Empowering Development Teams: A web-based solution for evaluating code complexity and enhancing contributor performance

Anthony Charles Clermont

May 2024
Bsc. Computer Science

Supervisor - Stephen Riddle

Word Count: 14608

Abstract
This dissertation addresses the evolving landscape of code analysis in software engineering, spurred by the seminal work of Fagen (2011) and propelled by advancements in technology. It delves into the gap between existing code quality evaluation tools and contributor performance assessment within software development teams. Drawing from firsthand experience and leveraging features from already successful tools like SonarCloud, the project aimed to engineer a web-based solution, adhering to fundamental engineering principles, ensuring best practices and a robust and systematic approach to software engineering.

The project successfully delivered a robust application with a strong emphasis on security through its authentication system. The well-designed user interface enhances user experience and accessibility, and the project effectively begins to tackle the challenge of analysing the limitations and performance of contributors. The metrics utilised lay a solid foundation for assessing performance. However, it acknowledges the potential for improvement, particularly in achieving more intelligent insights through the integration of machine learning. Discussions on the potential applications of machine learning in this context are initiated, paving the way for future enhancements. Potential implementations for further development have also been explored.

Acknowledgements
I would first like to thank my supervisor Stephen Riddle. He has shown keen interest in the project throughout and has helped guide this project. This guidance has aided the success of the project. I would also like to thank my placement company; and specifically Chris Reynolds and Max Rose. Without whom the project's concept would not have materialised and I would lack the technical skills required.

Declaration
“I declare that this dissertation is my work except where otherwise stated.”

Table of Contents

Table of Figures

1. Introduction

This chapter introduces the project, outlining its aim and objectives, and providing the reason for its necessity, along with the general structure of this dissertation.

1.1 Purpose

Code analysis stands as a fundamental principle of software engineering. Design and Code Inspections to Reduce Errors in Program Development (Fagen, 2011), proved to be the catalyst for code reviews being an imperative step in the software engineering cycle. Over time, code analysis has progressed; with the assistance of advanced technology, many tools aid developers in producing higher-quality code and more robust solutions.

Drawing from my experience within a software development team, I've gained first-hand familiarity with SonarCloud: a widely used analysis tool. SonarCloud performs code quality analysis on linked repositories by calculating several key code metrics such as the number of lines of code, code complexity, test coverage and bug identification *(Puspaningrum et al., 2023)*. Its primary focus is to assist developers in writing cleaner code, thereby improving the overall performance of contributors within a team.

While tools like SonarCloud have become indispensable in evaluating key code metrics used during peer code reviews, a notable gap persists in higher-level overview metrics, particularly in performance assessment. Project leads are often required to possess a profound understanding of the codebase as well as the capabilities of the developers working on the project. Without practical overview metrics, the assignment becomes increasingly challenging.

This project intends to engineer a web-based solution which provides high-level overviews of contributor performance. A potential use case would be highlighting which contributors perform the best on given aspects of the codebase. One contributor might write higher quality code for the backend but perhaps isn't as strong at writing front-end code. This type of analysis would be crucial during sprint planning, helping to identify which contributor would be most applicable to given tasks, and what is and isn't realistically achievable.

1.2 Aim and Objectives

To evaluate how existing tools compute code metrics, explore possible solutions for determining code quality and develop a web-based code analysis tool, utilising static code analysis to determine contributor's code quality and other performance-based metrics.

1. Investigate and analyse potential solutions to determining code quality based on published research
2. Research and identify the most suitable technology stack for development and implementation of this web tool
3. Adhere to agile methodologies specifically Scrum, for an optimal software engineering process

4. To become familiar with GitHub's API, including the authentication process and API calls for relevant project data
5. Ensure industry standard security principles are followed, including hashing/encryption of sensitive information, an appropriate authentication flow and correct authorisation controls to restrict unauthorised access
6. To utilise the findings from objective 1 and implement a suitable solution which is capable of determining contributor's code quality
7. To achieve a minimum of 80% test coverage across the application to comply with correct software engineering principles

1.3 Dissertation Outline

This section will introduce the remaining chapters, summarising the overall purpose and pick out some key outcome aspects of each.

Chapter 2

This chapter begins to discuss some of the project's key technological decisions, including the technology stack, web security and static code analysis. Identifying a good technology stack for this project to use (Python, FastAPI, Typescript and Angular), how security and authentication will be token-based utilising JSON web tokens and concluding on three static analysis code metrics which will be used to determine code quality.

Chapter 3

This chapter introduces the reader to the structure and management of the project's engineering cycle, the system architecture and UI, UX and branding design decisions. Although no development is specifically discussed, the chapter should provide aid during the remaining chapters due to a better understanding of the system.

Chapter 4

This chapter delves into the major development milestones. It covers important development steps, providing a high-level overview and diving into some specific code explanations where applicable. It will also explain challenges which arose during development and how these were overcome. Most notably, performance improvements with features such as caching and re-writing SQL queries to be more optimal, supporting multiple languages through tokenized code and handling asynchronous requests to name a few.

Chapter 5

This chapter discusses the project's implementation of testing, both manual and programmatically. It will explain the overall approach and evaluate potential improvements which could be made including end-to-end testing and specific end-user testing.

Chapter 6
This chapter will first review the final system, exploring a walkthrough of the application from user registration to repository analysis, visualising the project's outcome as an end user. It will then evaluate the outcome the project's aim and objectives, and discuss potential improvements to better meet some of the objectives.

Chapter 7
This final chapter concludes the project. Firstly, identifying and discussing personal development, before outlining future work which could be undertaken to improve the application, including deployment of the application, social and collaboration features; enabling repository owners and contributors to utilise the application simultaneously and how machine learning could improve the code analysis.

2. Technical Background
This chapter explains the research performed for this dissertation and discusses the key technological decisions.

2.1 Technology Stack
In the initial stages of this project, a key objective was to decide on a technology stack. There were three major aspects to consider: database system, back-end and front-end. In the modern age, there are an unlimited number of technology stacks, all with their advantages and disadvantages.

"Simplifying Web Application Development Using-Mean Stack Technologies" (Dunka et al., 2018), evaluates the advantages of modern technology stacks such as MEAN (MongoDB, ExpressJs, Angular and NodeJs), highlighting the speed and flexibility they provide.

Each decision made for this project involved an extensive amount of research. This research began with the database system. This previously mentioned paper suggests the use of a NoSQL database system (MongoDb). NoSQL databases have surged in popularity (Li & Manoharan, 2013), this paper, "A performance comparison of SQL and NoSQL databases", compares the performance of traditional SQL and NoSQL database systems. The findings of their work showed little correlation between performance and choice of database system, showing that the performance depends entirely on the structure and need of the database rather than the technology.

