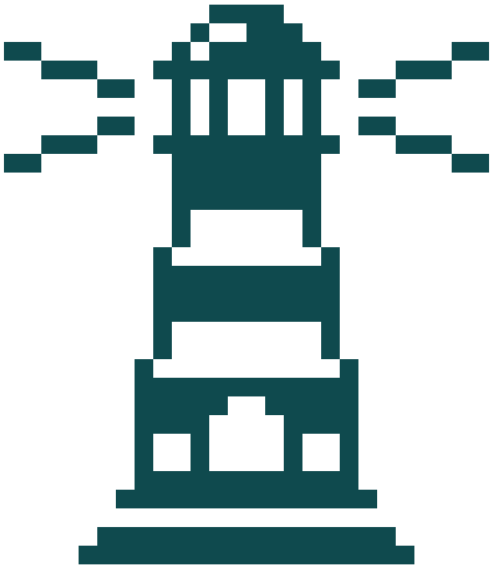

Project Alexandria

Building a platform for global scientific collaboration

by

Eve Smura, Francisco Cunha, Jannes Kelso, Maxime Caux and Miruna Cosmina Negoitescu

Group:	17B
Coach:	Prof. Benedikt Ahrens
Teaching Assistant:	Ana Marcu
Client:	Andrew Demetriou and Cynthia Liem of the Multimedia Computing Group TU Delft
Course:	CSE2000 Software Project
Report place and date:	Delft, 21 June 2024

TU Delft

Preface

This report describes a project done for the TU Delft Software Project course in the Computer Science and Engineering Bachelor's degree. The authors of the report are team 17b, a group of computer science students doing this course in the academic year 2023/2024.

We would like to thank some people and organizations without whom this project would not have succeeded:
The clients of this project, Andrew Demetriou and Cynthia Liem, for their initiative, cooperation and support during the development of Alexandria.
Our teaching assistant, Ana Marcu, for her guidance and advice.
Our coach, Benedict Ahrens, for his insightful questions.
The Software Project coordinators, for their support.
Our technical writing lecturer, Willem Vermaat, and our ethics lecturer, for their feedback on this report.
Our university, Technische Universiteit Delft, for hosting this project.

When reading this report to continue the implementation of Alexandria, the bulk of the relevant information can be found in chapter 5, which deals with the product design, chapter 6, which details the implementation process, as well as chapter 9, containing the current project team's recommendations for future development.

Delft, 21 June 2024

Eve Smura, Jannes Kelso, Maxime Caux, Francisco Cunha and Miruna Negoitescu

---

This text was written within the course Software Project CSE2000 of the faculty Electrical Engineering, Mathematics and Computer Science of Delft University of Technology.
Total word count: 12109
In writing this text, we did not use generative AI.

Summary

The scientific community faces many issues with publishing. It is slow, expensive, inaccessible, and difficult to reproduce. Researchers are flooded with peer review requests, and receiving feedback on an ongoing project is a challenge. Spontaneous international collaboration is rare, despite the success of open-source projects such as Linux. Much of scientific research involves artefacts like data and code, which are necessary for reproducibility. Despite that, since papers are most commonly published in ".pdf" format, the connected artefacts are rarely included in the final product.

To alleviate these issues, Andrew Demetriou and Cynthia Liem of the Multimedia Processing Group TU Delft devised Alexandria, a collaborative and open-source platform to publish, view, and review scientific literature.

This report presents the development of a proof-of-concept version of Alexandria, designed to showcase the possibilities of such a platform. The process was split into two parts.

The first part was defining the scope and design of the proof-of-concept. The most important requirements were identified based on Andrew Demetriou's high-level vision for the platform. We opted for a broad scope, to present the possibilities of Alexandria, and we identified the core functionalities that could be developed in the time frame of the current project iteration. The graphic design was heavily based on a previously created user interface, which was made by Zhuoting Wang. It mainly focused on streamlining the publishing process, to make the website easy to use for any members of the scientific community.

The second part concerned the development of the platform itself. We developed a web application that hosts posts which represent scientific publications. Functionality to create accounts and log in was implemented, together with the ability to publish research in the form of Quarto Markdown folders.

We also created a collaborative publishing process using Git, allowing any user to propose a change to an existing publication and easily reuse existing work. A peer review process for the publications was designed, allowing the community to verify the quality of the research, as well as reject or accept changes proposed to posts.

The platform was created as a web application consisting of a front end which users directly interact with, and a back end that handles logic and data persistence. The two elements communicate via a RESTful API (Application Programming Interface), with the front end choosing actions to perform and the back end responding with the result and current program state.

The proof-of-concept of Alexandria demonstrates the broad capacities of such an idea. It can be used to present the general idea of a collaborative open-source scientific publishing platform and convince potential sponsors or users that the problems existing in the modern research community can be addressed in this manner. It also allows for testing and refinement of the various functionalities, e.g. the peer review process, which are currently implemented in their most basic form.

Contents

1 Introduction

Scientific research is a cornerstone of society, yet publishing research is a long and trying process. The formal peer review process is slow [1], drastically stalling research. Journals have paywalls and charge article processing fees [2, 3], impeding accessibility, and scientific discussions are scattered across many platforms and mailing lists [4]. Code, data, and other materials produced to write papers are not published, making papers less reproducible [5].

Andrew Demetriou and Cynthia Liem of the Multimedia Computing Group TU Delft are heading a project meant to address these issues: an open-source collaborative online platform made for researchers inspired by Wikipedia[1] and GitHub[2]. The platform, called Alexandria, aims to solve the aforementioned issues through a simple system of posts, discussions and peer reviews. The platform aims to promote collaboration by: allowing any user to contribute to any post; enabling transparent reviews and discussions in real time; and serving as a database of knowledge for interested parties, since it is open-source and free. The aim of the project is to implement this platform.

The aim of this report is to account for the decisions made in the development of a proof-of-concept of Alexandria. In the initial version of Alexandria, users can post research projects that then become collaborative. This allows others to contribute, discuss or peer review it in real time, while the research is still ongoing. Any user may make contributions to a post, and detailed authorship is tracked. Changes can be approved via a peer review process to keep work quality high. Each post consists of a scientific publication, any files that were used in its making, and discussions relating to it.

The report is structured as follows. Chapter 2 lays out the research done into the relevant issues in academic publishing that Alexandria is meant to solve, while Chapter 3 describes how Alexandria as a platform tackles these problems, as well as the requirements elicitation process, and the requirements themselves. Chapter 4 lays out the development methodology used by the team, while Chapter 5 addresses major design decisions taken in development. Chapter 6 and 7 focus on the technical aspects of development and the quality assurance and testing, respectively. Ethical considerations are discussed in Chapter 8, while Chapter 9 contains relevant discussion points as well as a road map for future development.

[1] https://www.wikipedia.org/
[2] https://github.com/

2

Problems with Academic Research and Their Effects

In this chapter we will justify the need for a platform like Alexandria. Section 2.1 lays out main the issues the scientific community faces that Alexandria will address, Section 2.2 contains preliminary research into who is affected and Section 2.3 examines how they are affected. Finally, Section 2.4 examines the already-existing platforms that address these issues.

2.1. Issues in academia

This section lays out the most important problems researchers face in the field of academia.

Lack of collaboration With over 8.8 million scientists worldwide [6], there is an incredible amount of specialized knowledge. However, due to how decentralized this knowledge is, it can be difficult to connect the right experts with the right research projects. This is evidenced by the fact that only about 27% of research happens in the context of an international network[4]. Since it is likely that an expert with greater knowledge of a topic is in another country, this lack of formal international collaboration almost certainly causes the quality of research to suffer.

It also appears there is currently no primary forum for informal, decentralized, academic discussion. According to Andrew Demetriou, where Twitter used to be a hub for such debate, X has left a vacuum. These discussions are now fractured across many platforms, including X [1], BlueSky [2], and Mastodon[3].

Inaccessibility Journals do not pay authors for publishing articles nor reviewers for giving feedback. Usually, journals even expect authors or their institutions to pay for article processing fees, especially when access to the article is free of cost[3]. Partially due to this, academic publishing is a lucrative business, with global sales totaling over $19 billion, ranking it between the music and film industry[7]. Additionally, readers often need to pay to view articles[3]. This system builds barriers of entry to share and read about current research.

Slow publishing process Another hurdle researchers face is that it can take a long time to get a paper through peer review. According to one study, the average researcher finds 6-7 weeks to be an optimal waiting time from submission until first formal edits are made. However the average wait time can be longer, with research finding over 14 weeks in the field of biology, with some cases approaching a year[1]. This greatly hampers the speed at which researchers can work.

Lastly, a paper is usually just the tip of the iceberg; the culmination of much data gathering aggregation and thorough analysis. These are the basis for the insights discussed in the paper, but are normally not properly published with the paper. This makes it hard to verify how results were arrived at and what data they are based upon, impairing reproducibility.

[1] https://x.com/
[2] https://bsky.app/
[3] https://mastodon.social/explore

2.2. Stakeholders and use cases

In this section, potential users and how they would engage with the platform have been identified.

The main stakeholders include:

- **Researchers** who want to publish research, share insights, ask questions to the community, or are searching for research.
- **Experts** in their field and people with unique skills who wish to collaborate with others and contribute to the community, e.g. through discussions or peer reviews.
- **Students** who want to ask questions or participate in discussions.
- **Non-academics** who would like to read research and learn about ongoing conversations in their field of interest.

Due to this project's nature as a university course, there are diverse stakeholders with various desires and requirements. The team has decided to prioritize the requirements of the client, as long as they do not interfere with the successful completion of the course.

2.3. First hand accounts

This section lays out the interviews conducted with potential users.

Our client is not only the product owner but also part of the target audience, as a researcher and PhD student. Thus, he already has a vision for a solution, which he outlined with the help of user experience (UX) designer Zhuoting Wang. Interviews with the client and the designer, along with Wang's research, greatly contributed to our understanding of the problem. Additionally, we looked at existing platforms which aim to solve similar issues and what they lack.

Wang has conducted interviews with ten different scientists to understand their needs, frustrations, and current limitations of existing platforms. These scientists were in various stages of their careers, with diverse publishing and reviewing experience. Wang passed her insights, along with a documented version of her findings[8, 9] onto us. Her observations are summarized as follows:

- Researchers in niche fields often find it difficult to get feedback on their work.
- The peer review process is generally seen as slow.
- Sharing underlying data, code, and other artifacts is as important as the final paper.
- Feedback can be of low quality and from perspectives that are too similar.
- The communication process is unclear and inefficient.
- There is desire for more trust and cooperation within research.

These observations reflect the problem as previously described and further show its relevancy. Additionally, the participants in Wang's research[9] share a common sentiment that the current review and publication process lacks healthy incentive mechanisms. Namely, reviewers feel like they do not get the recognition they deserve, and the primary motivation mechanism is strongly linked to personal financial gain. The participants expressed a desire for an incentive system rooted in mutual collaboration and respect.

2.4. Alternative platforms for science

There are multiple platforms that work to solve some of the issues Alexandria aims to address. They have been examined to ensure the need for Alexandria exists.

Open Science Framework (OSF) The OSF[4] is a platform that aims to facilitate collaboration between researchers and serves as an open database of papers and projects. Individuals can post projects they are working on, as well as preprints of papers.

However, the OSF focuses on collaboration within research teams, rather than spontaneous collaboration between strangers. It lacks an easy way for members of the community to publicly leave comments and does not provide peer review.

[4] https://osf.io/

Open review Open Review[5] addresses the black-box aspect of the peer review process. Researchers can submit their papers to venues and receive feedback quickly. They can also ask questions about papers and start discussions. There is a space for public reviews and comments.

Still, Open Review fails to solve the other aspects outlined in our problem statement. It does not allow researchers to post or receive feedback on active developments of a project and still relies on venues and conferences for reviewers.

Wikipedia Wikipedia[6] is a ”free multilingual online encyclopedia”[10]. It is maintained by a decentralized community of volunteers, and anyone can post or contribute to articles. In this way, Wikipedia does a great job at promoting spontaneous collaboration.

Nonetheless, Wikipedia is not a hub for scientific research papers. The articles are not formulated accordingly, and the review process does not meet academic standards.

Git Git[7] is an open-source tool that focuses on collaborative software development. It allows working on individual components in parallel and for academics to easily contribute to and build off of others work. Furthermore, platforms that host Git services, such as GitHub[8] and GitLab[9], offer additional tools which allow for reviews, feedback, and discussions.

However, a big downside of Git is its high barrier to entry; it requires domain-specific knowledge to be used effectively due to its technical nature. This severely limits its potential for large-scale scientific collaboration.

[5] https://openreview.net/
[6] https://www.wikipedia.org/
[7] https://git-scm.com/
[8] https://github.com/
[9] https://gitlab.com/

3

Alexandria's Vision

This chapter outlines the idea behind *Alexandria* - the proposed solution to the problems in chapter 2. Section 3.1 describes the vision for the platform. Section 3.2 details the most important requirements.

3.1. High-level vision for Alexandria

This section presents the idea behind the platform whose development is outlined in this report. We begin by outlining the vision for the platform, then discuss the solutions it poses for the issues described in chapter 2, and finally explain the previous iteration of the project.

3.1.1. Description of the platform

Alexandria is a collaborative open-source platform dedicated to publishing, discussing and developing scientific research. It is designed to be community-oriented and intersectional, merging the functionality of version control with an intuitive interface. Any user can post their reflections, ask questions or publish research.

Any work published on the platform becomes property of the community - anyone can make additions to it. Community members with the relevant expertise are able to peer review the proposed additions, approving or rejecting the changes. In this way, the community controls how a post evolves.

A post is a repository of Quarto files[1], rendered by the platform into a human-readable format. This allows users to share the code behind their work, merging the gap between the process and the final result of research. Quarto is easier to use than LaTex, as it is a form of Markdown, and allows for the incorporation of complex figures using the four most common scientific programming languages: Python, R, Julia, and Observable. Using Quarto as a rendering tool is a hard requirement from the client, and informs the project's core vision.

