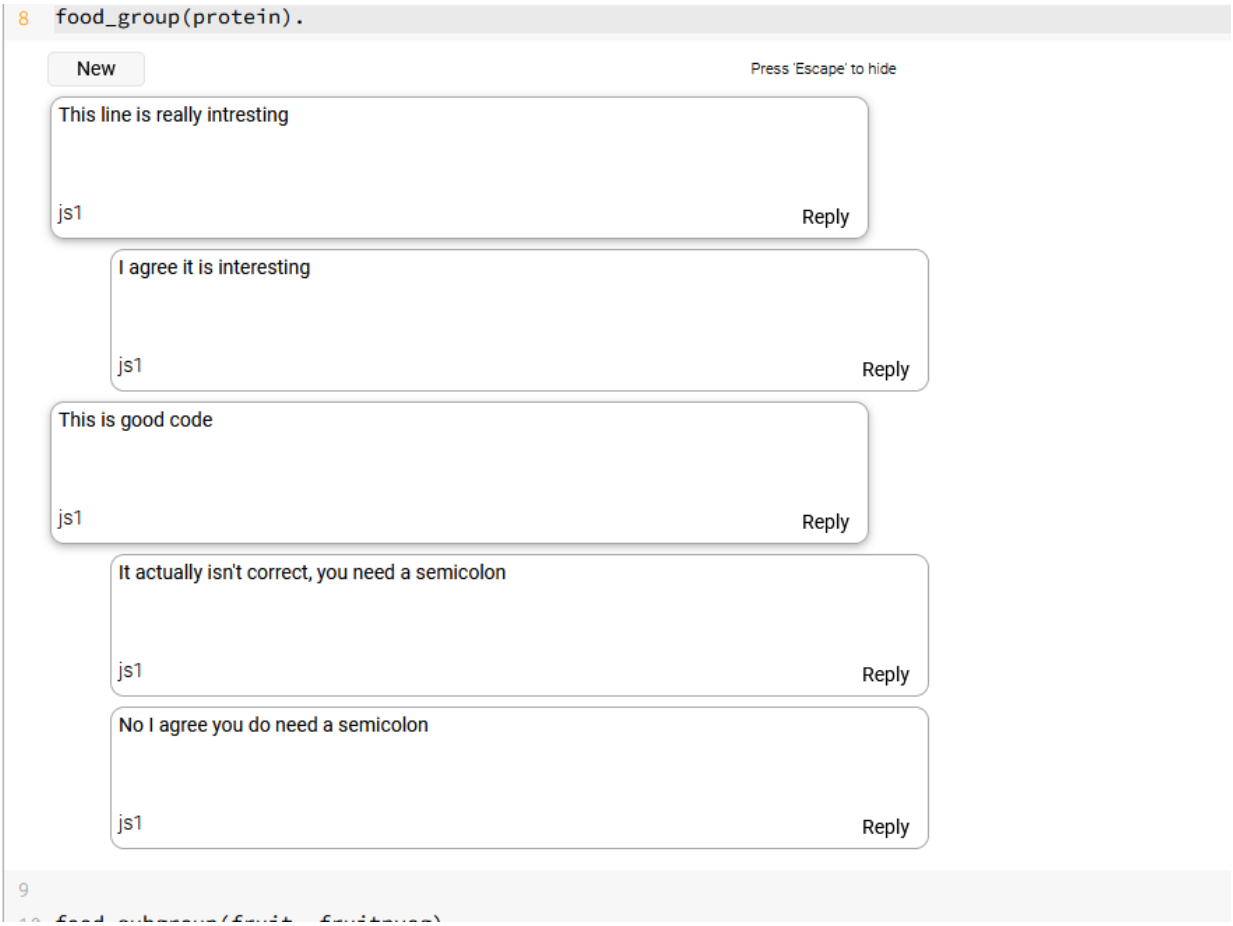
Reply to a comment:

We will continue testing on the same line as the previous example. If we click back to line 8 and then click reply we get the following