Due to these findings, I have decided to use a SQL database system as the advantages provided by NoSQL systems: large volumes of data, complex data relations and semi-structured data don't apply to this project. To briefly note, this project would ideally use a system such as PostgreSQL, however, due to the project time constraints I have decided to use a lightweight solution: SQLite3.

The choice of back-end language and API framework relies less on published research and more on my own experience. Although the original paper suggests ExpressJs, the lack of technical knowledge of this framework arose as an issue. Thus instead I decided on Python as my back-end programming language. Python has a plethora of API frameworks which were considered, notably: Flask, Django and FastApi. Due to the limited functionality Flask provides, and the single synchronous architecture of the framework, it is not an ideal choice for this project. This leaves Django versus FastApi.

The article “FastAPI vs. Django Rest Framework: A Comprehensive Comparison” (Singh, 2023), discusses the advantages and disadvantages of both but concludes that the best choice entirely depends on the system requirements. Considering the project's needs, FastApi serves as the best choice, most notably because the performance and asynchronous architecture prove most advantageous.

The most straightforward stack choice was the front-end framework. As suggested by “Simplifying Web Application Development Using-Mean Stack Technologies” (Dunka et al., 2018) I have extensive knowledge of the Angular ecosystem and framework, the encapsulation provided and intuitive structure allows for cleaner code development over other popular front-end frameworks such as React. Typescript further emphasises these Angular principles allowing for strict type checking and a more coherent project structure.

To summarise, this project will use a range of technologies which can be visualised in *[Figure 1]*. From major frameworks and languages as discussed (angular, typescript, python and FastApi). It will also utilise technologies for testing: Karma and Pytest. Gunicorn for concurrent connections enabling FastApi to run asynchronously. RxJS allows the front-end to handle asynchronous responses and update information in real-time and Pydantic is a data-validation package enabling Python to have stricter type checking and overall security improvements. This project is complex, with a range of technologies all working together to create a system capable of determining code quality.

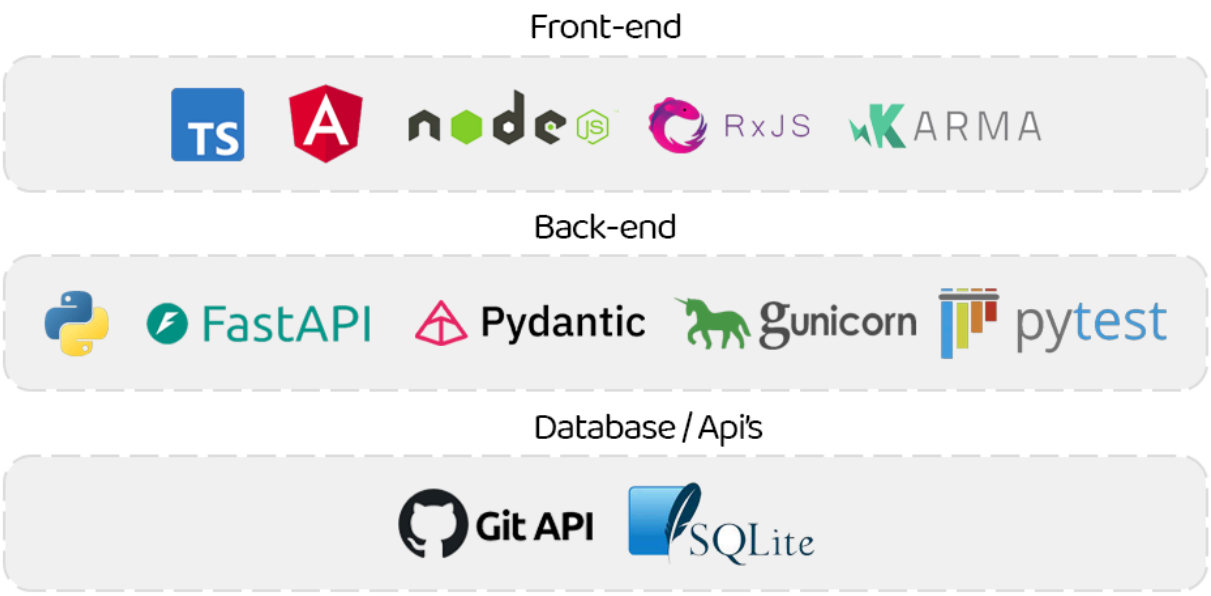


Figure 1 - Application's technology stack

2.2 Web Security

Due to this project needing access to users' GitHub accounts and repositories the need for correct and industry standard security is paramount. "Security in web 2.0 application development" (Noureddine et al., 2008), highlights key security vulnerabilities and processes to explain industry best practices and precautions to mitigate and/or prevent these vulnerabilities.

This paper emphasises the importance of considering vulnerabilities for both server and client-side attacks, as both hold personal and business information. The project's most applicable vulnerability is Cross-Site Request Forgery (CSRF), an exploit which relies on vulnerabilities in both the server and client. Tricking the server by using the authenticated user's credentials to submit a request to the server. A potential attack could cause catastrophic harm, from altering user passwords to making online purchases (Noureddine et al., 2008).

Although for this project, hackers making purchases is not applicable, changing a user's GitHub password, or email address is a definitive possibility. A more recent paper, "An authentication based scheme for applications using JSON web token" (Ahmed & Mahmood, 2019), begins to resolve this vulnerability using JWT's (JSON Web Token). The tokens can be sent from the server to the client as a same-site cookie, preventing the browser from sending these cookies across the origin. Furthermore, the API can contain a CORS (Cross-origin resource sharing) configuration restricting the origin which requests can be made from.

The documentation for FastApi contains detailed documentation on how to configure a CORS policy and provides secure route protection. Utilising these features should allow for authenticated routes to be correctly protected and prevent CSRF attacks with a well-configured CORS policy and sending the JWTs as same-site cookies.

(Ahmed & Mahmood, 2019) also demonstrates how to safely use JWTs and potential implementation. JWTs usually contain three main elements: the header, payload and signature *[Figure 2]*. The header typically contains the hash algorithm used when creating the JWT signature. The payload contains data, most commonly the user's ID or email along with an expiry time. Finally, the signature is digitally signed using a secret key known only to the server, this way the token received with a request can be verified by the server. This token is then given to the client and stored as a cookie, when the user requests the server this token is attached to the header of the request, and the server can verify the user's authenticity.