3.1.2. Alexandria's solutions

As outlined in Chapter 2, there are many issues in academia that Alexandria aims to solve. In this subsection, we detail how Alexandria might solve these problems.

Firstly, international collaboration, which is often difficult to arrange, becomes easy through Alexandria - anyone who wishes to can collaborate on a post. The platform allows for free discussion in the format of a forum, leaving space for users to express their thoughts.

Secondly, Alexandria is designed to be accessible to anyone. It is completely free and open-source. It has a simple user interface, to allow people with no technical expertise ease of use, and unlike scientific journals, there is no admission process to publishing research. It is also possible to create posts anonymously.

Thirdly, Alexandria also addresses the difficulties involved in publishing - users are free to post their work at any time and in any state - incomplete and unpolished. This allows for sharing knowledge in real time, without the delays enforced by the publishing industry.

[1] https://quarto.org/

The ability to post the files behind the research allows for stronger reproducibility and better understanding. It also encourages users to ask for help from the community. Others are able to continue research from the point it was left off, without the need to recreate it from scratch.

Finally, the peer review mechanism of the platform allows users without the necessary connections to still get feedback on their research. Since any user with the relevant expertise is encouraged to review, the length of the process becomes much shorter. It also removes the burden from the person publishing to reach out to people and request reviews.

3.1.3. Previous iteration of Alexandria

This is the second iteration of building Alexandria within the Software Project. The previous version of the platform[11] is available as open source software under the GNU General Public License v3.018, which the client has encouraged us to make use of.

Being the first iteration, the previous team did not have clear guidelines for what Alexandria should be, and had to scope out the most important features themselves. They helped lay the foundations of Alexandria, the vision being later refined with the help of a UX design student. As such, the initial project did not fully align with the requirements our team engineered (see 3.2), and the code base of the previous team was not utilized - no code was re-used. However, it was relevant in the planning process, inspiring the choice of technology stack and some of the requirements.

3.2. Main requirements

This section presents the main requirements found by the elicitation process.

As described above, Alexandria aims to solve many issues with the publication process, while being an intuitive and highly social online platform. Andrew Demetriou's goal with this prototype is to attract future developers and showcase the potential of such a platform. Thus, our team opted for a broad scope, integrating many small features, as opposed to implementing few features with a great level of detail.

One of Alexandria's main goals is hosting scientific articles that evolve through continuous community contributions and have their quality assured via peer reviews. This is the goal chosen as the focus of this prototype – therefore we identified the critical components to be: publications in the platform, a process to suggest changes to a publication and a process to review and approve these suggestions. Additionally, a publication must be composed of a Quarto project, and as transparency is core to Alexandria's vision, the files that make it up also must be accessible from within the website itself.

To have these components in the platform, the team designed the aforementioned processes, and from them devised the complete list of requirements, out of which the most important ones are:

- A member can create a new post by specifying a title, authors and collaborators, uploading their content, and then publishing it.
- A user will, by default, view a post as a rendered version of its Quarto project files.
- A user can choose to view a rendered post as the Quarto project files that underlie it.
- A member can create a new branch for an existing post. They must provide the modified version of the post files. They may also propose changes to the metadata of the post.
- A user may view a list of a post's peer-reviewed, open for review, and rejected branches.
- A user can select one of the branches listed on the post page to view the branch's page.
- On the branch page, a user can view:
  - its reviews,
  - its approval status,
  - the proposed new version of the post (as its files or rendered document),
  - the version of the post before modifications.
- A member can peer review a branch by giving written feedback and accepting or rejecting the branch. Up to three members can review one branch.

- If a branch is accepted (by exactly three members), the Quarto project files associated with it will replace those of the post. The authors of the branch will be added as contributors to the original post, and the branch and post will be marked as ”peer reviewed”.
- If a branch is rejected by any reviewer, the post is not updated, and the branch is marked as ”rejected”.

For additional details, the full list of requirements created with the MoSCoW system [12], together with a glossary, can be found in Appendix D.

Changes in the original requirements During the development process, minor changes were made to the requirements devised at the start of the project. Most notably, using Git for implementing the back end was added as a must have, non-functional requirement. More details about this decision in Section 5.2. Another important addition, as a should have non functional requirement, was the use of JWT tokens for user authentication and password hashing. Other changes include renaming ”Merge Request” to ”Branch”, more explicit wording for ”tags” and ”scientific fields”, and other minor rephrasing for the sake of consistency and clarity.

4

Development Methodology

This chapter explains the general approach used during the development of Alexandria. Section 4.1 explains the tools used to aid the coding process.

4.1. Planning and organization practices

This section describes the methods and tools used during the development of Alexandria to allow for effective planning.

Scrum-based development approach To aid in the planning process during the project, we decided to use a development approach called Scrum [13]. It is an iteration-based planning approach that allows for flexible planning and easy extensibility. Considering the experimental nature of the project, such a planning approach was considered important. Other reasons for choosing this approach are: the fact that it allows for mutual collaboration and shared responsibility, as it does not rely on having one sole team leader; the team had previous experience with this approach, one team member having had the role of scrum master multiple times in the past.

The general structure of a project using Scrum is shown in figure 4.1. At the beginning of a project, a general list of all tasks, called the backlog, is created. This is used for the whole duration of the project. The planning is done in weekly intervals, called sprints.

Every week a retrospective and planning meeting is held. During this meeting, the team first reflects on the completed tasks and encountered issues from the previous sprint. Then during the planning phase a sprint backlog is created. The team decides which tasks from the overall backlog to complete this week. In addition to this, short daily meetings were held, in which every developer explains what they worked on the previous day, what issues they encountered and what they plan to do that day.

To keep team relationships good and communication effective, we created a code of conduct. It details the general practices and standards upheld by all of us, as well as strategies for dealing with conflicts. The full document can be found in Appendix B.

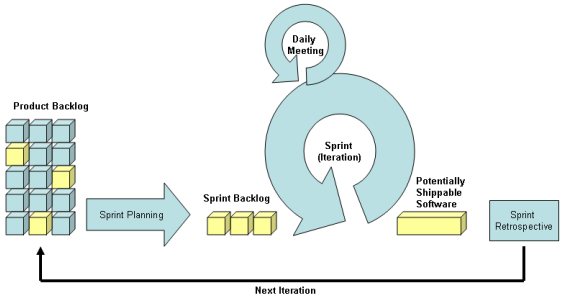

Figure 4.1: The Scrum development process. Source: `https://www.codemag.com/article/0805051`

Work division and planning During the weekly sprint planning meetings, we first chose the backlog for the iteration, and then assigned the issues to team members. We chose to have a consistent task division to allow for more effective coding, with two team members taking up the front end development, and the remaining three working on the back end. We attempted to divide the issues equally. The final work distribution can be found in Appendix A. In addition, team members were assigned permanent roles during meetings, with a designated chair and note-taker.

Definition of done In order to decide if an issue has been completed, we decided on a ”definition of done”. This is a list of requirements that must be fulfilled for the task to be considered finished. An issue is considered done, when:

- All subtasks relating to it have been completed;
- The code has been reviewed and approved by at least two team members(excluding the author);
- Its tests achieve at least 80% statement or branch coverage.
- The code has comments;
- If necessary, the code has been documented in the Wiki;

A functional requirement will be considered done when all issues relating to it have been completed, and it works as expected by the team and client.

Documentation practices To ensure the code base is understandable and can be understood by future developers, we documented it through comments in the source code. In addition, a Wiki contained in the GitLab repository [1], details the most important aspects of the design.

Client communication In addition to flexible communication with the client via Mattermost, we ensured that the team’s prototype stays true to the client’s vision by holding weekly meetings. Within these meetings:

- The client is shown a demo of the platform’s current state.
- The client receives a status update about the current progress.
- The client receives a briefing of the sprint retrospective and planning.
- Any questions the team or client have are discussed.

In the case in which the client wants to propose a change or an addition to the functionality, several aspects will be considered:

- how much time/effort it takes to implement
- if the team is required to re-implement some feature
- if the project requirements have to be changed
- how essential it is to the project’s vision

If a change requires re-implementation, changing the requirements, or relates to an issue being currently worked on, the team might have a discussion about the feasibility and importance of the change. The team would then inform the client of their thoughts and reach a conclusion together. For minor changes that do not relate to the issues in the active sprint backlog, they might be taken into account during the next sprint planning.

[1] https://gitlab.ewi.tudelft.nl/groups/cse2000-software-project/2023-2024/cluster-v/17b/-/wikis/home

4.2. Docker

Incompatible development environments are a prevalent issue in software development. To ensure that the application behaved consistently on everyone's machine during development, Docker[2] was used to containerize builds - in other words, run them on a custom-built virtual computer, that would act the same everywhere.

Docker was used consistently from start to finish on the back end. It greatly aided the database and test database setup. The front end used Docker for the majority of the project, but encountered very difficult networking bugs in later weeks. As such, it is recommended to run the front end locally. Fortunately, consistent development environments are most critical for the back end and database, so the front end has encountered no issues as a consequence of this.

[2] https://www.docker.com/

5

Design of Alexandria

In this chapter we lay out the design of the Alexandria platform. We begin by introducing the architecture of the application from a technical standpoint in section 5.1. Section 5.2 presents consequential decisions that the team had to make during the development process. Finally, section 5.3 describes the user interface design and the overall flow of the application.

5.1. Software architecture

This section lays out the architecture of the Alexandria platform. First, we describe the architecture in broad terms, introducing the *front end* and *back end* components. Then, subsections 5.1.1 and 5.1.2 detail the specifics of the front-end and back-end architectures, respectively. In subsection 5.1.3 we explore the database and data model designs. Finally, subsection 5.1.4 explains Alexandria's API (Application Programming Interface), which describes the contract of communication between the front end and back end.

The Alexandria project uses a client-server architecture [14], which is the most commonly used structure for web applications. The main components of the system are therefore the client, usually called the front end, as well as the server, which is mostly referred to as the back end. An overview of these components can be seen in Figure 5.1

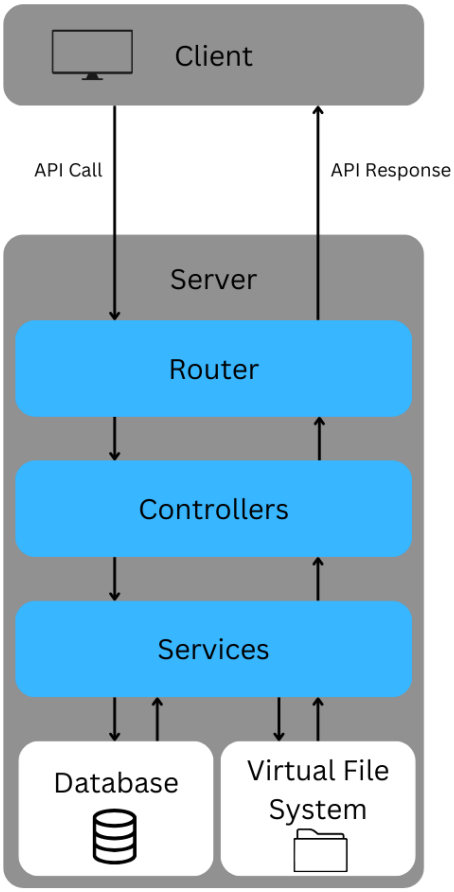

Figure 5.1: High Level Architecture Design

The front end initiates the communication with the server, which is done via an application programming interface (API), and makes sure the data that it receives is displayed appropriately on screen. The back end handles requests from the client, storing and retrieving information. The server side is further split into three parts: the main part handling routing and business logic, the database system, and, in our case, the file system. We proceed to elaborate more on each of these components.

5.1.1. Front end architecture

The front-end uses an SPA architecture. There are two potential approaches to front-end design: multi-page applications (**MPA**) and single-page applications (**SPA**) [15, 16]. MPAs are the traditional approach to web applications. For each update to the user interface they require a page reload, and thus a request to the server. In contrast, SPAs rely on JavaScript APIs [17] to update the user interface within the browser, without reloading the entire page. Both have their respective strong points, which are presented in table 5.1.

Table 5.1: Comparison between SPA and MPA applications.[15, 16]

	SPA	**MPA**
Advantages	• Faster response times • Faster development and easier debugging • Streamlined user experience	• Search engine optimization • Website statistics • Fewer security risks
Disadvantages	• Worse search engine performance	• Generally slower • Harder to develop and maintain

As the goal for our implementation iteration is to produce an extensible proof-of-concept for Alexandria, we opted for an SPA. It allows for a smoother user experience, making the final prototype more appealing. Additionally, it speeds up the development process, an important improvement considering the time constraints of the project.

To implement a SPA, we used React in combination with the Next.js framework, as described in section 6.1.1. The Next.js App Router enforces the following structure: starting with the *app* directory, each sub-directory represents a segment of the URL, and contains the React component that makes up the page at that URL. The *lib* folders contain business logic, such as custom types, formatting utilities, fetchers for interacting with the server, and custom hooks. The *components* folders contain code for reusable UI elements.

5.1.2. Back end architecture

We now cover the high-level architecture of the application's back-end, first characterizing it, then describing the path that a typical request takes, and finally detailing the method of communication between the client and server.

The Alexandria prototype uses an adaptation of the classic Model View Controller (MVC) architecture, an industry standard for web servers. This architecture consists of three components; the *view*, the client-facing interface, the *models*, the underlying data models shared by all parties and *controllers*, pieces of code that implement business logic.

When a web browser visits the Alexandria website, it is served the 'view', which is the front end described in section 5.1.1. The front end then, from within the user's browser, makes further specific requests to the server. Each type of request is mapped to a 'controller', a component responsible for one piece of high-level business logic. For example, a controller might create new data, save it, and notify other code about it. The 'models' are structured data with relationships between them, such as 'Members' and 'Posts'.