We can then fill out a reply and press send

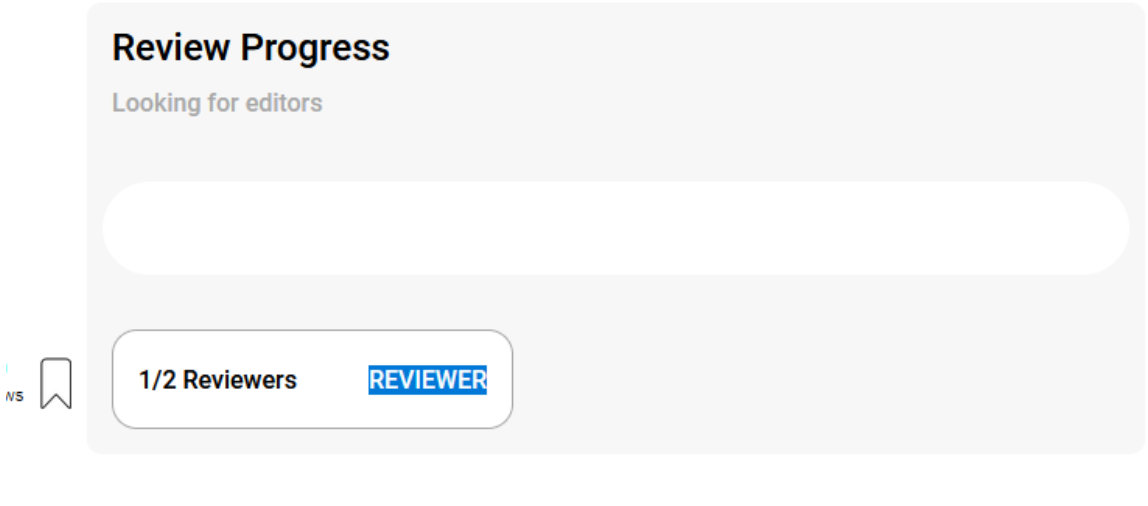
We can carry on adding replies and comments to get a nested comment structure



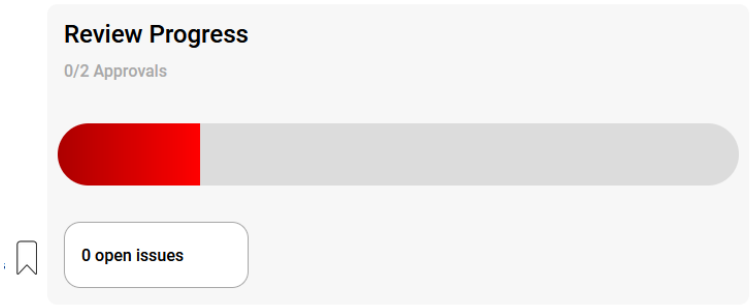
Reviewing code

For this section I created a new code submission for peer review, and I logged into another account (alb1) to review.

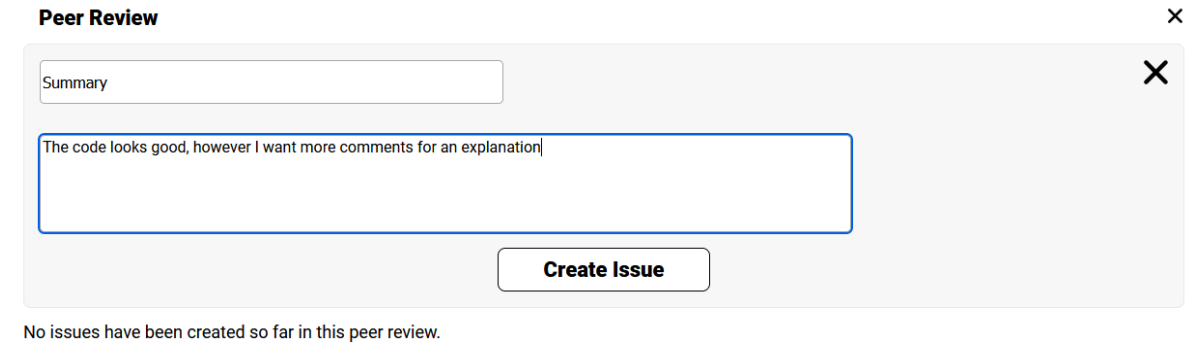
After clicking to join the review you become a reviewer.



Another account (sr1) will join the review process, and as you can see both reviewers are added