Source - https://supertokens.com/blog/what-is-jwt

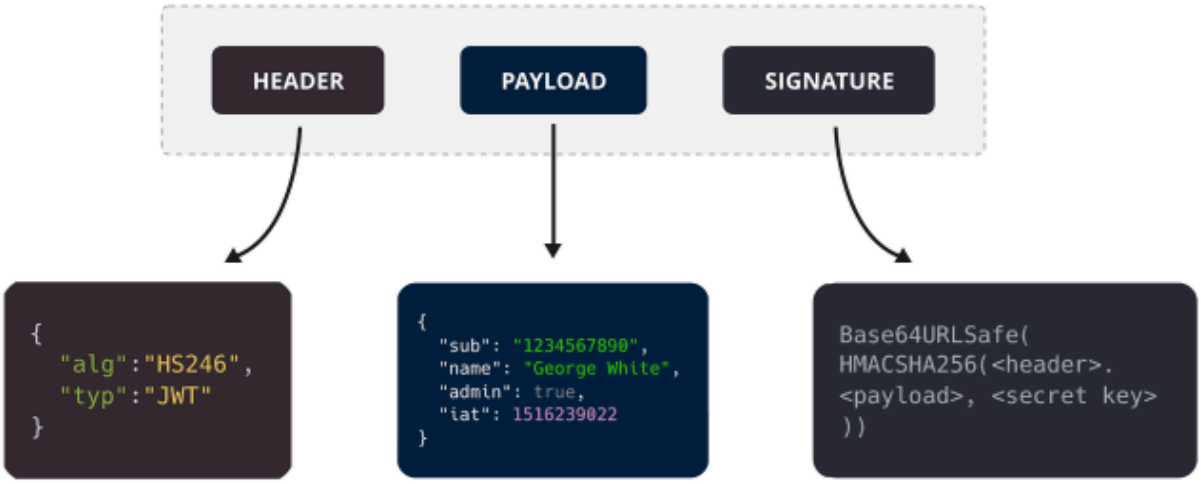


Figure 2 - Typical JSON web token structure

So far this research has focused on authentication, but what happens if a cyber attack occurred that granted a hacker access to the project's database? The database will store the user's GitHub access token. With these tokens, a hacker could easily cause catastrophic harm. "Database Encryption", analyses and evaluates a mitigation for this attack. Encrypting database data can provide strong security for data at rest (Bouganim & Guo, 2011). This paper proceeds to outline the key factors which must be considered when encrypting data. Most notably, which application layer the encryption is performed and how to store the encryption key. Firstly, this paper outlines the three choices for the execution of the encryption/decryption

1. Storage-level encryption
2. Database-level encryption
3. Application-level encryption

With almost every technological choice, there is no right and wrong, the correct choice depends on the system requirements. Storage-level encryption is well suited for encrypting files or entire directories and, thus not required for this project. Database-level encryption would be a good choice, as it encrypts/decrypts the data as and when it is inserted and retrieved from the database, this requires the database system to be able to perform data encryption.

Unfortunately, SQLite3 doesn't support this and due to the project's constraints, this choice of system cannot be reconsidered. This leaves application-level encryption, which allows for the unique advantage of being able to separate the encryption key and the encrypted data.

How to store the encryption key? The database protection solution is only as good as the protection of the keys (Bouganim & Guo, 2011). If the keys are accessible by the attacker, the protection from encryption is nullified. For this reason, the project will utilise a .env file, containing all secret keys which are needed for the project to run. This .env file will be omitted from the GitHub repository, therefore only being shared with the necessary personnel.

To summarise my findings, web security is a major challenge for any software engineering project, and due to advances in web technologies, cyber attacks continue to increase. This project cannot possibly address every known security issue, but what the project can do is observe which are the most applicable, and implement measures to mitigate and prevent such vulnerabilities. This research has guided the project to utilise Json Web Tokens as the authentication method. Even a rudimentary implimentation of these tokens provide better security to prevent CSRF attacks. Along with same-site cookies, a well-defined CORS policy, route protection, database encryption and securely storing private keys, this project can ensure industry standard security principles are followed, fulfilling a key project objective.

2.3 Static Code Analysis

The largest challenge this project faces is how to perform and produce metrics using static code analysis. Researching this topic has produced two viable options.

1. Machine Learning
2. Formula calculations

"Measuring software development productivity: A machine learning approach" (Helie et al., 2018), discusses a potential approach to determining development productivity, through the use of machine learning. The main challenge of this approach is being able to collect a relevant dataset which is large enough to accurately train a model. This paper utilises an existing static code analysis tool to evaluate code from a dataset they found, then this clean data can then be used to train the machine learning model. For this project, this approach has one major issue, Time. Collecting, cleaning and preparing the training data, training the model, evaluating the accuracy of the model, re-training and improving the predictions and deploying the model to be running locally in a docker container or the cloud, all this combined could put the success of this project into jeopardy.

Alternatively, the second approach would be to utilise formula calculations, each calculating a given metric and using these calculations to evaluate the overall 'grade' of code. "Measuring Software Maintainability" (Kukreja, 2015) evaluates key metrics which can be used to evaluate code. It focuses on maintainability, using the maintainability index formula; comprising four other metric calculations: Halstead Volume, Cyclomatic Complexity, Total SLOC (source lines of

code) and Comments Ratio. Which metrics are most suitable, is not a simple answer, and the maintainability index alone is debatable.
One solution to this issue could be to understand which metrics are considered best by actual developers., “Developers talking about code quality” *(Börstler et al., 2023)* is a paper which does exactly this. Conducting interviews with 34 professional software developers, programming teachers and students, each participant presented what they believed were good/bad code examples. From this, the researchers concluded readability, structure and documentation were the most common identifiers for quality code. As mentioned prior cyclomatic complexity. Which is a formula able to determine both readability and structure, along with the maintainability index calculation should show the relative ease of maintaining the code analysed code. As for documentation, fully automating this analysis is not possible. Most commonly, documentation is separate from the code repository. However, what can be automated is the comments-to-line ratio. A calculation which determines how well commented code is.

To compare, machine learning seems to be the accepted approach for similar existing technologies (Snyk for example), the automation and detailed insights this approach provides cannot be matched using calculations. However, the scope required for a successful implementation is too large. Creating a machine learning model could easily be an entire dissertation on its own, and although calculating code quality is part of the aim, this project aims to focus on the software engineering cycle and full-stack web development. Therefore, instead this project will utilise the findings from the calculations article and the developer interviews paper to implement a solution capable of calculating metrics such as maintainability index, cyclomatic complexity and comments-to-line ratio. These metrics combined should accurately determine code quality and maintainability, meeting this project's main aim.

3. Software Engineering Process

This chapter will introduce the reader to the structure and timeline of the project’s engineering cycle, the system architecture and UI, UX and branding design decisions.

3.1 Planning

Due to the project’s significant scope, a well thought out and structured development plan was essential for a successful outcome. This plan was first constructed as a Gantt chart *[Figure 3]*, providing a higher overview of the project’s key progression steps. The chart utilises an agile development approach, with a focus on sprint development.