Let us consider the architectural 'flow' of a request sent to a controller (see Fig. 5.2 for a diagram of the components and the direction of their interactions with each other). The controller is responsible for performing a single, specific task, but may make use of *services*. Services are not specific to a task, but rather to a model or a particular domain (such as file system operations). Services use *repositories* to interact with the database. Repositories provide nothing but simple CRUD operations (Create, Read, Update, Delete) for models.

The web communication between the client and the server does not use the models *directly*, however. The back-end makes strict use of Data Transfer Objects (DTOs) for all transfer of model-related data to and from the client. These are objects that are associated one-to-one with the models, and define what subset of information in that model is relevant to the client - excluding the rest. This abstraction allows for separation of the complete server-side view of a model from the client's view, as the former may contain too much or irrelevant information.

In addition, client requests that contain *incomplete* model data (e.g. creation parameters for something that does not exist yet) are not represented using DTOs, but 'forms'. These are objects that

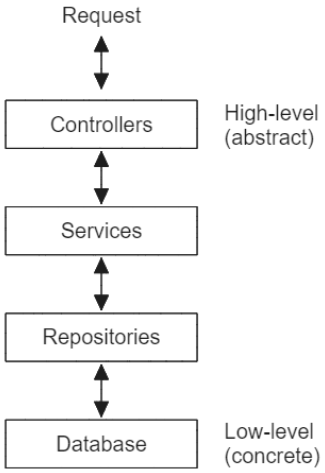

Figure 5.2: Flow of a request to the back-end

describe a specific operation, such as 'CreatePost', and contain strictly the data necessary for this operation. This design choice was made because the alternative is sending malformed DTOs; DTOs that contain 'false' data, such as a blank identifier for an object that has not been created yet.

5.1.3. Database design

The database is responsible for persisting the system's data models, which were defined in Subsection 5.1.2. As is standard practice, we chose to make use of a relational database in our architecture. This provides robustness through strong consistency constraints, and easy-to-understand representations of associations between objects. The relational database models that were designed for the prototype can be found in appendix E.

We now briefly cover some points of interest relating to the design of database models, and the associations between them. First, we consider the use of *composition* in the model design. Then, we describe the use of 'container' objects for re-usability in one-to-many database associations. Finally, we discuss a point of note on posts and branches.

The database models make use of composition, meaning that new objects are created by wrapping around other objects. There are two notable examples we discuss here: a) project posts encapsulating posts and b) closed branches encapsulated branches.

Project posts are a certain type of post, providing extra metadata and reviewing/merging functionality (as described in 3.2). To avoid data/functionality duplication, a project post is a wrapper for a post, fully containing it. As a result, for a project post, there exists both a project post ID and a post ID. The same goes for closed branches; a 'closed branch' is a branch with extra metadata. As such, the closed branch object fully contains a branch as a field. Composition of models did lead to integration issues with the front end, further elaborated in Chapter 9.

Another point of interest is the use of container objects. In order to re-use data models such as 'discussions' in many contexts (e.g. posts, branches), we created specialized 'container' models for each (e.g. a 'discussion container'). This was necessitated by the strict constraints imposed by the relational database on associations between models.

Finally, we touch on a design issue related to posts and branches. They contain much similar data and functionality, as a branch is conceptually an 'updated post', designed to replace the post. However, the current design duplicates this data and functionality, due to difficulty working around the database's strict foreign key constraints. Further elaboration follows in chapter 9.

5.1.4. API design

The client communicates with the server through HTTP requests, which are standardized using an API, specifically REST API[1]. The back-end development included the creation of a detailed API specification,

[1] "What is REST?" `https://restfulapi.net/`

which can be found on the projects Wiki [2]. This API specification includes every endpoint necessary to fulfill all of the requirements detailed in appendix D, and as such not all of them are functional.

The specification follows standard design practices, such as path conventions and versioning. As such, the endpoints are split under 8 categories, based on the main database models they interact with. As such we have, endpoints for: branches, discussion containers, discussions, filtering, members, posts, project-posts, and tags.

There are several design choices made to ensure effective and secure data transfer between the front end and the back end. Since the communication happens through the endpoints, this is part of the design of the API specification.

Firstly, none of the endpoints return data models as they are stored in the database. The bodies of HTTP responses are always DTOs (Data Transfer Object). This allows for increased security, as only necessary data is sent to the front end. For instance, passwords are excluded from member DTOs. The exact data that is included in the DTOs is present in the specification, so that the front-end always knows what information it will receive.

Secondly, the DTOs do not fully contain other DTOs. This is to prevent recursive data models from returning the entire database. Instead only ids of the associated objects . For example, when sending a post object, it will include a list of author ids, as opposed to containing the DTOs of all the authors. This also minimizes network traffic, allowing the front end to make specific requests for only the data that it needs.

5.2. Important design choices

In this section, we lay out the reasoning behind the most important decisions made during the development process. Those decisions include the integration with Git, the prioritization of security measures and the constraints of the file upload.

Using Git The requirements the team devised (see section 3.2) could easily be implemented without actually leveraging Git, as the prototype does not allow for complex version control, branching or merging functionality; it only allows for overwriting files and storing different versions. This posed the question of whether or not Git itself should be used in the implementation of the back end.

The advantages of not using Git would be the simplified implementation process. However, not having Git integrated into the back end right from the start could prove to be a big setback for future developers, as a large amount of code would have to be rewritten in that case. Seeing that the prototype is intended to be picked up by other developers, the team opted to future-proof and use Git, at the expense of additional complexity.

Low prioritization of security measures As mentioned in section 3.2, the current proof-of-concept has a broad scope. The goal is to showcase as many features as possible. With this in mind, and due to time constraints, security concerns were not considered a priority. Another factor that played a role in this decision was the team's lack of experience with implementing cybersecurity measures.

Thus, the only implemented security aspects were basic user authentication, using tokens, and password hashing as to not store passwords in plaintext, as described in Chapter 6. These particular aspects were chosen due to their ease of implementation, and the team's familiarity with these concepts.

File upload constraints A core element of Alexandria is the ability to upload Quarto projects. These projects oftentimes represent entire directory trees, with files, folders and subfolders. It is possible to give users the option to upload all of these elements individually, however, it would be cumbersome implementation wise. There would have to be functionality for recursively uploading all files in a directory, storing the structure, and transferring all of this over the network. For this reason, users are required to upload their projects as a zip archive. This makes the implementation easier overall, as the front end only needs to prompt the users for a single file, and there only needs to be one file sent over the network.

[2] https://gitlab.ewi.tudelft.nl/groups/cse2000-software-project/2023-2024/cluster-v/17b/-/wikis/Documentation/Backend/API-specification

Simplifying the peer review process The peer review process is an integral part of science. Alexandria has to incorporate it, as it is a platform that presents an alternative framework for science. However, the traditional process had to be modified as there are aspects of it that go against the platform's goals. Three of these are: its opacity, the lack of attribution to reviewers, and its rigidity and consequent lengthiness.

We tackle its opacity by permanently displaying the peer reviews of each publication and of each proposed modification directly beneath them. The author of a review is not shown alongside it, as it important that a review is anonymous. However, an aggregate list of all individuals who've reviewed a publication and proposed modifications to it is shown next to each publication in order to credit reviewers' work. Lastly, the platform does not have a traditional "editor" role, and instead any user can choose to review a publication if it has not yet gathered three reviews and they have specified they are experts in that publication's field when their account is created. The process is more flexible and efficient with these modifications, at the downside of being less robust.

5.3. UX design

As discussed previously, in Section 3, between the two project iterations our client hired a design master student, Zhuoting Wang, to both do research into the target audience's needs and design a user flow for the application to fit those. Her designs are extensive and cover functionality beyond of the scope of the project[18]. Thus, the flow of our team's application represents a simplified version of Wang's designs. The development team created the visual details of the website to be as close a recreation of the user interface design given to us as possible.

The most important user flows would be creating a post, adding discussions, proposing changes to a post, and reviewing posts. Below are the flow diagrams for these interactions:

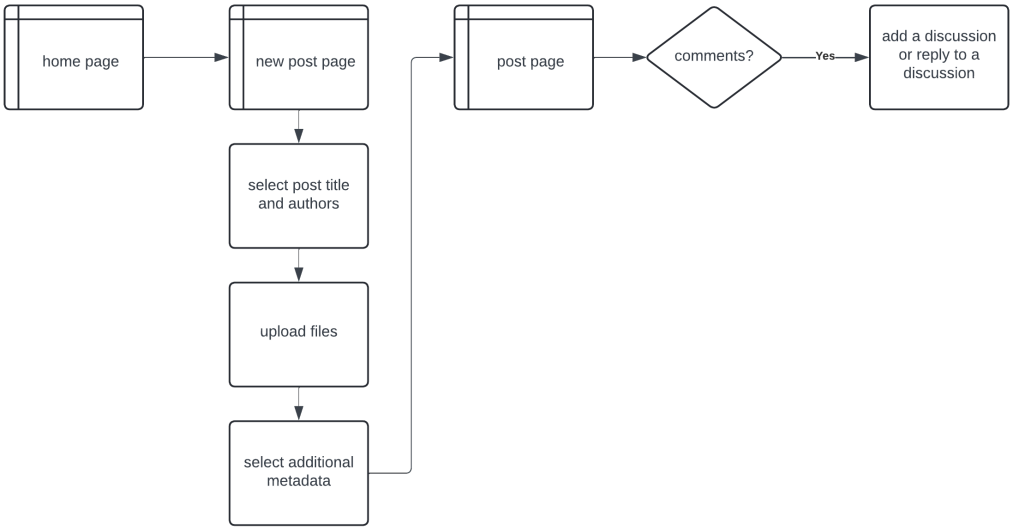

Figure 5.3: User flow for creating a post and adding a discussion. Diagram created by the team.

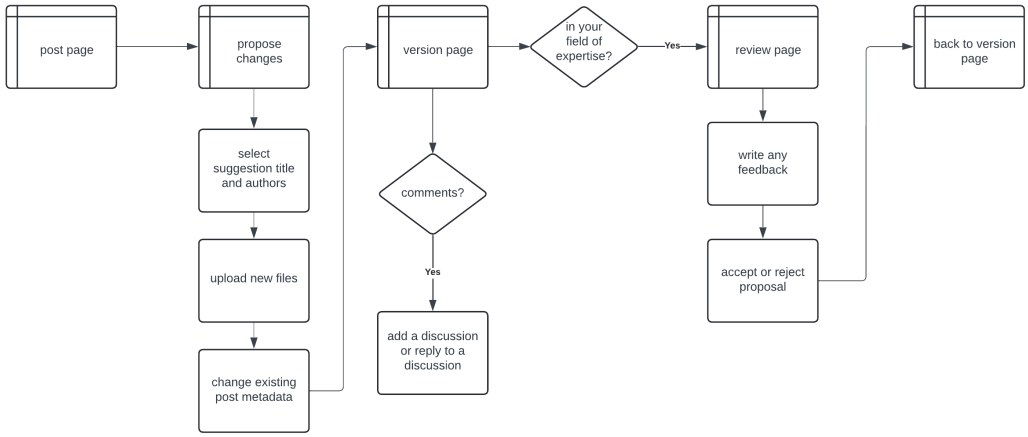

Figure 5.4: User flow for proposing changes to a post and adding a review. Diagram created by the team.

6

Implementation

This chapter presents the workings of Alexandria's prototype. First, Section 6.1 presents the technologies that were used in development. Then, Section 6.2 describes how Git was incorporated into the application. Section 6.3 details concurrency, 6.4 explains Quarto integration and finally 6.5 describes which features were implemented.

6.1. Technology stack

This section outlines the technologies, programming languages, frameworks and libraries that were used during the development of Alexandria. Subsections 6.1.1 and 6.1.2 describe the technology stack of the front end and the back end, respectively. Following that, Subsections 6.1.3 and 6.1.4 describe Git and Quarto, the two main tools used in the implementation of the prototype.

6.1.1. Front end

In order to implement the Single Page Architecture described in Section 5.1.1, the React library[1] was used. This is because it is an open-source, widely used, robust library, and thus is very well documented. Additionally, the learning curve is small, and one of the team members had worked with it in the past.

React is written in the JavaScript programming language, which is similar in syntax and structure to other languages that the team was familiar with. However, the team opted to use TypeScript, a language built on top of and that is compatible with JavaScript instead, for the additional type safety.

Since React is concerned specifically with the rendering of Document Object Model (DOM) elements, the team also used the Next.js framework[2] for handling routing, server-side rendering and static rendering. It was chosen for similar reasons to React, as an industry standard.

Furthermore, NextUI[3], next-themes[4], and Tailwind CSS[5] were used to achieve a streamlined and cohesive design for the user interface. These work together to provide, respectively, pre-made elements, and a simple way to style them, greatly speeding up the development process.

6.1.2. Back end

The back end's responsibility is two-fold: first, running a web server and serving responses to requests; second, persisting data in a database. These requirements motivated our back-end tech stack choices.

The back end was created in the Go programming language[6]. Go is type-safe, memory-safe and built for concurrency, making it ideal for a web server.

The Go language's Gin[7] package is Alexandria's HTTP framework of choice. It is widely used, has

[1] See `https://react.dev/`
[2] See `https://nextjs.org/`
[3] See `https://nextui.org/`
[4] See `https://github.com/pacocoursey/next-themes`
[5] See `https://tailwindcss.com/`
[6] See `https://go.dev`
[7] See `https://gin-gonic.com/`

a high performance, boasts easy-to-use middleware for processing requests, and provides convenient abstractions for routing.

Going further down the tech stack, Go's Gorm[8] package was used as Alexandria's ORM (Object Relational Mapper), responsible for mediating between the web server and the database. Like Gin, it has wide-spread adoption, and is very developer friendly.

As for the choice of database: we opted to use the MariaDB[9] DBMS (Database Management System), an open-source community fork of the MySQL[10] DBMS. MySQL and MariaDB are both industry standard DBMS, but the latter is more permissive in licensing and avoids Oracle vendor lock-in [19].