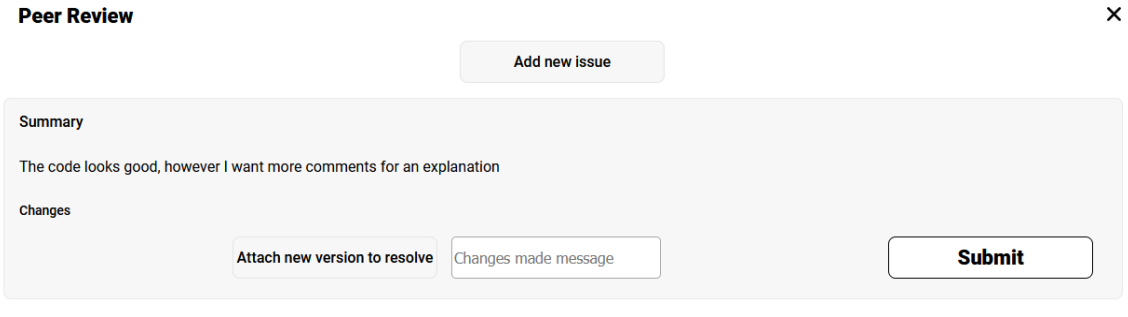


If we now click the review tab we can add an issue

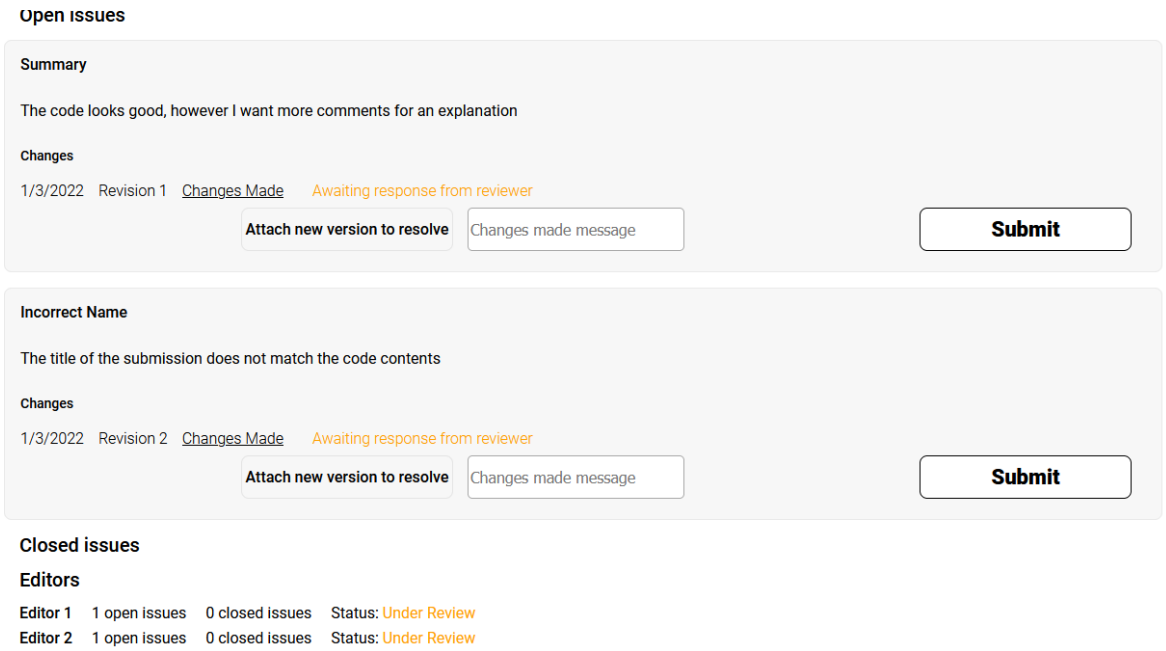


When we press create issue this issue is added
(EDIT)

If we now go back to jf1's account we can see this issue. Note I also added an issue using sr1's account