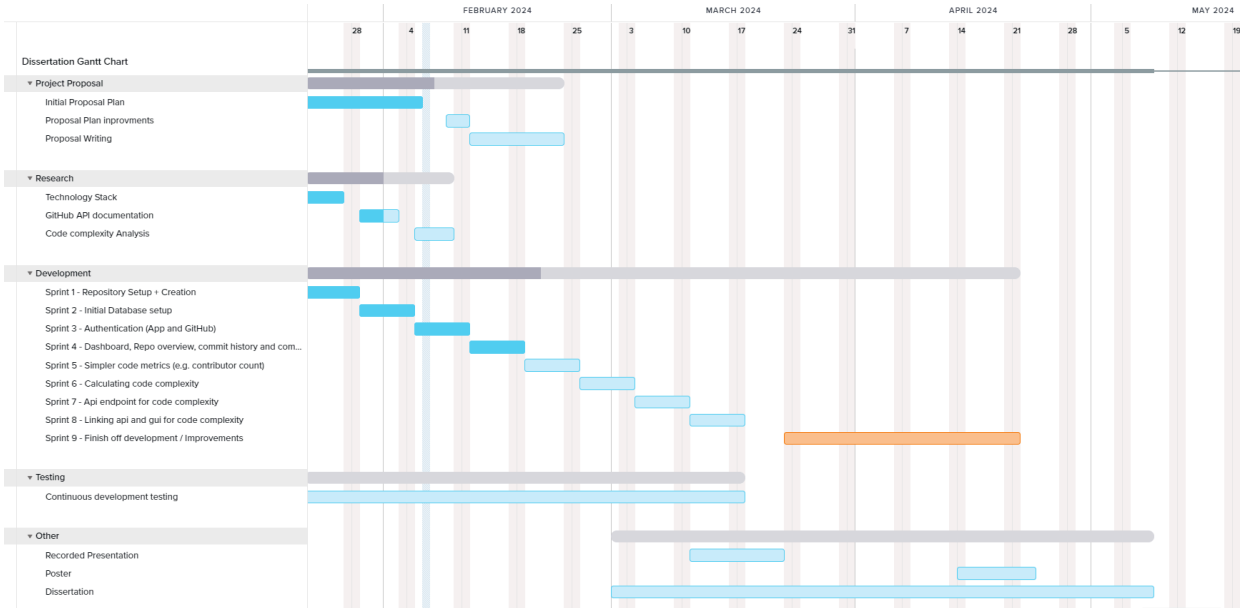


Figure 3 - Project's Gantt chart

Agile allows for better time, quality and productivity and enhanced flexibility to incorporate evolving requirements (Kaur & Jajoo, 2015). Moreover, my prior experience working within an industry agile development team will support a familiar and thus productive engineering process.

A key management decision was to incorporate Kanban methodology. Before the research stage, this project intended to use scrum methodology. Scrum attempts to build the work in short iterations where each iteration consists of short time boxes (Sachdeva, 2016), and is suited best for small to medium-sized development teams. As previously mentioned this initial plan was influenced by prior experience. Although, through research, kanban emerged as the most suited for this project.

Kanban is a lightweight agile methodology, consisting of key principles. Work is visualised and the workflow it follows, limits work in progress, manages flow and makes management policies explicit (Kirovska & Koceski, 2015). Adopting these principles a Kanban board can be created to comply with this chosen approach. This is where Trello, a software management tool has been incorporated *[figure 4],* the task's key development stages are layout out, moving chronologically across the x-axis, aligning with the first key principle. The second principle is respected through the use of sprint planning. Each key chunk of development has been initially separated into weekly sprints, this approach helps to manage workload and ensure focus can be applied to the necessary tasks.

From the Gantt chart *[figure 3]*, the overall development aim has been deconstructed into nine, one-week sprints where a key objective has been defined for each. *[figure 4]*, visualases the combination of Kanban methodologies and sprint development approach.

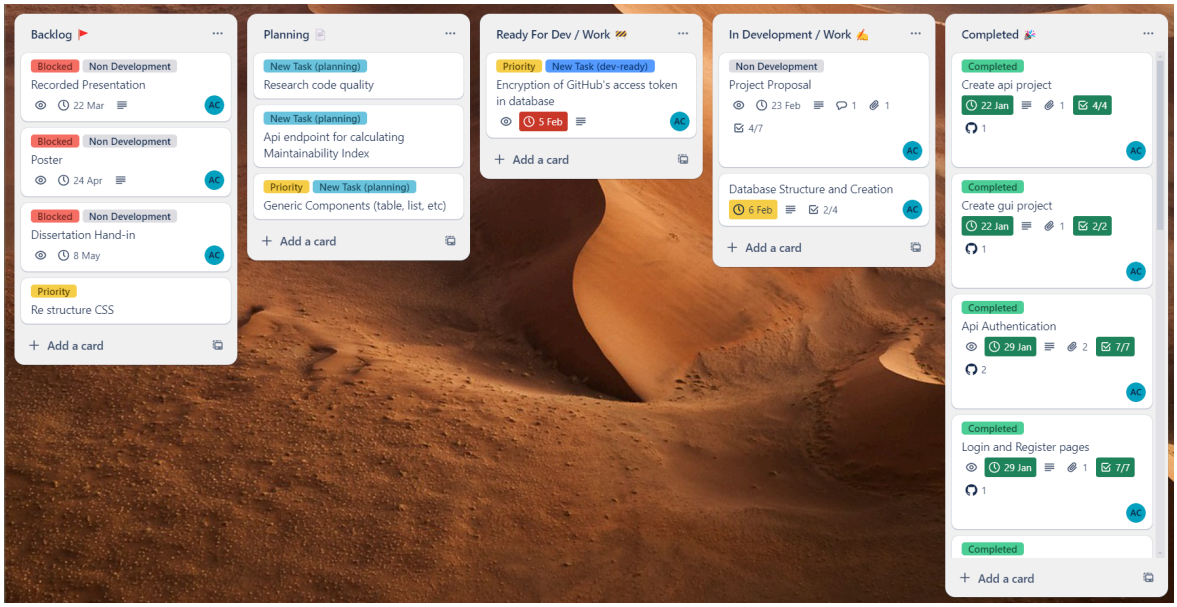


Figure 4 - Project's Trello management board

3.2 System Design

This section intends to provide an overview of the solutions' architecture and system design without going into specific detail. Hopefully, to provide a better understanding of the technical content discussed later in this paper. Worth noting this section relies upon *[figure 5]*, so refer to this figure for a visual representation.