6.1.3. Git and Go-Git

To enable versioning of posts, we made use of the Git CLI. It is widely used and provides support for collaboration. It allows for files to have a common source of truth and for local modifications to be made and integrated into those files. Go-Git is a wrapper around git, which makes it easier to use Git without running shell commands.

6.1.4. Quarto CLI

One of Alexandria's aims is to make research reproducible. Often, papers have graphs and plots that are generated using code. Such artefacts do not make it into the final paper, and thus remain hidden from potential readers.

To address this issue, our client proposed the use of Quarto[11]. To quote their documentation[12]: *"Quarto is an open-source scientific and technical publishing system built on Pandoc. Quarto documents are authored using markdown, an easy to write plain text format"*. Quarto projects also allow for embedding code fragments. By using Quarto, all artefacts that constitute a publication can be stored, shared and rendered into a readable format.

To integrate Quarto into our project, we used the Quarto Command Line Interface[13]. More information about how it was implemented can be found in Section 6.4.

6.2. Integrating Git in the platform

To allow uploading files, we created a virtual file system (VFS), which houses all posts and version Quarto project files. Their metadata is stored in the database. Each post corresponds to a single directory, which is itself a git repository. Each version, including the main version, is stored within their respective post's repository as a git branch. For an illustration of the file structure of the VFS, see Fig. 6.1.

We created a suite of key methods to interact with repositories, which are utilized by services in order to create repositories, create branches, checkout branches, commit, merge, and rollback changes. Though Go-Git is very useful, some operations or specific settings are not available. In those cases, manual shell executions were necessary to achieve the desired result.

When merging a suggested change into the original post, we do not perform a git merge. Instead, in order to simplify git interactions as much as possible for the user, we override the main version with the approved version. This is done via a reset.

In the current state of the application, it is not necessary to use Git for Alexandria's version control system. However, since the platform is a proof-of-concept, we decided it was important for future extensibility. Features such as showing differences between versions, tracking detailed authorship, and complex branching structures are all included in the final vision of Alexandria. As they are all features inherent to Git, basing the functionality on the framework will make the implementation of these features much easier.

[8] See `https://gorm.io/`
[9] See `https://mariadb.org/`
[10] See `https://www.mysql.com/`
[11] See `https://quarto.org/`
[12] See `https://github.com/quarto-dev/quarto-cli`
[13] See `https://github.com/quarto-dev/quarto-cli`

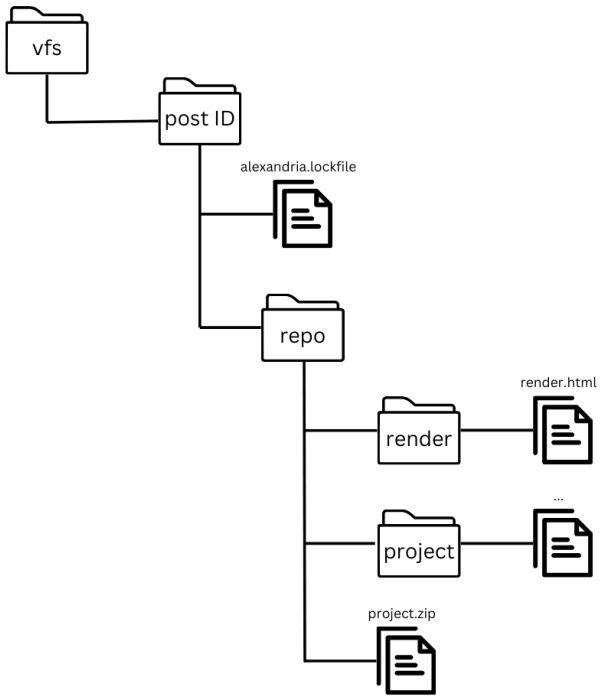

Figure 6.1: Virtual File System Structure

6.3. Concurrency and locking

During implementation, requests for resources contained in different branches of the same projects began to cause concurrency issues. This is because of switching to a new branch while there are ongoing processes associated with the previous one, as the filesystem is in the same state for all the processes.

This also caused issues with accessing posts, which were fixed by using file locks. Each file repository has a lock file, and any process that wants to access a repository must first acquire the lock for this file. Once it is acquired, until the process has finished, is is not possible for another process to access the same repository.

While this solution is effective, it is rather slow. The waiting time increases, since it disables concurrent requests for the same post. We address this issue in our recommendations in Subsection 9.2.2.

6.4. Integrating Quarto in the platform

In order to render documents from a collection of files, as described in Section 3.1, we used the Quarto command line. The process of rendering a Quarto project is described in Subsection 6.4.2.

6.4.1. Rendering a Quarto project

In order to display a readable version of a post, the underlying Quarto repository is rendered into an HTML page. This is done using the Quarto CLI.

As the users are only able to upload zipped folders, the first steps in this process include saving and unzipping the project, as well as ensuring the files include a correct yml/yaml configuration file. We verify the project type and install any missing dependencies.

Following that, to ensure the correct rendering layout and format, we inject a custom page layout into the configuration file. Then the Quarto CLI is used to render the project. A specific series of flags is used to adjust the command.

As a result of this process, a single HTML file is placed in the render folder, which is verified after the render process is complete. During this process, since the duration of it varies depending on the complexity of the repository, a pending status will be displayed on the website. Finally, the render status of the branch is set to ”success”, or if an error occurred during the process, to ”failure”.

Changes to the specific Quarto settings used for rendering may be cumbersome, as they would require corresponding changes on the front end. However, on the back end, it is easy enough to simply add or remove flags from the render command used. These flags are set to create a document with a minimalist layout, allowing it to be shaped by the website's own interface.

6.4.2. Displaying the rendered project

When visiting the page of a specific post, the rendered version of that post, in the form of an HTML file, is requested by their browser. The document is displayed within the post's page through an iframe[14], which allows embedding separate HTML file into a web page.

This procedure alone would display the same contents as the user would get by using Quarto themselves to render the project. However, since it is embedded within a website, there are some clashes with the interface.

Firstly, the document will always have a white background, which might not match the current website colors. This is fixed by adding custom CSS to the document.

Secondly, the document's container in the website will have a fixed height that is different from the document's own, cutting the document short. To address this, a JavaScript function that makes the document relay it's height to its container in the web page was created.

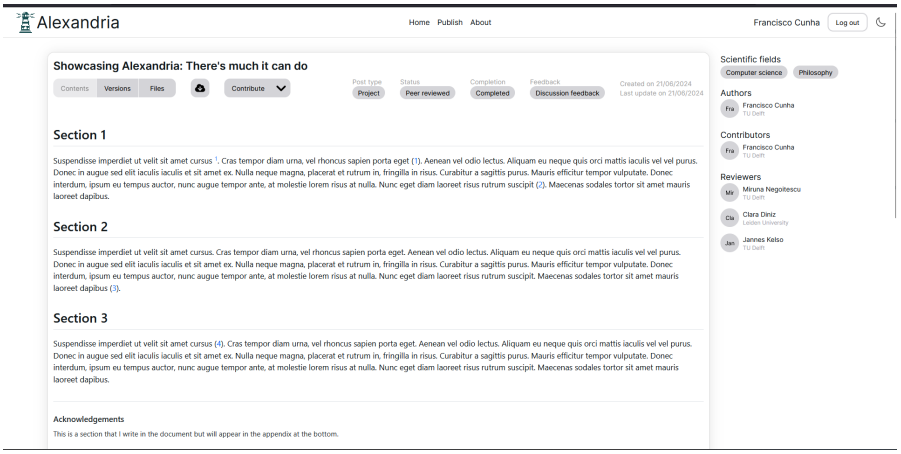

Figure 6.2: Post page

Following this process, the document is seamlessly displayed within the platform, as if it were native to it. An example of this is shown in Figure 6.2.

6.5. User feature implementation

In this section we describe in detail the features that we have implemented, the general flow of the application and how various elements interact with each other. The features are grouped into 5 big categories: posts, branches and peer reviews, discussions, homepage, and users.

Posts Posts in the platform are divided into regular and project posts. Project posts engage in the peer review process and allow iteration, while regular posts are static entities without reviews. In the interface, both are shown in similar pages - with slight differences in the metadata that is displayed.

Logged in users can create a new post by selecting the "Publish" button in the navigation bar at the top of the website. As shown in Figure 6.3, on the publish page the user can input the post's data. Before submitting, there is basic client-side validation in place, to make sure required fields are filled out and that text field inputs do not exceed the character limit. Currently, the only way to import files is by uploading them as a zip archive from the users personal machine. The reasoning behind this is described in Section 5.2. There is an additional constraint on the way files are zipped, namely the archive must be made from the files in the root folder, not the root folder itself.

[14]https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

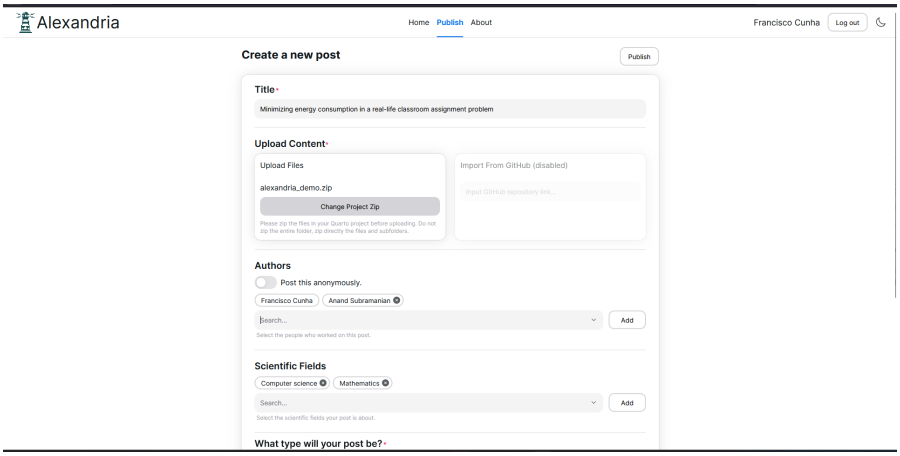

Figure 6.3: Publish page

Then, the form data is sent to the server with two requests. First everything except the file data is sent. Additional validation is performed, checking for example if the authors are all existing users and if scientific fields are all valid, then a database entity is created. If this request succeeds, the files are then sent as multi-part form data, and the Quarto project begins to render as described in Section 6.4. If this request fails, the post entity that was created gets deleted. This is due to the fact that we need a post ID to reference when we upload, but only the server can provide such an ID by creating a post.

The page for a given post in shown in Figure 6.2. The rendered project is shown at the center. To its side are a list of its scientific fields, and an exhaustive list of its collaborators split into authors (those who wrote the first version) and contributors (those whose suggested changes have been incorporated). Above it are shown it's title, preferences set when publishing, its peer review status, and buttons that allow one to interact with and contribute to a post. Available buttons include "Download" which download all the project files to the user's computer, "Contribute" which creates a new branch for that post, and "Review" which allows one to peer review a post if it hasn't yet gathered three reviews.

One can also switch to a "File view", where instead of a rendered Quarto project, the post is shown as the collection of files that make the project up. This view is shown in Figure 6.4. The files can be navigated similarly to any operating system, clicking folders to see the files inside, and specific files to see their content, as shown in Figure 6.5. Finally, a "Versions" button takes the user to a list of branches for that post, described below.

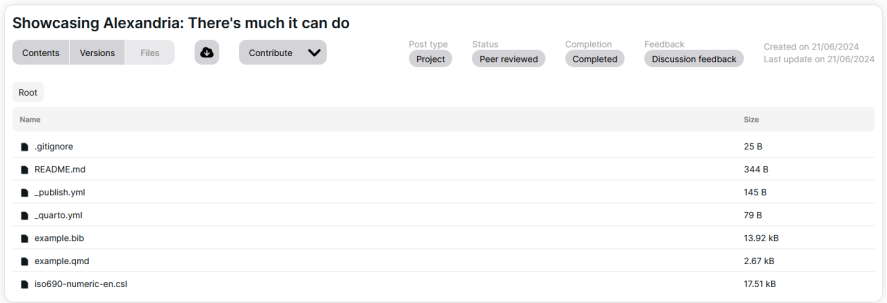

Figure 6.4: File view for a post

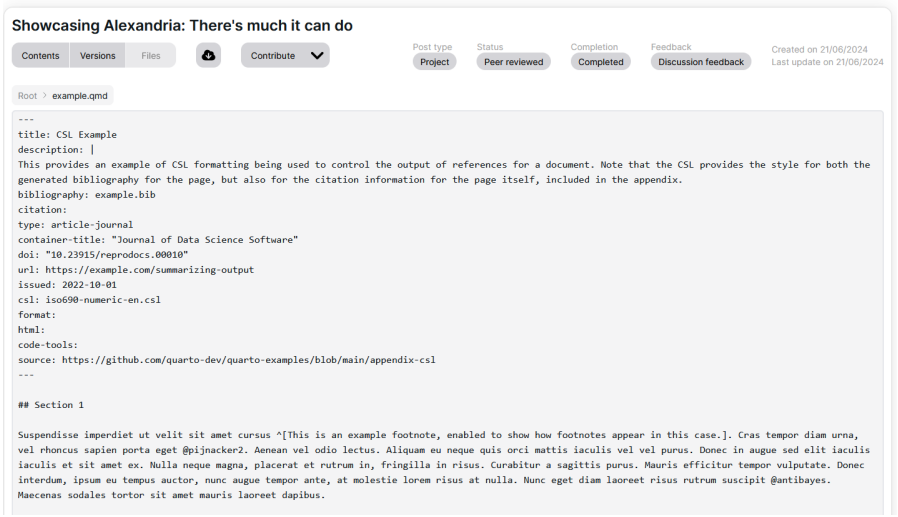

Figure 6.5: Contents of a specific file

Branches and peer reviews Branches can be created through a process similar to posts, the difference being their creation comes not from the ”Publish” button but rather the ”Contribute” button on a post, which can be seen in Figure 6.2. If the post is not a project post, its ”Contribute” button is disabled. Since a branch allows changing all aspects of a post, all their input fields are the same.