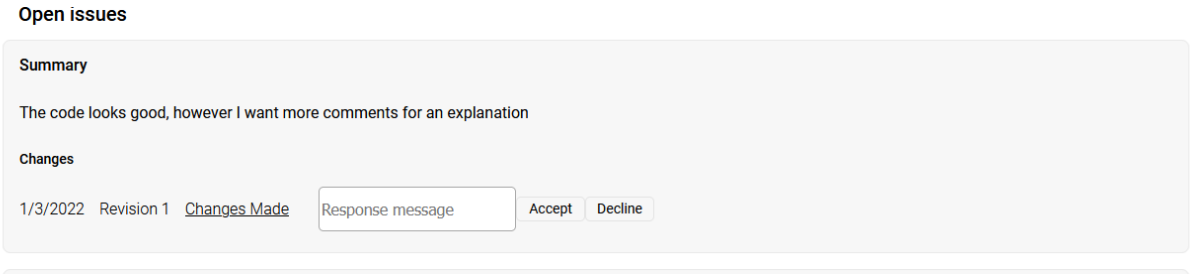


We can now attach a new version and a message to attempt to resolve this issue

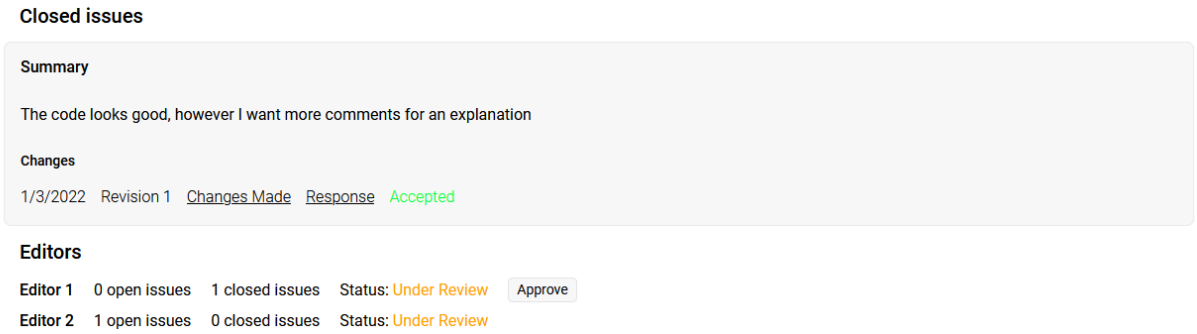


Note we can now go between all of the different revisions

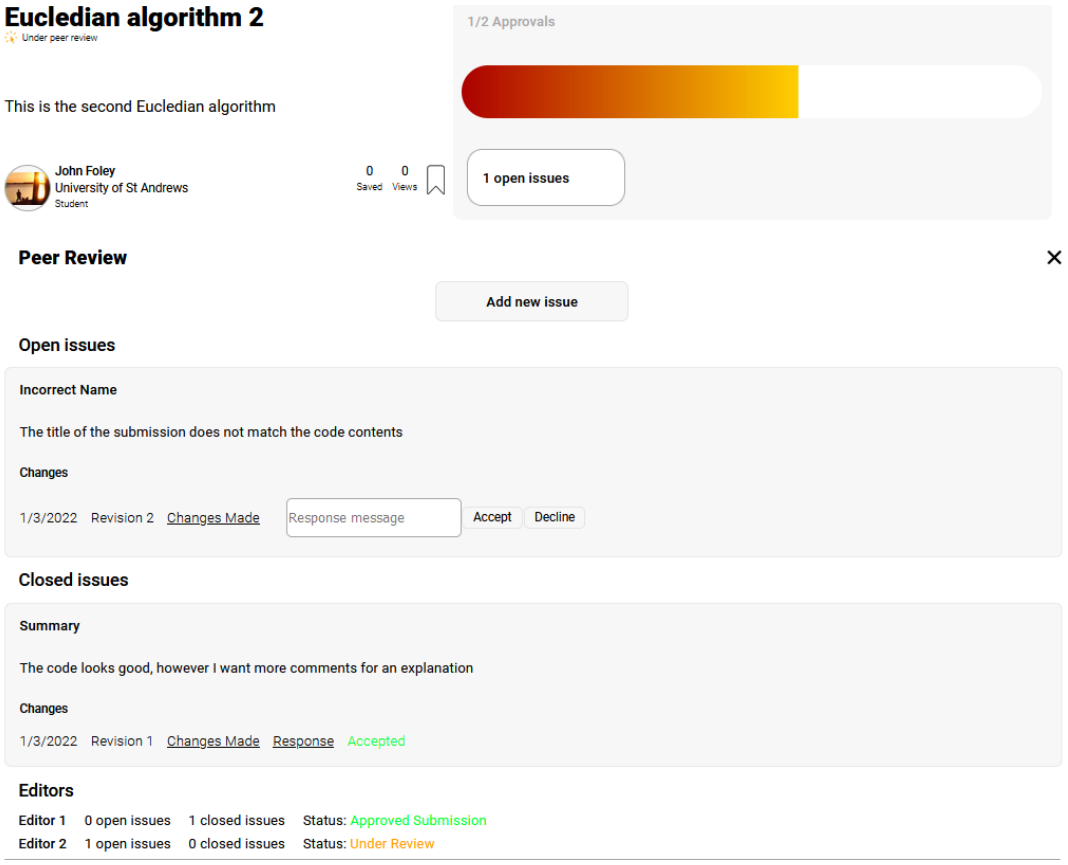
If we go back to the reviewer we can accept these changes or reject them



In this example we will accept them



Now that it has been accepted we can approve using the approve button



As you can see it is approved and the progress bar has gone up.
If the other reviewer accepts the changes and approves the code is now peer reviewed.