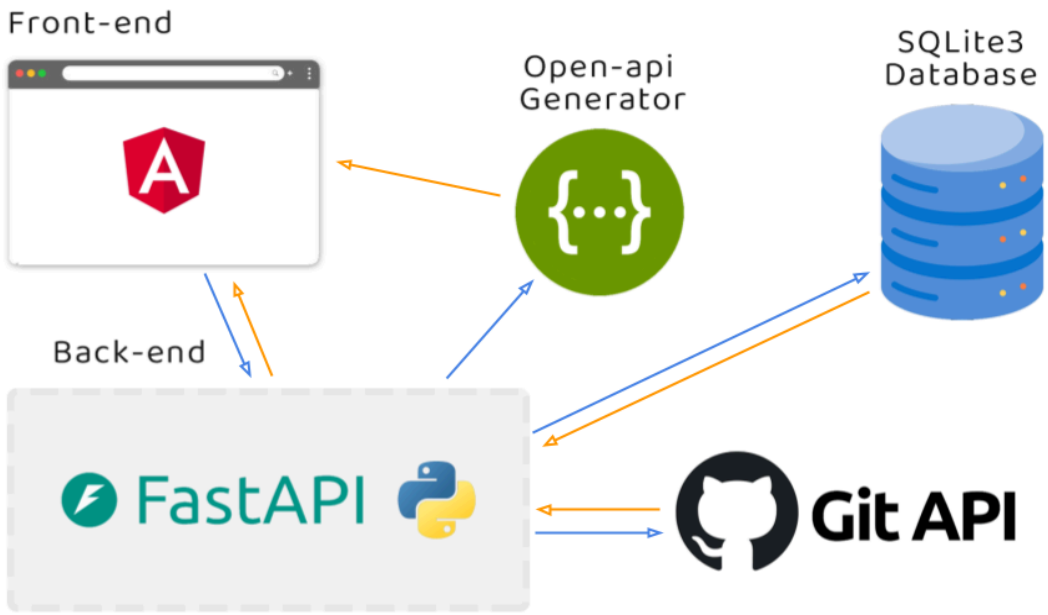


Figure 5 - Application communication between system layers

Beginning with the centre of the systems process', the back-end and API. The system revolves around the back end. Sending requests and processing the response from Github's API, storing and querying the database, handling the system's authentication functionality, general processing of data and the functionality behind the front-end.

The front end holds very little information and relies on the back-end systems to operate. Communicating with any of the other areas of the system through the back-end, this way ensures better security and more control over the system. Moreover, any sensitive or private keys can be kept safely on the server (back end), and away from the client (front end).

Lastly, the API's endpoints are defined with a return type. This type is defined as a Pydantic model, increasing security as it ensures what data is expected to be returned from the request. Utilising an npm package (openapi-generator-cli generate Fast-API), allows the front-end system to request the models used in the API and fetch them into the front-end code repository. This approach allows for the data types to be pulled into the front end, ensuring the different subsystems remain consistent and updated, supporting stricter type-checking throughout the application.

3.3 UI, UX Design and Branding

To encourage the sense of a proper software engineering project a brand was created. This brand should sustain a focused and consistent approach.

Firstly, to discuss colour. The 60-30-10 rule is an internal design rule that is always used by the designers intuitively and deliberately. with this principle, the images create balance and allow the eyes easily to move from one point to another. (Barševska & Rakele, 2019). The concept is to have a dominant colour covering 60% of the space, 30% as a secondary colour and finally a 10% accent colour. *[figure 6]* shows the colour palette compared with the mock-up, although the palette consists of some slight shade alternatives, we can see how white has been used as the predominant colour, dark purple as the secondary and the brand purple as the accent.

Typography is a key aspect of design and branding. Also seen in *[figure 6]*, the choice of font and readability were considered during design, this figure also shows how the chosen palette affects the readability of the text. Moreover, spacing and the general layout of the application have been carefully considered and constructed minimally to promote the salient information.

Lastly, to tie together the design and branding, the Angular framework allows for easier styling inheritance and shared general styling throughout the application, this global styling achieves a more consistent interface and easier development. The styling for a HTML tag or class can be defined once in the root application styling and then inherited throughout the web application.

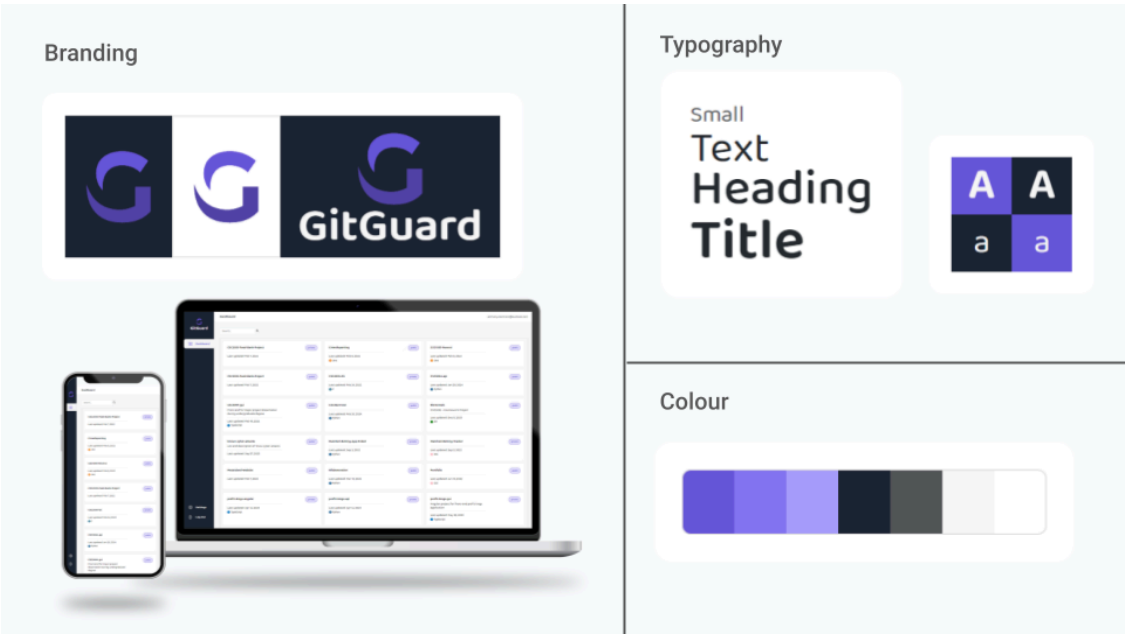


Figure 6 - Branding and application UI theme graphic

3.4 Ethical Considerations

Due to the project's design, there weren't many major ethical considerations. Most notably however, was the collection of identifiable personal data. This was considered because identifiable data, such as repository contributor data is collected within the tool, email addresses', forenames, profile images for instance.

However, it is important to note this data would be provided knowingly and consensually by users when connecting their GitHub account, as they are informed of the data the tool will have access to. In order to eliminate any potential ethical issue, the project's development will solely use only my own personal data.

4. Development

This chapter will dive into the development progression, targeting key milestones during the process, and explained in a chronological manner.

4.1 Project Initialisation and Version Control

The first step towards development was the setup and initialisation of the project's repositories. For both the back and front end, JetBrains was chosen for the IDE (integrated development environment), with PyCharm and Webstorm respectively.