A user can choose to see all the branches of a post. The branch list is divided in three: ”Version history”, ”Rejected changes”, and ”Proposed changes” - the first showing all branches that were peer reviewed and accepted, shown in Figure 6.6; the second all that were peer reviewed and rejected; and the last all that have not gathered three peer reviews, shown in Figure 6.7. The distinction between the three is done via the property ”Review status” in a branch’s database model, and the three lists are filtered in the back end and returned via an API call.

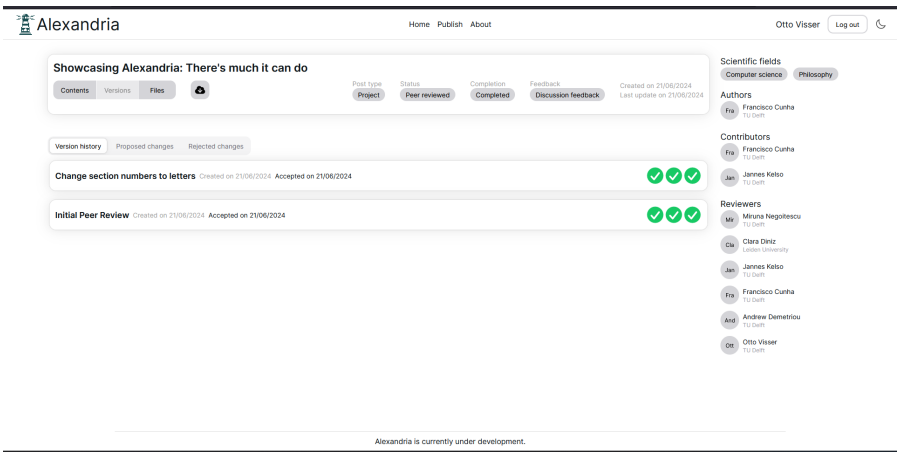

Figure 6.6: Version history of a post

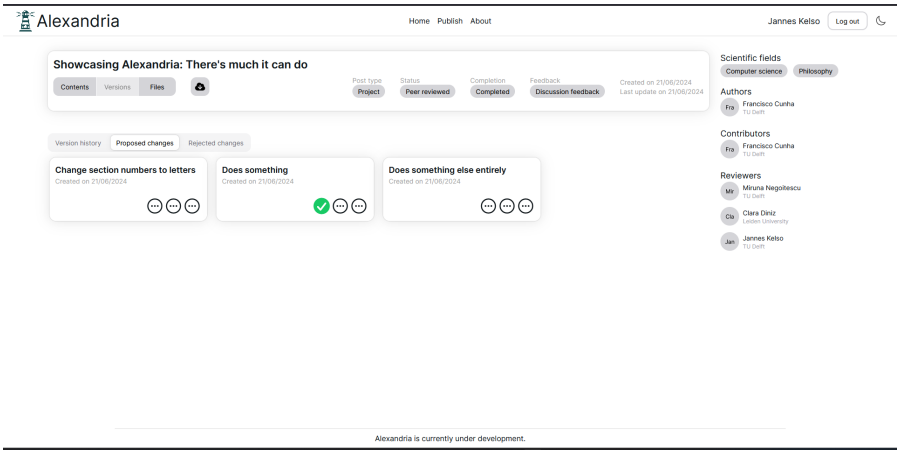

Figure 6.7: List of a post's proposed changes

Selecting a branch takes the user to that branch's page, shown in Figure 6.8. It has similar contents to the post page, with two main additions: first are the peer reviews shown beneath the rendered project with their feedback and decision, shown in Figure 6.9, and second is a "Compare" switch which when selected allows displaying both the replaced and new project side by side.

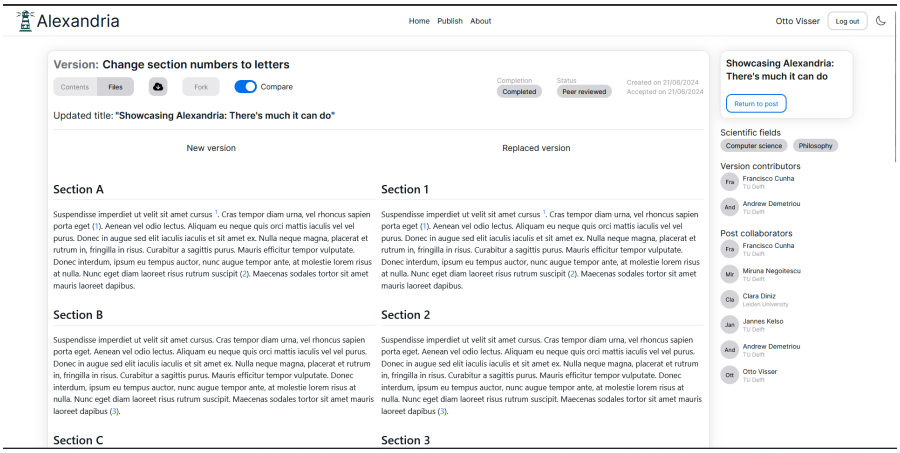

Figure 6.8: Branch page

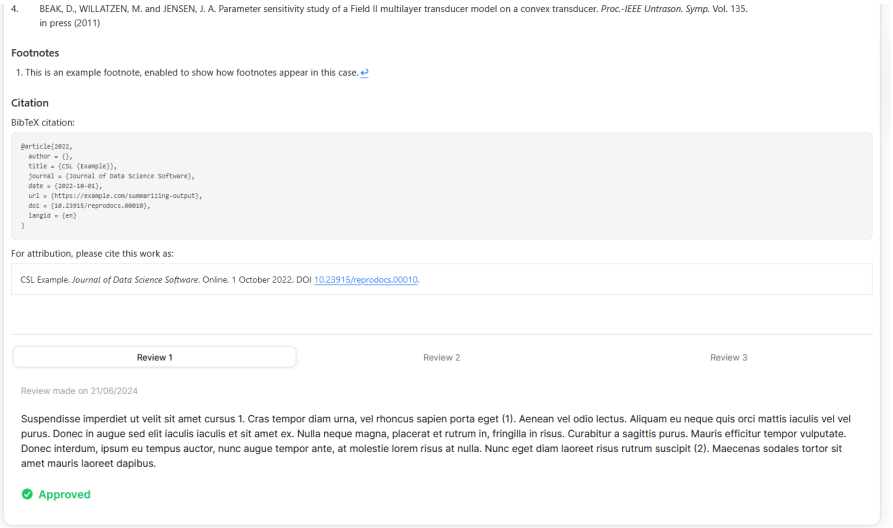

Figure 6.9: Peer reviews for a branch

To review a branch, a user will be redirected to the branch review page, where they can write their feedback and select ”Approve” or ”Reject”, as shown in Figure 6.10. Forms are used to submit the review to the back end, with basic client side validation to make sure the review field is filled out and that it is either mark as approved or rejected. When a new peer review is created, the back end will check if it is the third review for that branch and, if so, update the branch database entity to reflect its review status as rejected or accepted. The same is done for the relevant post in case that is the post’s first peer review. The back end rejects and ignores any attempt at adding a peer review past the third.

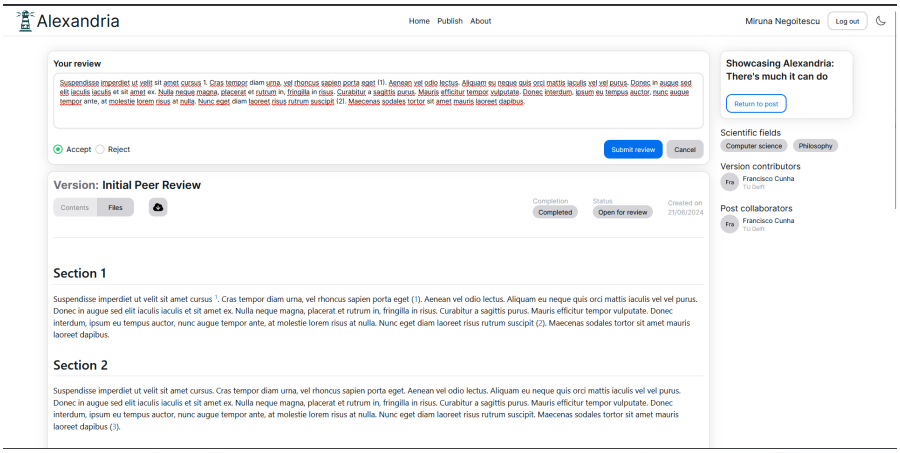

Figure 6.10: Reviewing proposed changes

Homepage The homepage is displayed in Figure 6.11. It shows a list of posts, each including their title, review status, list of authors, type and scientific fields. All the posts shown are ordered by decreasing publication date. The page initially shows a limited number of posts, which are requested through a paginated endpoint. At its bottom is a button labeled ”Load more posts” which, when clicked, requests the next page of posts from the back end and adds them to the bottom of the list.

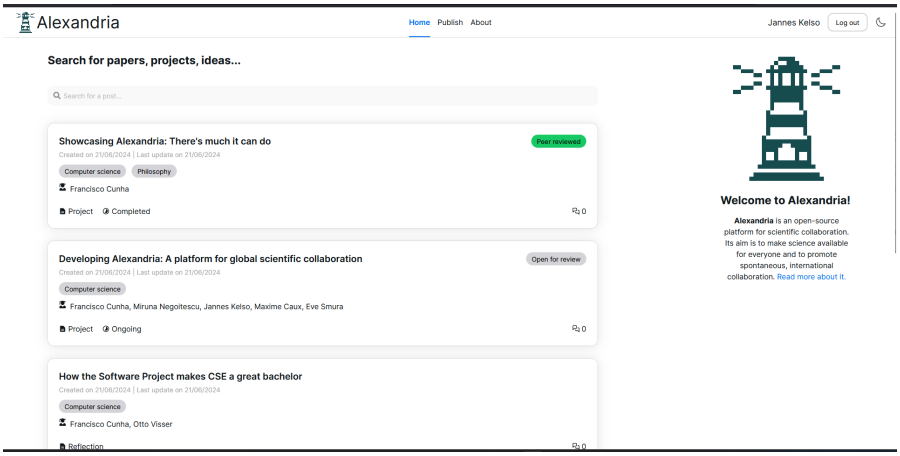

Figure 6.11: Alexandria's homepage

Discussions A list of discussions is shown beneath the rendered project in both the post and branch pages. At its end one can find an input box that allows creating a new discussion. Discussions follow a nested structure: each discussion holds a list of other discussions which are replies to it; discussions that do not reply to another, but rather directly to a post or branch, are nested within a "discussion container", an entity that each post or branch has one of and serve to get deal with these "root" discussions.

Users This section describes the process of creating a new account, logging in and out, as well as user authentication.

When creating a new account, a user must specify personal details - names, their scientific institution and relevant scientific fields; as well as account details: email address and password.

The front end performs basic verification on the password and email. This information is then sent to the back end in plain text, which is a security concern addressed in section 9.2.5.

A new user is created and the password is hashed, using the bcrypt algorithm [20]. When the user logs in, first we find the user associated with the given email and then compare the hashed password with the password they provide.

Further user authentication is done through a simple system of session tokens - when a user logs into the platform, they are assigned a unique access token and refresh token, which are stored in browser session cookies, along with user id, name and email. The access token is then sent in any further requests to determine whether the user is allowed to take certain actions.

As an example, a user needs to be authenticated in order to create a post. Another use of authentication is that a member cannot approve their own proposed changes.

Because the access token is used so frequently, and therefore more likely to be intercepted, it expires after only 15 minutes. At this point, the refresh token will be used to request a new pair of refresh and access tokens. This allows the user to continue their session without needing to log in again.

The refresh token expires after 3 days, because it is used infrequently and less likely to be intercepted. Upon logging out, both tokens are deleted. These tokens are implemented using the JSON Web Token standard [15].

[15] https://jwt.io/introduction

7

Asserting the Quality of the System

This chapter describes the software testing and verification processes we used during development of Alexandria. Proper testing is a critical part of software development: it ensures that code is written correctly and performs as expected. In Section 7.1, we detail the *unit tests*: small compartmentalized tests on individual pieces of functionality. We describe the *integration tests* - tests using the full tech stack - in Section 7.2.

7.1. Unit testing

In this section, we detail the unit testing process and the obtained results. Unit testing can be summarized as follows: one selects a piece of code that has a single responsibility. Then, one methodically checks whether the outputs of this code meet expectations. This entire process is automated, allowing one to regularly check the correctness of the entire code base.

This section is structured as follows: Subsection 7.1.1 describes unit tests implemented on the front end. Then, Subsection 7.1.2 covers unit testing on the back end.

7.1.1. Front end

Test for the front end are done using Jest[1] and React Testing Library[2].

As discussed in Section 5.1.1, the front end is split into reusable UI elements called components. The components themselves are built by combining multiple native React components, as well as components imported from NextUI (see Section 6.1.1). As such, traditional unit tests for components are not possible. However, the individual components are tested by making sure all expected elements are rendered, and that interacting with the components (clicking, inputting text) produces the expected behaviour, with the help of mocked functions. Additionally, snapshot testing was employed. This which involves capturing the exact HTML generated by a component to ensure it remains the same to avoid unintended UI changes.

It is important to note that functions that interact with the API do not contain any logic, and simply rely on native functions for data fetching and JSON conversion. Testing them would mean mocking nearly their entire codebase, essentially only testing the mocks. As such, functions in the */app/api* folder are not tested, as it was not considered relevant.

Additionally, as the pages themselves represent compositions of native React elements and custom components, which are tested individually, it was also not considered relevant to test any *page.tsx* files.

Apart from the files mentioned above, all components and utility methods were tested. The team aimed for at least 80% branch coverage for all front end code. The final coverage for the entire repository is shown in Figure 7.1.