Eucledian algorithm 2

Peer Reviewed

This is the second Eucledian algorithm

John Foley
University of St Andrews
Student

0 Saved 0 Views

Peer Review

Open issues

Closed issues

Summary

The code looks good, however I want more comments for an explanation

Changes

1/3/2022 Revision 1 Changes Made Response Accepted

Incorrect Name

The title of the submission does not match the code contents

Changes

1/3/2022 Revision 2 Changes Made Response Accepted

Editors

Editor 1 0 open issues 1 closed issues Status: Approved Submission
Editor 2 0 open issues 1 closed issues Status: Approved Submission

180023970

User Stories

No	User story	Done?
1	A user can sign up to PeerBit	Yes
2	A user can sign in to PeerBit	Yes
3	A user can reset their password	Yes
4	A user can see their personal account	Yes
5	A user can edit their personal account	Yes
6	A user can publish code submissions	Yes
7	A user can save a draft of a submission	No
8	A user can see a list of their submissions	Yes
9	A user can view a code submission page (minimum for editing, no code rendering display)	Yes
10	A user can edit details of their code submission	No
11	A user can delete their code submission	No
12	A user can update their code with a new version	Yes
13	A user can see the code submission view with a code and folder browser	Yes (No to folder browser)

14	A user can navigate through versions of code in the submission	Yes
15	A user can download code submissions as zip	Yes
16	A user can see and attach comments to code locations	Yes
17	A user can see, start, and participate in general discussions	Yes
18	A user can attach images or files to comments in code and general discussions	Yes
19	A user can sign into other journals via our journal	Yes
20	A user can sign into our journal via other journals	Yes
21	A user can export a submission to another journal	Yes
22	Our journal can accept an incoming submission from another journal to an existing user	Yes
23	Our journal can accept an incoming submission from another journal to a not yet registered user	Yes
24	A user can submit a request for their code to be peer reviewed with parameters selected	Yes
25	An editor can select a submission to review	Yes
26	An editor can create issues to be resolved	Yes
27	A user can see the review progress item, with current stage, issues by each of the reviewers	Yes

28	A user can submit a new version of code and select which of the issues were addressed, as well as attach a comments	Yes
29	Editor(s) can review newly submitted versions and approve/decline + comment	Yes
30	User gets notified via email on review progression	No
31	A submission that is undergoing a review has a respective public badge	Yes
32	A submission that has been approved by reviewers and journal admin gets a “peer reviewed” badge	Yes
33	A user gets the feed of all code submissions	Yes
34	A user can filter for submissions that are peer reviewed	No
35	A user can search for submissions or authors via indexed search bar	No

190004593

Group Meeting Minutes

27th January 3pm meeting

All group members present

Group agreed to create a Jira board that co-exists with the school's gitlab. This way we won't have any accessibility troubles and we suffer less with gitlab. Artem will set up.

Continue our 2-week sprints from last semester into this semester.

Plan a fake deadline for the final product to prevent last minute rush.

Group agreed to have bi-weekly standup meetings.
Ben has volunteered to research app development testing so we have a better understanding of how to test our product.

Arranged to meet 30th January 3pm.

30th January meeting postponed due to illness

8th Feb meeting 11am

All group members present

Artem has presented a Jira board filled with epics.

More ideas were raised for the app such as an admin account privileges, chat feature and code editor page.

Agreed to call on Friday to discuss work flows testing and user stories.

12th Feb meeting 2pm

All group members present

We have arranged a meeting for the progress report due on the 16th.

Artem has created a new UI sketch for the web app, introducing better UI and commenting by subject.

17th Feb meeting 3pm

All group members present

Group has assigned members to certain user stories.

17th March meeting 3.30pm

All group members present

Matthew has volunteered to email Alexander about extensions or allowing some leniency to group.
Group members continuing work and restructuring sprints.

22nd March meeting 1pm

All group members present

Assigned new roles to our restructured sprints
Restructuring of our read and writes of backend
Minor changes to code submission file format

26th March meeting 2pm

All group members present

Report added to google drive
Sections of report assigned to members

28th March meeting 4.30pm

All group members present

Merged all branches

190018018

Dependencies

As Peerbit is created using npm-react, the group can easily install npm packages that are crucial to the development of the app. The group agreed that having the least amount of npm packages necessary is also very important to the loading speed of the web app.

- firebase
- redux
- formik
- JSzip
- react-router-DOM
- styled-components

190018018

Bibliography

[1]: Wikipedia: Incentive program
[2]: Zoom: A Message To Our Users