Starting with the back-end systems. A GitHub code repository was created, the virtual environment was initialised and an interpreter was set up to use Python version 3.10. The first files created were a requirements.txt file along with a .gitignore file to ensure necessary files were omitted from GitHub, for instance, the .env file.

Requirments.txt is a special file within a Python project which the interpreter uses to understand the project's packages; what is and isn't installed or up-to-date. Throughout development, any packages added to the project were included in this file. This approach ensures all development devices are aligned and run using an identical virtual environment.

The same is automatically configured for the front-end, due to the Angular framework which utilises node.js as a package manager. This means any Angular project will contain a package.json which performs the same functionality as the previously mentioned requirement.txt file.

From here the back-end project installed Fast-Api and a basic hello world example was implemented to test the communication between the front and back end. I served both applications locally and had the front end make the basic root call to the API during the root components initialisation. Unfortunately, the request responded with a “Cross-Origin Request Blocked” error, which was resolved by setting up the Api’s CORS policy. During development, the policy has been basically defined, as it allows requests from all origins, all methods and all headers. This approach is still perfectly acceptable during local development, as on a private network there are no threats of attacks, however, if the application was deployed this CORS policy would need to be updated to restrict the allowed origins to only from where the front-end is hosted.

Lastly, the OAuth application needed to be registered on GitHub. This is then how the application is granted access tokens from users. The app was originally registered giving basic information and the private API key was stored in the API’s .env file. Later in development, the Oauth application was altered to include further details, which will be discussed later.

4.2 Authentication

The first major element of development was authentication. Security being a key project objective, the importance of correct industry standard security due to sensitive GitHub repository access was imperative, thus the implementation was well considered and thoroughly tested.

The first challenge the project faced was during the implementation of an authentication system, how to implement a globally accessible and robust authentication system, able to hash and verify user passwords and how to encode and decode JSON Web Tokens. To achieve this, the application makes use of a wrapper class, which can be utilised globally throughout the API. This class ‘AuthHandler’, contains two methods for password hashing and two methods for JWT handling. Password hashing is handled using ‘CryptContext’, a Python package which allows the project to define a schema and then has one method to create the hashed password and the other which can verify the provided password during login with the stored hashed password.

The JWT encoding and decoding were more challenging to implement. Firstly, to discuss encoding the tokens. Unlike password hashing, the structure of a JWT can vary greatly, therefore the challenge was deciding on what data it should store and how long-lived it would be. The project's JWTs contain three crucial pieces of information, the issued at and expiry timestamps and most importantly the user's ID. The expiry time is defaulted to twenty minutes, however, the structure of the function allows this default parameter to be overwritten, (used for encoding refresh tokens which was implemented later on). The most important piece of data in the token is the user's ID, this allows the API endpoints to know which user made the request, which would be required for querying this user's GitHub access token for instance. Once constructed, the token is signed using a hashed secret. This secret is a 32-bit string stored in the .env file, this way the decode function is also able to validate the authenticity and know the token hasn't been tampered with.

The decode function is simpler, the function attempts to decode the provided token, where if expired an 'ExpiredSignatureError' is caught, then a HttpException error is returned from the API, with the message of an expired token. The other issue could be an invalid token, this could mean it was altered, in this scenario an "InvalidTokenError" is returned from the API. A successful decode of the token means it's valid and hasn't expired, this will then return the user's ID to the endpoint which can then be used to gather the requested information. The wrapper function utilises this decode function and was later incorporated into API endpoints to validate request credentials *[figure 7]*.

```
/commit/changes", response_model=CommitDetails)
hanges(sha: str, repoOwner: str, repoName: str, user_id=Depends(auth_handler.authWrapper)):
```

Figure 7 - Route protection using the auth_handler

4.3 Authentication Endpoints and Auth Service

Following the authentication system was the implementation of the API's authentication endpoints and the front-end auth service.

After creating a new api file, 'auth_routes' I registered this file as a new router and linked it to the main application. The project's API consists of a large collection of endpoints, all of which follow a similar structure, hence I won't repeat the explanation. The first endpoint was the register endpoint, the first line in *[figure 8]* is the decorator, defining it as part of this file's router, the request method, in this case, a POST request, the actual route and the response model.

```python
@auth_router.post("/register", response_model=Token)
async def register(user_details: UserRegistration):
    user_details.password = auth_handler.getPasswordHash(user_details.password)
    user_id = database.register(user_details)

    access_token = auth_handler.encodeToken(user_id)
    refresh_token = auth_handler.encodeToken(user_id, 10800)
    return Token(accessToken=access_token, refreshToken=refresh_token)
```

Figure 8 - Register endpoint

The response model utilises the Pydantic Python package, serialising raw data into constructed, defined objects. This approach is beneficial in two ways, security being the first. The response has a defined object which it expects to return, therefore if for some reason the data has been manipulated through some kind of attack, the request will cause a 400 server error due to the data being returned not matching the specified model, thus not returning any protected data and improving application security. The second reason is it provides more robust and type-defined code. Using the previously mentioned openapi-generator-cli, these Pydantic models define the response structure expected by the front-end which can then be used to access properties *[figure 9]*.

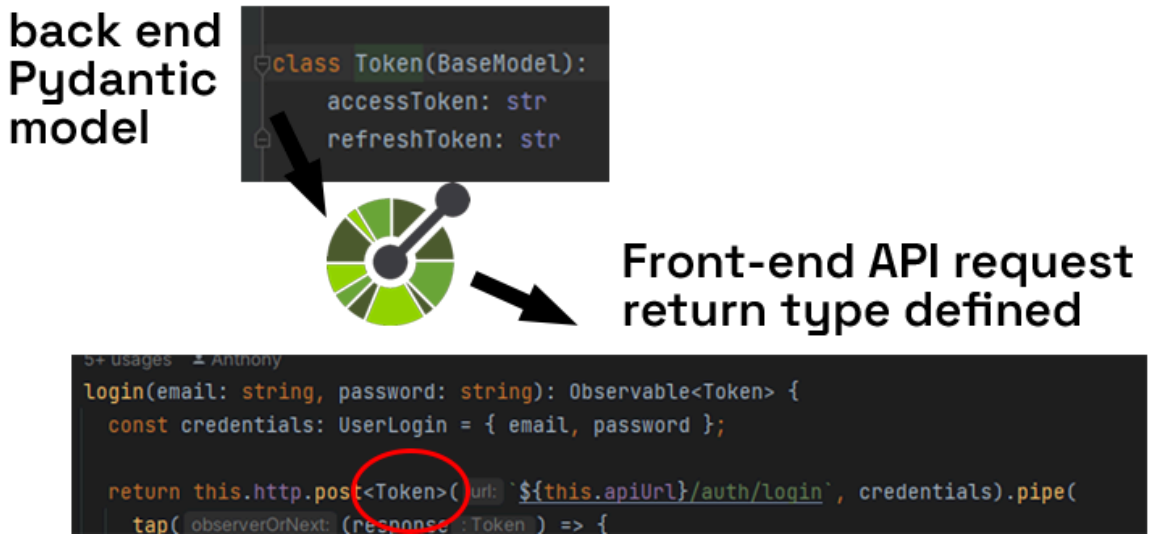


Figure 9 - OpenApi CLI generator diagram

This register endpoint takes UserRegistration model data as part of the path parameters, which contains the sign-up information required *[figure 8]*. From here we first utilise the auth_handler to hash the user's password, before storing the user's login details in the system's database. Finally, the response is handled. The response model of Token returns an access and refresh token. Important to note, to begin with, the system only returned the access token, as the

functionality for refresh tokens was added later. The endpoint then uses the new user's id, and requests for an access token for this user to be generated by the auth_handler, before being returned to the client.