[1] Jest Testing Framework `https://jestjs.io/`
[2] React Testing Library `https://testing-library.com/docs/react-testing-library/api/`

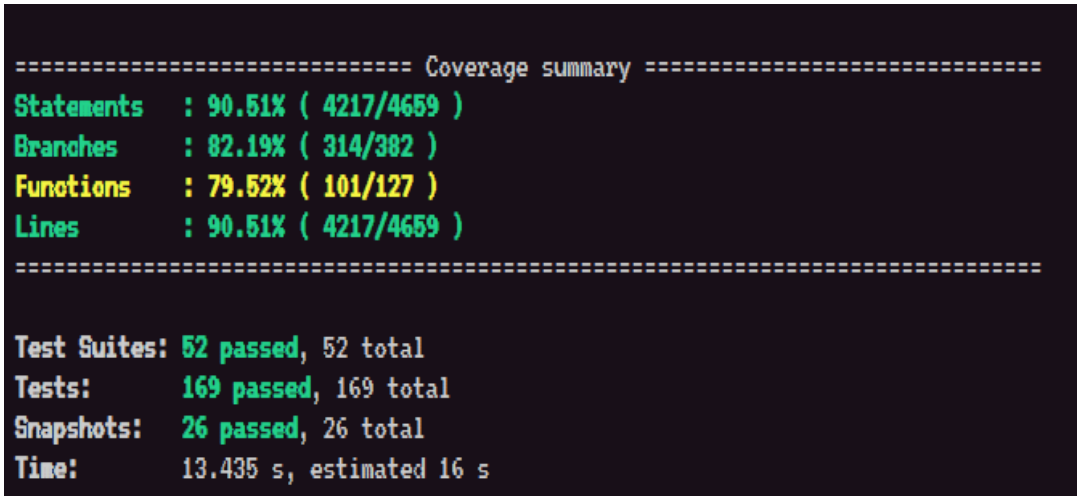

Figure 7.1: Front end repository test coverage.

7.1.2. Back end

We used Go's[3] native `testing`[4] package for all back-end testing. It the facilitates creation of simple automated test suites, including test setup and tear-down. It was chosen for its wide support and usage in the Go community. The tests are structured according to the `testing` package's recommendations: test files sit next to the files they test, and are suffixed with `_test`.

Most back-end unit tests cover the *controllers* and *services* described in Subsection 5.1.2. This is because they contain the majority of the business logic, and are therefore the most important components. In addition, we thoroughly tested the data models and *DTOs* described in the same subsection.

Truly compartmentalized testing of code requires *mocking*. Code under test usually has dependencies: for example, testing post creation requires a database to create the post in. Mocking injects 'fake' objects as dependencies, and gives the test control over them. This keeps unit tests clean and containerized, only testing what is *strictly* necessary.

We integrated mocking into Alexandria's back-end test suites using Go's `gomock` page[5]. It is widely used and has broad support, making it the first choice. Mocking was used in virtually every unit test, given the deeply interconnected nature of the application.

The test suite achieved a back-end test coverage of 79.72 percent - 0.28 percent below our goal as decided during project planning, which is acceptable. *Statement coverage* was used as the coverage metric, as it proved to be the easiest way to get the suite running.

7.2. Integration testing

This section details integration tests: tests that verify code correctness while running the full Alexandria system, without any mocked ('faked') components. Subsection 7.2.1 describes integration tests on the back end, which primarily consist of database interaction tests. Subsection 7.2.2 covers front end integration tests.

7.2.1. Back-end database tests

The database is the most difficult component to integrate in back-end testing, because it is a separate program and hosts permanent data. As such, there exists a separate *testing database* that is spun up alongside the regular database. This allows one to create, update and delete test objects without affecting the data storage of the application. The contents of this database are cleaned after every test.

The test database was used to test our *model repositories* described in Subsection 5.1.2. Integrating with a test database, instead of mocking it, helped identify several flaws in the team's assumptions about database interactions. This was key to fixing several problematic bugs in Alexandria's review and

[3] See Subsection 6.1.2 for a description of Alexandria's back-end tech stack.
[4] See `https://pkg.go.dev/testing`
[5] See `https://pkg.go.dev/github.com/golang/mock/gomock`

branch merging process. Additionally, these tests contributed to our statement coverage percentage as described in subsection 7.1.2.

7.2.2. Front-end tests

The quality of the integration between the front end and the back end, which was done during the final week of the project, was verified through merge request reviews. The team members assigned to review a feature ensured it works by using the website in real time. By interacting with the features in the same way a user would, we tested the general flow and usability of the application.

It was attempted to test the integration of the front end with the back end in a reproducible manner using the Cypress framework [6]. Integration tests were created for the pages for user login, creating a new user, the about page, as well as the pages for creating and viewing a post. However, since we were not able to include the integration tests in the main branch of the front end code base at the time of writing this report, it is not included here.

[6] Cypress Testing Framework `https://www.cypress.io/`

8

Ethical Considerations

This chapter is dedicated to analyzing the ethical implications of our project and possible solutions thereof. While the creation and existence of Alexandria do not pose many ethical quandaries, and our team strongly believes the mission of the project is highly beneficial to the scientific community and the world at whole, there are still issues worth considering. Section 9.1 will deal with issues caused by data storage, while Section 9.2 discusses social considerations like authorship.

8.1. Data management

This section deals with the ethical concerns associated with the way Alexandria stores and manages data.

Personal data It is important to carefully consider personal user data. Alexandria only stores data filled in by users during profile creation/editing. The data is never archived, and is not given or sold to any third parties.

However, Alexandria does store user information. Due to the time limitations of the project, we will not be implementing robust and secure data storage and protection, leaving it as a priority consideration for future development. While this would be a serious ethics issue in a fully functional platform, the current data security is sufficient for a proof-of-concept, as it will not be in broad use and any potential users for the current platform are assumed to be of good-faith.

It is recommended for future developers continuing this project to implement the security described in Section 9.2.4.

Licensing Due to the open-source, community-based nature of the platform, any project posted on the website is licensed under the CC BY 4.0[1]. An ethical consideration is ensuring that all users are aware of this. Another possible solution would be allowing users to select which type of license they wish to publish their work under. However, it remains out of scope for this iteration and potentially goes against the spirit of the project.

Data transparency To encourage repeatability, data artefacts are uploaded to the platform as part of publications. These are stored in the Alexandria servers and are completely open to the public. This current approach for the proof of concept is not sufficient long term, as studies done on sensitive or protected data must omit their code for ethical and legal reasons.

To join reproducible results with protected data, we look to the future, as there are currently systems under development that would allow university libraries to host data and allow API access to it to allow queries, without making it accessible to the general public. As with other considerations in this section, while a serious concern and worth debating, it is not our priority within this project.

[1] https://creativecommons.org/licenses/by/4.0/

8.2. Challenges to societal values

This section explains the ethical issues concerning the societal and personal aspects of Alexandria - namely the concept of shared authorship and the choices of platform moderation.

Publication authorship It is integral to Alexandria's vision that publication authorship is not attributed to the few individuals who first created it, but rather to an evolving list of contributors that grows as more improvements are made to articles. This challenges society's notions of authors having ownership over their writing, but it is a core aspect of the system that aligns closely with principles of open science. Moreover, individual contributions to a publication are tracked via the version control system, such that due attribution can be made to each collaborator.

Platform moderation An additional consideration is platform moderation. It is needed, as large scale usage of such a platform might lead to users writing harmful comments, vandalizing publications, or similar actions that lead to an unproductive virtual environment.

However, on a platform containing scientific research, a moderator would have power over science itself. This raises crucial ethical questions about the authority to appoint moderators, as well as the individual ability to be one.

Here we can look to a platform that has inspired Alexandria: Wikipedia. There any user can become a moderator (called *Administrators*) by being "active, regular, and long-term Wikipedia editors, be familiar with the procedures and practices of Wikipedia, respect and understand its policies, and have gained the general trust of the community." [21] and submitting a candidature that is approved by the platform's own community. Alexandria's peer review systems could be used for such voting, as well as for solving any possible disputes between users and moderators.

9

Result Discussion and Future Recommendations

This chapter presents the outcome of this iteration of the Alexandria project. First, Section 9.1 analyzes the product itself, and what it achieved. Section 9.2 lists our recommendations to developers who work on Alexandria in the future.

9.1. Final product analysis

This section provides a short overview of the implemented requirements. The application includes all of the Must Have requirements (23/23), as well as half (13/25) of the Should Have requirements. See Appendix D for the full list of requirements.

Implemented requirements We now list the implemented requirements, in full written form:

- A user can sign up for a 'member' account, providing personal details and scientific fields. Members can log in and out of their account.
- Members can create posts and project posts, specifying the scientific fields, feedback preferences and completion status. They can specify an author list, or keep it anonymous.
- Members can view posts and their metadata. The server renders posts using Quarto, as described in 6.4, and this render is displayed as interactive HTML on the post page. Members can view and browse the file tree from which the project was rendered, view the raw contents of individual files, and download project files to a zip archive. They are also able to start a discussion on a post or respond to other discussions, with the option to remain anonymous.
- A member can propose new changes to project posts. This is done by augmenting the author list (although they can choose to contribute anonymously), uploading new contents, and specifying changes to the post's metadata.
- There exists a simple peer review process. Members with relevant expertise can choose to review proposed additions, approving or rejecting them. Proposals that have gained three approvals override the main post's contents and metadata, allowing for collaboratively developed "living" projects. This is implemented on the back end using Git, as described in Section 6.2.

Assessment of final product The goal of this iteration of Alexandria's development process was to create a proof-of-concept to showcase the feasibility and main advantages of an online, open-science publication platform, as per Section 3.1. The most important requirement, namely using Quarto for rendering, as well as displaying the project artefacts, has been achieved. Furthermore, most requirements have been met. The code is also well documented via comments, and all features and bugs

are documented on the project Wiki[1], making it easy for future developers to pick up where we left off. Thus, we consider this iteration of the project a success.

9.2. Future work

The final vision for Alexandria is far greater than what was implemented. As our team has only implemented a proof-of-concept for this platform, many points of improvement remain. This section outlines the most relevant points.

9.2.1. Improvements to user flow in system

This section introduces possible improvements to the overall usage of the system.

Implementing remaining requirements A good starting point for future developers continuing the project would be the original list of requirements we have created. Implementing remaining points from the Should Have and Could Have sections of this list would greatly enhance the user experience. Features such as a user page, the ability to preview files before posting and forking possibilities would be a good first step for further enhancing the user experience.

Improvements to the peer review process The current peer-review implementation is a simplified version of the process used by the scientific community, created for demonstrative purposes and to enable further research. There are many guidelines and restrictions which would need to be met for the process to meet academic standards, such as official publishing formats and credible reviewers. This is something that future developers should explore. In this regard, another concern is verifying member expertise, as this is currently not moderated.

Quarto Rendering Quarto supports rendering with Python, R, Julia, and Observable, and their various libraries, and additionally supports a host of Quarto add-ons. All of this is supported in Alexandria, but all relevant libraries have to be included in the quarto project that is submitted to Alexandria, besides R libraries, which are automatically installed according to an renv.lock file[2]. Further automatic dependency installing has not been provided, due to lack of time. This would be a nice feature to add in the future, as it lowers to complexity and the current way this works may be confusing to less tech-savvy users.

9.2.2. Improvements to system architecture

This section presents suggestions for improvements to the overall architecture.

Front end and back end communication A good web application requires a well-built data model and a consistent contract of communication. While the current API specification is well defined, it has several issues we discuss here: a) a design mismatch between front end and back end related to data ownership and b) duplication of data and functionality.

First of all, the front end and back end adhere to different data model design principles. This is a consequence of the languages they are written in - typescript allows for 'union' types, similar to a *base class* that you inherit from in object oriented languages - while Go has no inheritance at all. As a result, in the back end, a 'project post' encapsulates a 'post', and acts as the owner of their association. However, in the front end, a 'post' could either be a regular post *or* a project post, meaning that ownership is inverted: *the post owns the project post*. This led to significant integration slowdowns. Future work on Alexandria must reconsider how 'inheritance' and association ownership is represented in the back end and communicated to the front end.

Secondly, the back end duplicates file and render functionality on posts and branches, due to the relational database's strict constraints. Consequently, their respective API endpoints also contain duplicate implementations. Future work on Alexandria should unify these implementations under one consistent one.

[1] Alexandria Wiki `https://gitlab.ewi.tudelft.nl/groups/cse2000-software-project/2023-2024/cluster-v/17b/-/wikis/home`

[2] These are the standard to specify dependencies

Concurrency and version control How to store and render Quarto files under version control is one of the Alexandria prototype's leading research questions. While a basic implementation has been successful, many design considerations follow. In this subsection, we consider these questions.

Each Quarto project is initialized with a Git repository, to keep it under version control. Many user actions, however, require interacting with this repository directly - such as viewing a different version than is currently loaded. The current issue is that *only one Git user exists per project* - meaning one cannot have two different versions of the same post loaded.

This iteration's solution is to use file locks, to allow for asynchronous requests, but still process them in a synchronous manner. This solution is not scalable, as a larger volume of users will cause far longer response times. Future work on the Alexandria project should consider how to implement local asynchronous access to Git repositories, and research how Git providers such as GitLab implement it. An example suggestion is to use the *git worktree* feature, in order to checkout different version of a post to different directories at the same time. We only discovered this feature at the end of the project, so there was no time to refactor to this design.

9.2.3. Improvements to security

This section highlights known issues with security in Alexandria.

9.2.4. Data protection and storage

One of Alexandria's core aspects is the ability to share artefacts, as described in Section 3.1.2. This arises the problem of users having to upload very large databases, as well as databases containing confidential data. This presents and ethical concern, as discussed in Section 8.1. To address this issue, the vision is for future developers to collaborate with university libraries, and use their data servers to offer large quantities of storage space, as well as data protection. Moreover, terms and conditions should be elaborated, and users should be made aware of what data is collected from them.