The login endpoint shares the same response type as the register endpoint, however, how achiev this response differently. It instead takes a model of type UserLogin and queries the database to try and find a user with the given email. If a user is not found the endpoint will deny the request with a 401 error being returned. Otherwise it will use the auth_handler to compare the stored hashed password along with the provided plaintext password, once again a 401 is returned if they don't match.

With the API's endpoints written for login and register functionality, development switched to the front end. Angular documentation suggests the use of services; They provide a way to separate data and functions that can be used by multiple components throughout the application. Therefore the application uses an auth-service to handle the processes of authentication for the front-end, communication with the API for instance. The auth service was continuously developed and evolved throughout the project. Initially comprised of login, register and logout functions. They all use RXJS, a powerful javascript library used with reactive programming, specifically asynchronous code, therefore crucial due to the nature of the project's API being structured.

The Login and Register functions processes are very similar. They both take data provided by the user and make a request to the API. The response is tapped into, with the tokens then stored in local and session storage. Unfortunately, the project wasn't able to use HTTP-only cookies, which are considered better for storing tokens than local and session storage, but due to the project structure, this wasn't possible. Logout doesn't make a HTTP request and instead simply removes the tokens, due to auth-guard (which will be discussed later) will automatically return the user to the login page.

Finally to discuss the actual front-end pages. All of which are defined as separate components. The login form utilises Angular's FormBuilder to construct the login form. This provides better controls over the individual field and allows for validation to be applied *[figure 10]*. Each field within the HTML has custom error messages, and the form submit button is disabled until all fields are valid. The CSS was written to style the components mostly utilising global styling. Once valid and the form submit button has been clicked, the component makes a call to the auth service's login function. Then with the received response, will either redirect to the dashboard page or display the error message returned, depending on the success of the request. The register page is structured the same and operates in a very similar way, except notably contains more validators within the form fields. The form utilises custom build form validators, one to match the password and confirm password fields match *[figure 11]* and the other to ensure the project password policy is followed: at least one uppercase, lowercase character and a number.

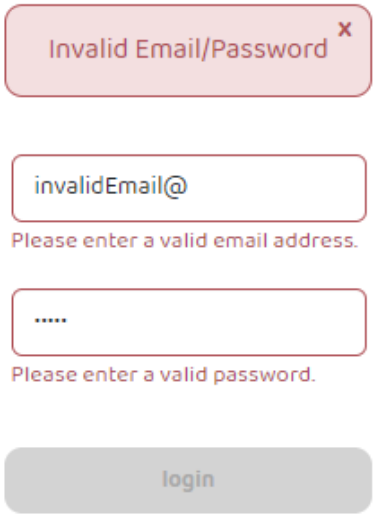


Figure 10 - Invalid login error and form validation errors

```
export function passwordValidator(): ValidatorFn {
  return (control: AbstractControl): { [key: string]: boolean } | null => {
    const hasUppercase = /[A-Z]/.test(control.value);

    const hasLowercase = /[a-z]/.test(control.value);
    const hasNumber = /\d/.test(control.value);

    const isValid = hasUppercase && hasLowercase && hasNumber;

    return isValid ? null : { 'passwordRequirements': true };
  };
}

2 usages  Anthony
export function matchPasswordValidator(matchControlName: string) {
  return (control: AbstractControl): { [key: string]: boolean } | null => {
    const matchControl = control.root.get(matchControlName);

    if (matchControl && control.value !== matchControl.value) {
      return {'passwordMismatch': true};
    }

    return null;
  };
}
```

Figure 11 - Custom field validation for user registration

4.4 Loading state with RSJX

As previously mentioned, due to the project's asynchronous nature there are numerous requests to the API which their responses must wait for. To solve this issue, the project uses a global

loading component to provide visual feedback to users, with the challenge being how this can be achieved.

The simple section of this functionality was the actual loading component. Taking two inputs, text which is defaulted to “loading…”, and whether it should be an overlay within the container or cover the entire screen *[figure 12]*. The HTML simply applies the conditional loading class to the div and displays the small animated loading graphic, designed and animated with CSS *[figure 13]*.

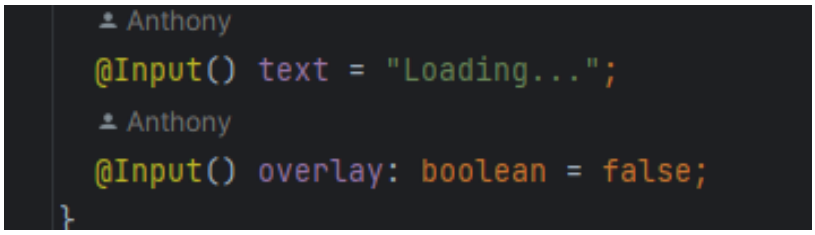


Figure 12 - Loading component input parameters

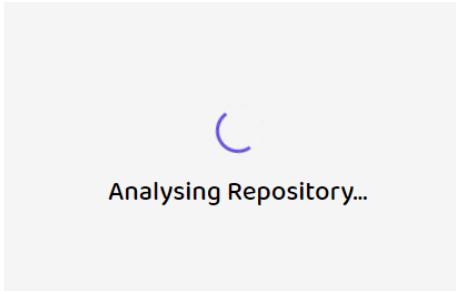


Figure 13 - Loading component example

There are two ways the loading component can be used. Firstly, contained within another page the component can be injected and controlled by the stand-alone component. For instance, the login page. While waiting for the response during login, a variable ‘submitted’ is set to true enabling the visibility of the injected loading component. The more complex way to activate the loading component is to use a service to control the visibility of the loading component, allowing it to be activated globally. The way this works is by injecting the loading component into the root HTML document and using a behaviorSubject within the service *[figure 14]*, emitting the boolean value to the subscribed methods. This subscription is then passed into the loading component to control its functionality. The service uses public functions to set and deactivate the loading state *[figure 14]*. Importantly, to prevent memory leaks, the root component upon destruction will unsubscribe from the loading state behaviorSubject *[figure 15]*.