9.2.5. Cybersecurity concerns

As mentioned previously in Section 5.2 security concerns were not a priority for the team for this prototype. However, since Alexandria is meant to be a public, online platform, security is of crucial importance for the final product. We only outline a few key points the team has identified.

To start, the platform will need better input sanitation to protect against SQL injection attacks. Another security improvement would be to sandbox the rendered Quarto projects on the front end, as they contain runnable code. Sandboxing is a technique by which applications are run in isolation, meaning they are run on the host machine but with heavily restricted access to its resources. Thus, by sandboxing project iframes, the client machine and servers would be further protected from malicious code.

Lastly, account data could be encrypted, and passwords could be hashed using salts. Currently, account data is stored in plain text, and passwords are hashed using bcrypt[20]. Encrypting all the data would add an extra level an attacker would have to bypass, should they gain access to the user database. Additionally, all data is unencrypted while being sent to the back end, which should be changed in the future.

10

Conclusion

The goal of this report was to account for the decisions made in the development of a proof-of-concept of Alexandria: a platform to host scientific publishing in a way that lessens the rigidity of traditional publishing, allows for spontaneous collaboration between unknown parties and makes publications and related artefacts, including their reviews, open and transparent.

The created platform is a web application: an interface that can be accessed from a web browser (the front end) and a server application that communicates with the interface to manage information and handle logic (the back end). In Alexandria a user can: create publications using the Quarto format, which contain more data than a traditional PDF; suggest modifications to publications, which can be incorporated after being peer reviewed; review publications and proposed changes, accepting or rejecting them; participate in detailed yet informal scientific discussions about a project; browse through all the publications; and see the publication that was rendered from a Quarto project, as well as the source files that underlie it.

The resulting application demonstrates the platform is feasible, although shortcuts were taken to create it in the limited time frame. The main difficulty faced was using Git to manage different versions of a publication, as that tool's strategy of replacing available files complicated simultaneous access. This was solved by using locks that block access to files while Git handles them. A number of other issues remain unsolved, which is considered acceptable for a proof-of-concept version. Future development of the platform should fix cyber security vulnerabilities, specially those relating to running user-written code; and improve the system's architecture to ameliorate stability and response times, specially in operations that use Git.

In closing, Alexandria demonstrates that by leveraging modern technologies, a new framework for scientific publications is possible: one that is more flexible, connected, and transparent.

Bibliography

[1] V. M. Nguyen *et al.*, "How long is too long in contemporary peer review? perspectives from authors publishing in conservation biology journals," *PLOS ONE*, vol. 10, no. 8, Aug. 2015. DOI: 10.1371/journal.pone.0132557.

[2] *Elsevier records 2% lifts in revenue and profits*. [Online]. Available: https://www.thebookseller.com/news/elsevier-records-2-lifts-revenue-and-profits-960016.

[3] D. J. Solomon and B.-C. Björk, "A study of open access journals using article processing charges," *Journal of the American Society for Information Science and Technology*, vol. 63, no. 8, pp. 1485–1495, Jul. 2012. DOI: 10.1002/asi.22673.

[4] S. Kyvik and I. Reymert, "Research collaboration in groups and networks: Differences across academic fields," *Scientometrics*, vol. 113, no. 2, pp. 951–967, Sep. 2017. DOI: 10.1007/s11192-017-2497-5.

[5] A. M. D. C. C. S. Liem, "Treat societally impactful scientific insights as open-source software artifacts," *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2023.

[6] G. Naujokaitytė, *Number of scientists worldwide reaches 8.8m, as global research spending grows faster than the economy*. [Online]. Available: https://sciencebusiness.net/news/number-scientists-worldwide-reaches-88m-global-research-spending-grows-faster-economy.

[7] M. Hagve, "Pengene bak vitenskapelig publisering," *Tidsskrift for Den norske legeforening*, 2020. DOI: 10.4045/tidsskr.20.0118.

[8] Z. Wang, *Alexandria user research presentation*, 2023. [Online]. Available: https://www.canva.com/design/DAF0V3qlh7E/mC6KrCjE45-G4xExNkZBmg/view.

[9] Z. Wang, "Alexandria ux research report," Tech. Rep., 2023.

[10] *Wikipedia contributors.* Apr. 2024. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia.

[11] A. van der Meijden, M. de Wit, J. Sloof, E. Witting, and A. Zlei, *Alexandria: Proof-of-concept publication platform that treats academic outputs like software artifacts*, 2022. [Online]. Available: https://github.com/prjct-alexandria/Alexandria.

[12] *"what is moscow prioritization?"*. [Online]. Available: https://www.productplan.com/glossary/moscow-prioritization/.

[13] *What is scrum?* [Online]. Available: https://www.scrum.org/learning-series/what-is-scrum/.

[14] *"client-server oveview"*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview.

[15] *"single-page applications (spas) — what they are and how they work"*. [Online]. Available: https://business.adobe.com/blog/basics/learn-the-benefits-of-single-page-apps-spa.

[16] *"single page applications (spa) vs. multi-page applications (mpa)"*. [Online]. Available: https://medium.com/stackanatomy/single-page-applications-spa-vs-multi-page-applications-mpa-b597eeb78b78.

[17] *"spa (single-page application)"*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

[18] Z. Wang, *Alexandria ux design flow*, 2023. [Online]. Available: https://www.figma.com/design/TbrqYaCl0Adc6ND2BvZSVO/Alex-design?node-id=0-1.

[19] *Mariadb vs. mysql*. [Online]. Available: https://mariadb.com/database-topics/mariadb-vs-mysql/.

[20] N. Provos and D. Mazieres, “A future-adaptable password scheme,” *The USENIX Association*, 1999.

[21] May 2024. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Administrators#Becoming_an_administrator.

A

Work Matrix

The following table gives an approximate distribution of the work done on Alexandria:

Table A.1: Alexandria team work distribution

Who (front end)	**What (front end)**
Miruna 40%, Francisco 60%	Home page
Miruna	Create a post page
Francisco	Post page (Quarto rendering, file tree, post metadata)
Francisco 70%, Miruna 30%	Discussions
Miruna	Create branch page
Francisco	Branch page
Francisco 80%, Miruna 20%	Reviews
Miruna	Signup
Miruna	Login (UI, Session cookies)
Francisco 95%, Miruna 5%	Navbar and footer
Francisco	About page
Who (back end)	**What (back end)**
Maxime	Set up database
Maxime	Write database repositories
60% Maxime, 25% Jannes, 15% Eve	Create database models
50% Maxime, 40% Jannes, 10% Eve	Write services
55% Maxime, 30% Jannes, 15% Eve	Write controllers
Eve	Scientific field tags
Jannes	Git / filesystem interactions
Jannes	Quarto rendering
Jannes	Branch merging
60% Maxime, 30% Jannes, 10% Eve	Unit testing
80% Eve, 20% Maxime	API specification
70% Maxime, 15% Jannes, 15% Eve	Manual integration bug testing
Eve	Integration tests
Eve	Communication
15% Maxime, 15% Jannes, 70% Eve	Report writing

B

Code of Conduct

Group:	17B
Team name:	Metal Bar Clanging Sound, Yippie! (MeBaClSoY)
Team members:	Eve Smura, Francisco Cunha, Jannes Kelso, Maxime Caux and Miruna Cosmina Negoițescu
Coach:	Prof. Benedikt Ahrens
Teaching Assistant:	Ana Marcu
Client:	Andrew Demetriou and Cynthia Liem of the Multimedia Computing Group TU Delft
Course:	Software Project

Shared team values

1. Communication - the team values this very highly, as we believe it will allow for effective work distribution and collaboration. Therefore, we will prioritize maintaining effective channels of communication at all times, including daily updates, a dedicated channel for questions, and weekly meetings.
2. Honesty - the team believes that having a clear idea of the general status of the project is essential to success. We also believe that being completely transparent with the client and every team member will allow us to avoid unrealistic expectations and keep a clear overview of how things are progressing.
3. Kindness - we believe in a people-first approach, which includes ensuring everyone is treated kindly and fairly.
4. Trust - being able to rely on team members for support and having trust in their well meaning makes working together much easier and enjoyable.
5. Respect - we believe that treating everyone's opinions as valuable and respecting the input of all team members will allow everyone to feel respected.

Assignment description

Our goal in this project is to create a working proof-of-concept for the assignment given by our client, which is a collaborative open-science platform where users are able to post their research and collaborate with others, as well as participate in discussions with other members of the community. This includes being able to leave comments on posts and a peer review process to enable community-based verification of research. The exact requirements can be found in the project plan.

Target or ambition level

Our priority is creating a basic version of the platform, the details of which have been agreed upon with the client. This is represented in the must-have requirements. We aim to also implement the features detailed in the should-have section of the requirements. Since they were created in close cooperation

with the client, our goal is primarily to deliver a product that is in line with the client's vision and performs as expected. Additionally, we would like to receive a grade of at least 8 for the course.

Products

At the end of the project, we should deliver a working proof-of-concept platform, documentation that allows other teams to continue its development, as well as a technical report detailing our work process and the decisions made in the course of the project. We will use the GitLab wiki to share documents internally, and send important ones to the client through our main communication channel: Mattermost. While drafting documents, we edit them collaboratively in CodiMD.

Planning

Our planning involves using a modified version of Scrum, overseen by Miruna who has been designated as the scrummaster. We will have one week sprints, and a weekly sprint retrospective and planning. In these meetings the team will oversee the tasks accomplished in the previous sprint, decide if we are progressing at a satisfactory rate, and decide on tasks for the next week. This will be aided by a dependency graph we have created, as well as the fact we have prioritized our tasks using the MoSCoW model. We will be using a built-in issues tracker contained within GitLab, the platform our code repository is housed in. Each team member is responsible for tracking their own issues, overseen by the scrummaster. The scrummaster will also oversee epics and milestones, to keep a broader overview of the general progress.

Francisco will submit deliverables when needed, and in case he is unavailable he may contact Eve and she will submit them in his stead. We will use the process outlined under section ”Decision making” to decide if a document is good enough.

Behavior

All team members will act kindly and with understanding towards each other. We want to foster a judgement-free atmosphere in the team, and ensure that everyone feels safe, comfortable and supported. We also want to encourage asking questions and sharing progress. All team members are free to ask for help with their individual tasks and the whole team is commited to making sure no one feels discouraged or judged. We would also like to take into account personal circumstances from team members, and prioritize the wellbeing of every person involved in the project over strict punctuality and detailed rules.

However, in the event of a team member acting in a way that cannot be excused by personal circumstance or causes distress to other people, we will handle it in the manner detailed in section ”Dealing with conflicts”.

One-on-one conflicts can be resolved by invoking a third team member as mediator, and if the conflict is related to the entire team, the issue shall be brought up during a Sprint retrospective.

Communication

Our team will use a Discord server for internal communication, which includes dedicated channels for questions, shared resources and daily updates. To communicate with the client, the course staff and the teaching assistant, we will use Mattermost. Communication with the coach will be done through email.

Commitment

In general, the quality of each team members work will be evaluated via code reviews and the impression gained by the other team members. The chair and minute taker were chosen based on their qualification and commitment to the role, so we do not foresee issues in that regard. If there is an issue with the quality of their work, this will be addressed in the same way as any other conflict within the team. Issues can also be brought up in sprint retrospectives and if meetings are ineffective we will have a brief feedback round on how to improve meetings.

Division of tasks and roles

Two team members have volunteered to act as the chair and minute-taker, which the other members of the team have no objection to. Unless issues arise, these roles will not rotate, to allow for routines

and to gradually gain experience. There are also others prepared to assume those responsibilities if these team members are unavailable or decide they no longer wish to fulfill these roles.

Meetings

The team will attempt to work together on campus at least 3 days a week, including one day dedicated to writing the technical report. To allow this, we will attempt to book project rooms for those days. In addition, there will be a team meeting every Monday for a sprint retrospective and planning session, as well as a weekly meeting with the teaching assistant.

Every Wednesday we will also meet with the client. There is also a shared agenda document available for any scheduled meeting, so that any team member can bring up relevant points.

In case someone is late for a meeting, the team will wait up to 15 minutes before starting without that person. If the chairman is late, Eve shall take the role of chairman and if the notetaker is late, Miruna shall take notes. Additionally, if the client requests a spontaneous meeting, one of the team members will try to reach out to all others - the meeting will start after 10 minutes if and only if at least one other person is present.

Decision-making and dealing with conflicts

Decisions will be made by the whole team. Since we value the opinion of every team member, we will make every effort to reach all decisions by consensus. If there are disagreements or conflicts, every team member will present their perspective and we will attempt to reach a consensus through a discussion. If we are unable to reach an agreement, the team will look to external sources, for example do additional research into the matter, ask the client for clarification if applicable, or consult our coach and teaching assistant to see if they are able to offer guidance. If all of these measures still do not result in the team reaching a consensus, the decision will be made by majority vote.

Guidance

As a general rule we would like to be independent and self-directed, to remain true to the spirit of the course. However, we would like for the teaching assistant to warn us if any of the decisions made by the team or the general state of our progress seem unsatisfactory. We also hope that the coach will be able to offer technical advice and guidance when needed, and provide a different point of view to point out flaws in the design that we may have missed. In addition, we would like to refer to the team coach and the teaching assistant for feedback on deliverables.

Consequences

If a member of the team violates this code of conduct and does not keep to the rules the team has set, we will first attempt to resolve the matter internally by addressing it in one of the team meetings. If this does not prove effective, we are prepared to involve the teaching assistant or the course staff, which will be done as a last recourse.

In addition to this, if a team member consistently overestimates the number of tasks they are able to complete in a sprint, the other members of the team will restrict the amount of work they are able to take on.

Each member of the team is also responsible to inform the others if they will not be able to complete their assigned tasks before the deadline. We will generally leave it up to the team member in question if they want to have their task reassigned. However, if the task is particularly important or urgent, it will get reassigned to a different team member immediately, so that it can still be completed on time.

Success factors

We are passionate about using computer science to further scientific progress and strongly believe in open-source, ethical software development. We have varied experience and interests across many areas of computer science and beyond. In addition to being good friends, we have also successfully worked together before in both personal and university projects including video games, a neural network, and assisting professors teach CSE courses. We look forward to taking on a new challenge and using the software project as an opportunity to create something new.

Norms or evaluation criteria

We will evaluate the work of each team member according to these criteria:

1. Behaves professionally and respectfully at all times.
2. Maintains honest and frequent communication with the rest of the team.
3. Attends the agreed-upon meetings and participates in them actively.
4. Delivers good quality work, within the agreed-upon deadlines.
5. Takes initiative and is willing to help others.

Issues are considered done when:

1. They have 80% branch coverage.
2. The code has been reviewed by at least 2 members.
3. Its documented within our Wiki.
4. The code has comments describing it.
5. The task checklist is completed.

C

Feasibility and Risks

In this section, we discuss our prior expectation of feasibility of realizing the project given the time and scope. We then consider what technologies are available and how to mitigate development risks.

Time management

The vision described in chapter 3 is very complex and would take much longer than the available timeframe to realize in full, possibly taking years and a larger team. The agreed upon goal for this iteration of the project reflects this: we aim to have a proof of concept that shows off basic functionality and allows for future expansion. We have worked with the client to prioritize features in order to accommodate the limited timespan of this project iteration. This smaller scope makes it feasible to complete the project within the limited timeframe.

Technical

The scope can be summarized as the creation of a forum with specialized features, where the main challenges are rendering and embedding a Quarto project within a webpage, and allowing iteration on posts through a process similar to Git merges. We plan on tackling the former by using the Quarto CLI tool[1] to generate a file (either HTML, PDF, or GitHub Flavored Markdown) that can be embedded in a webpage. The latter can be achieved using Git server-side.

Other components of the system are common to most web apps, and technology is widely available to create them. Specifically, we will use the React framework[2] to build Alexandria's front end and the Gin framework[3] for its back end.

The aforementioned technology is well consolidated, coming from years of development and wide use in the industry. As such, we deem the likelihood of it setting us back to be low. Additionally, none of it has to be developed by the team. Therefore, the technological aspect of the project is entirely feasible.

This is the second iteration of building Alexandria within the Software Project, and the previous version of the platform[11] is available as open source software under the GNU General Public License v3.018, which the client has encouraged us to make use of. The requirement of following the new UI prototype limits how much of it can be reused, such that we do not plan on forking it directly. Nonetheless, this source code can serve as a base for features such as creating posts, comment threads, and comparing post versions.

Additionally, the chosen tech stack is the same as that iteration (described in detail in Chapter 6), and one of the team members has experience with it, allowing for easier learning for those unfamiliar with it. These considerations show the project is feasible.

[1] https://quarto.org/
[2] https://react.dev/
[3] https://gin-gonic.com/

D

Full list of Requirements

There are several systems needed to realize the main requirements outlined in section 3.2, and potential features that could improve the platforms usability and appeal. To define these the team devised a larger list of requirements, which are presented in this section.

The requirements were prioritised according to the MoSCow method [12]. This method involves splitting the features into four categories, in order of importance:

- **Must Haves**: these features represent the core of the application. Without these, the development iteration will not be considered a success;
- **Should Haves**: these features represent important functionality that the final prototype should have, but which are not essential to the success of the project;
- **Could Haves**: these are features that would be nice to have and would improve user experience, which the developers should focus on if they have time left;
- **Won't Haves**: these are features that the final application should have, but lie outside the scope of the current project.

Outlined are all requirements our team has identified.

Glossary

Below we describe 8 key terms that are present in the requirements:

1. A **user** is anyone using the Alexandria tool, logged in or not.
2. A **member** is a user that uses Alexandria while logged in.
3. An **admin** is a user that is meant to moderate Alexandria by removing harmful content and helping users with various issues using the platform.
4. A **post** represents a collection of Quarto project files, their rendered document and some metadata as specified in the requirements. It's **main version** is the one shown when the post is first opened.
5. A **project post** is a post that members can propose changes to.
6. A **branch** is a changed version of a post (including metadata) that may become the post's main version after going through the peer review process.
7. The **peer review process** happens when a branch is created. In order for the branch to become the post's main version, exactly three experts in the field of research of the post need to review the branch. An expert is determined based on the profile tags.
8. A **discussion** is a user-written comment related to a post, branch, or another discussion.
9. A (peer) **reviewer** of a post/branch is a member who has reviewed that post/branch.

Must have

Twenty five items have been identified as must have requirements, visible in tables D.1 and D.2.

Six items have been identified as non-functional must have requirements, visible in table D.3

Should have

A further twenty five items were categorized as should haves, visible in tables D.4 and D.5.

Four items have been identified as non-functional should have requirements, visible in table D.6

Could have

Twelve items are could have requirements, visible in table D.7.

The following 1 requirement has been identified as non-functional could have requirement:

1. All personal account data needs to be stored in a GDPR compliant manner (not implemented).

Won't have

These items were decided to belong to won't have requirements:

1. A user can go through a tutorial to better understand how Alexandria works.
2. A user can highlight parts of their post to encourage discussion/collaboration on a specific topic.
3. There is a reward system based on tokens.
4. There is an achievement system that members can view and interact with.
5. A member can directly message another member.
6. A member can invite another member to review their project post.
7. The homepage includes:
    - "active" discussions as a sidebar
    - posts tailored to the user's behavior
8. Posts can be created as private, allowing their creator to freely modify them until they're set to public.

The following 4 items have been identified as non-functional wont have requirements:

1. User sessions are securely implemented on the front end. Sessions cookies are encrypted.
2. Quarto projects and renders are run in isolation, inside a Docker container.
3. Input sanitization for any user input.
4. Users are notified about the session cookies necessary for logging functionality.

Table D.1: Must Have Functional Requirements Part 1

Number	Requirement	Completion Status
1	A member can create a new post by specifying a title, authors and collaborators, uploading their content, and then publishing it;	Fully implemented
2	Authors/collaborators must be existing members;	Fully implemented
3	A user will, by default, view a post as a rendered version of its Quarto project files, along with the following metadata: • title, • authors, • collaborators, • reviewers, • review status, • creation date, • last edit date, • a list of peer-reviewed, open for review, and rejected branches;	Fully implemented
4	A user can choose to view a rendered post as the Quarto project files that underlie it;	Fully implemented
5	A user can explore the Quarto file tree structure, and select files to view their raw contents;	Fully implemented
6	A post has one of the following review statuses: • open for review (has not been reviewed), • revision needed (was rejected by peer review), • reviewed;	Fully implemented
7	If a post that is ”open for review” or has ”revision needed” has changes proposed to it and those changes go through a peer review process and are accepted, it becomes ”peer reviewed”;	Fully implemented
8	A branch has one of the following review statuses: • open for review, • reviewed, • rejected;	Fully implemented
9	A user can select one of the branches listed on the post page to view the branch’s page;	Fully implemented
10	A member can create a new branch for an existing post; they must provide the modified version of the post files, branch title, and contributors;	Fully implemented
11	When creating a new branch, a member may also propose a change for the original title of the post;	Fully implemented

Table D.2: Must Have Functional Requirements Part 2

Number	Requirement	Completion Status
12	On the branch page, a user can view: • contributors (branch authors), • creation date, • last update date, • reviews, • approval status, • the proposed new version of the post (as its navigable file directory tree or rendered document), • the version of the post before modifications (as its navigable file directory tree or rendered document);	Fully implemented
13	At the time of approval or rejection of a branch, the main version of the post will be stored as the ”previous version”;	Fully implemented
14	A member can peer review a branch by giving written feedback and approving or rejecting the branch;	Fully implemented
15	Up to three members can review one branch, and a member cannot review a branch they are a contributor of or that they have reviewed already;	Fully implemented
16	If a branch is accepted (by exactly three members), the Quarto project files and metadata associated with it will replace those of the post; the authors of the branch will be added as contributors to the original post, and the branch and post will be marked as ”peer reviewed”;	Fully implemented
17	If a branch is rejected by any reviewer, the post is not updated, and the branch is marked as ”rejected”;	Fully implemented
18	When creating a post or branch, a user may do so anonymously, making the list of authors empty; otherwise, their name must be on the list of authors;	Fully implemented
19	Individual peer reviews on branches must be displayed anonymously;	Fully implemented
20	A member can download the Quarto project underlying any post or branch;	Fully implemented
21	A user can browse a list of the most recent posts on the homepage and select any post to view it;	Fully implemented
22	A user can create an account by specifying: email, password, first name, last name, and institution;	Fully implemented
23	A user can log in and out of their account.	Fully implemented

Table D.3: Must Have Non Functional Requirements

Number	Requirement	Completion Status
1	Documents must be rendered using the Quarto tool;	Fully implemented
2	The back-end must use Git for branches and file storage;	Fully implemented
3	Passwords need to meet the following requirements: • at least 8 characters, • use lowercase letters, • use uppercase letters, • use numbers, • use special characters;	Fully implemented
4	The front end will be implemented using TypeScript with React.js and Next.js;	Fully implemented
5	The back-end logic will be implemented using Go with Gin;	Fully implemented
6	The database will be MariaDB;	Fully implemented

Table D.4: Should Have Functional Requirements

Number	Requirement	Completion Status
1	There is a predetermined list of scientific fields that users can pick from;	Not implemented
2	A member can choose scientific fields to add to their profile when creating an account;	Fully implemented
3	When creating a post, a member can additionally pick the following metadata: • scientific fields; • post type (Project/Question/Reflection);	Fully implemented
4	Additionally, if a member is creating a project post they can specify: • completion status (Idea/Ongoing/Completed); • feedback preferences (Community Discussion/Formal Feedback);	Fully implemented
5	A user can view the following on the post page: • scientific fields; • post type; • reply count; • view count; • completion status (if the post is a project post); • feedback preferences (if the post is a project post);	Mostly implemented, missing view count
6	Branches can only be created for project posts;	Fully implemented
7	When creating a new post or branch, a user can see a preview of the rendered version of the uploaded files;	Not implemented
8	New branches, by default, inherit the scientific fields, completion status and feedback preferences of the original post. The user creating the branch may choose to add or remove fields, or change the completion status and feedback preferences;	Fully implemented
9	A user can view the scientific fields, completion status and feedback preferences of a branch on its respective page;	Fully implemented
10	When a branch is approved, its scientific fields override those of the original post;	Fully implemented
11	A member can peer review an open branch only if they have at least one of the fields of expertise which the branch has;	Fully implemented
12	A member can start a discussion on a post or branch and can choose to do so anonymously;	Fully implemented
13	A member can reply to a discussion and can choose to do so anonymously;	Fully implemented

Table D.5: Should Have Functional Requirements

Number	Requirement	Completion Status
14	A member can view a discussion beneath a post or branch, with information about the author, time of posting, replies, and, if applicable, the branches that the discussion is referencing;	Partly implemented, missing link to referenced branch
15	When a branch is approved, the discussions beneath it get added to the discussion list of the main post;	Not implemented
16	A member can fork a post. This is similar to creating a new post, except the new post will contain a clear reference to the post it was forked from to indicate that it builds off of someone elses research;	Not implemented
17	A user can choose to import their files from GitHub when creating a new post;	Not implemented
18	A user can enter Quarto-style markdown directly into a text box when creating a new post, instead of uploading files;	Not implemented
19	A user can search for posts on the homepage according to search terms and by filtering according to tags. These search terms are matched with authors, titles, and topics of the article;	Not implemented
20	A user can see the view count, reply count, and some tags of a post on the homepage before opening a post;	Partially implemented, missing view count
21	A user can add links to their Google Scholar, ORCID, GitHub, LinkedIn, and OpenScience accounts to their profile;	Not implemented
22	A user can view an account page for any user to see their linked accounts, fields of expertise, list of posts, and list of replies;	Not implemented
23	A member can change their account data, namely: 1. name, 2. institution, 3. scientific fields, 4. account links, 5. password;	Not implemented
24	A member can delete their account; Not implemented	
25	A user can view an ”about” page which gives credit to all contributors to Alexandria and provides an overview of the project;	Fully implemented

Table D.6: Should Have Non Functional Requirements

Number	Requirement	Completion Status
1	The scientific fields stored in the database must comply with popular scientific field classifications;	Not implemented
2	Passwords need to be hashed using bcrypt mechanism when stored;	Fully implemented
3	There is user authentication done using JWT tokens;	Fully implemented
4	JWT tokens are used for managing user sessions, logging in and out;	Fully implemented

Table D.7: Could Have Functional Requirements

Number	Requirement	Completion Status
1	A user can view a visualization of the history of changes made to a post as a timeline, with the author, date, and specific changes made in each branch;	Not implemented
2	Posts have their review status highlighted on the homepage and when viewing the post, to be transparent about reliability;	Not implemented
3	A user can sort posts on their homepage or which they search for by: • popularity (view count), • recent (date), • controversial (discussion rate);	Not implemented
4	The search functionality of the homepage also matches search terms with the contents of articles;	Not implemented
5	A member can save any post to their list of saved posts;	Not implemented
6	A member can view their list of saved posts, and select a post to view it;	Not implemented
7	A member can see a calendar of their activity in Alexandria;	Not implemented
8	A user can create an account using a: • GitHub account, • ORCID account, • institution account;	Not implemented
9	A member can report a post or a reply with a specified reason;	Not implemented
10	A member may add names to the authors/collaborator list that are not linked to member accounts;	Not implemented
11	An admin can delete reported posts and discussions;	Not implemented
12	An admin can review reports of posts and replies, and delete them if the reason for reporting is not deemed valid;	Not implemented

E

Model Diagram

A relational database system was used for storing information, as per subsection 5.1.3. The image below illustrates all database model relationships:

Figure E.1: Relational diagram of the back end database models generated by JetBrains DataGrip.