Figure 14 - Loading service

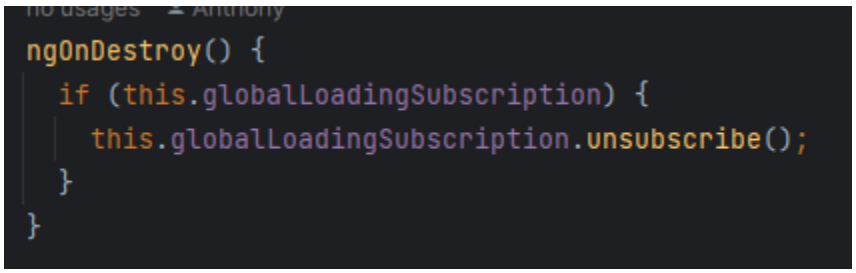


Figure 15 - Memory leak avoidance with ngOnDestroy

4.5 Token Expiry Handling & Refresh

To finalise development for the authentication tokens the project implemented refresh tokens and expiry handling. These tokens are imperative to how this application's authentication works. The overall concept of utilising both access and refresh tokens is to reduce the lifetime of access tokens. These access tokens are credentials and if they were intercepted the API could not distinguish between a genuine request and an attacker. Being able to reduce the time at which these tokens are valid, doesn't limit the chance of an attack but does mitigate the potential damage.

This is where refresh tokens are introduced, during an authentication request (login or sign-up), the user is issued two tokens, the access token is passed between the client and API during

requests, and then once expired, the client uses the refresh token to request an updated access token.

The implementation from the API's perspective is straightforward, we re-use the auth_handler to also encode a refresh token which can be attached along with the access token. The other aspect of the API is the endpoint to grant a new access token. It takes the refresh token as a parameter, and uses the auth_handler to verify its authenticity, before encoding a new access token. This is then returned by the endpoint. The refresh tokens are set to expire after just over a week, 10800 minutes, this way the user will automatically remain logged in for this time, and every 20 minutes a new access token will be granted. Once the refresh has expired, the API will respond with a 401 error during a refresh request, and then log out the user.

The front-end implementation is more complex, it must be able to handle the refresh of tokens and do this seamlessly, hiding the process from the user. To overcome this challenge the front-end uses an Angular HttpInterceptor, which can intercept both outgoing and incoming HTTP requests to compute functionality. In this case, the project uses an Interceptor to intercept all responses *[Figure 16]*, which looks for a response code of 401 and the message being 'Token has expired'. When this response has been returned, the interceptor will handle this issue. It will first send a request to the API for a new access token, using the refresh token to authenticate. The response is tapped into, and the updated token is stored before the previously 401 error request is retried *[Figure 17]*.

```
intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return next.handle(request).pipe(
    catchError( selector: (error) => {
      if (error instanceof HttpErrorResponse && error.status === 401 && error.error.detail === 'Token has Expired') {
        return this.handleTokenExpiration(request, next);
      }
```

Figure 16 - HTTP interceptor finding token expired response

```
private handleTokenExpiration(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return this.authService.refreshAccessToken().pipe(
    mergeMap( project: () => {
      window.location.reload();
      return next.handle(request);
    }),
    catchError( selector: (refreshError) => {
      return throwError(refreshError);
    })
  );
}
```

Figure 17 - Erroneous request retry functionality

This section completes the application's authentication flow. The entire user authentication system can be simplified and demonstrated through 9 key steps, visualised with *[Figure 18]*.

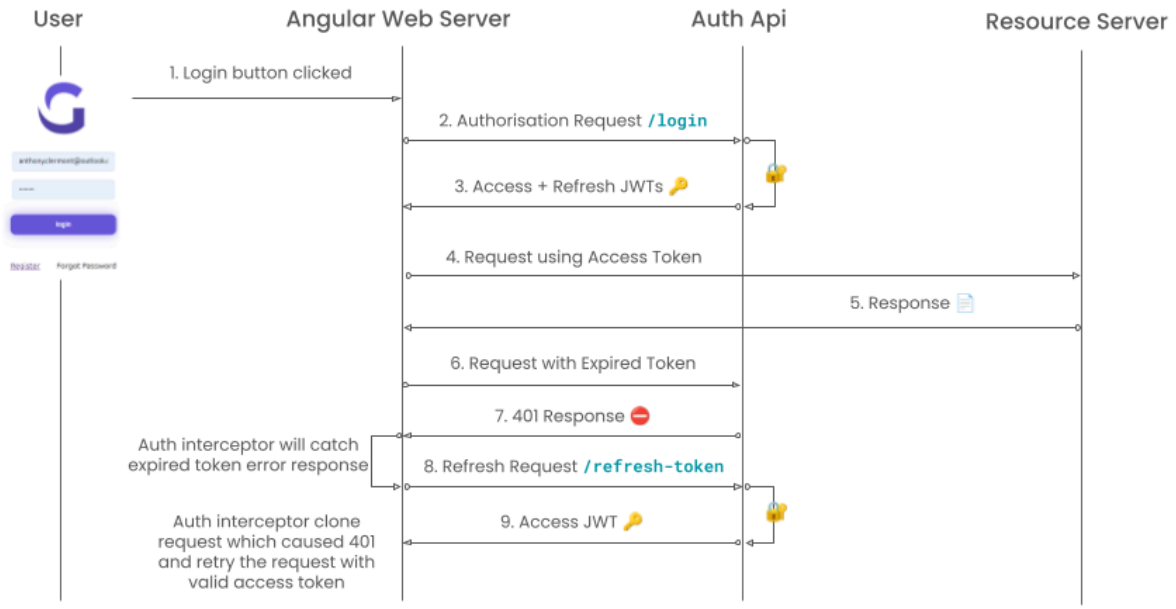


Figure 18 - Application's authentication flow diagram

First, the user enters their credentials and clicks login. An authentication request is then sent to the authentication API which validates the user's credentials before generating and returning access and refresh JWTs. The user then makes a request and the web client will attach the stored access token to the header of the request and the server will respond with the resources requested.

At some point, the client will make a request with an expired access token and the API will respond with a 401 "Token Expired" error. The auth interceptor will catch this response and make an access token refresh request using the refresh token. The Auth API will validate the refresh token before returning an updated access token. From here the auth interceptor has cloned the erroneous request and will store the updated access token before re-tyring the request.

4.6 Route Authorisation Control

The final aspect of authentication within this project was the protection of routes on the front-end. Some routes should only be accessible for authenticated users, some should only be accessible for non-authenticated users. For instance, the login and sign-up pages. To ensure this, the project uses guards and specifically Angular's CanActivate function in the app routing to protect these routes. These custom functions return a boolean value whether to accept or deny the route.

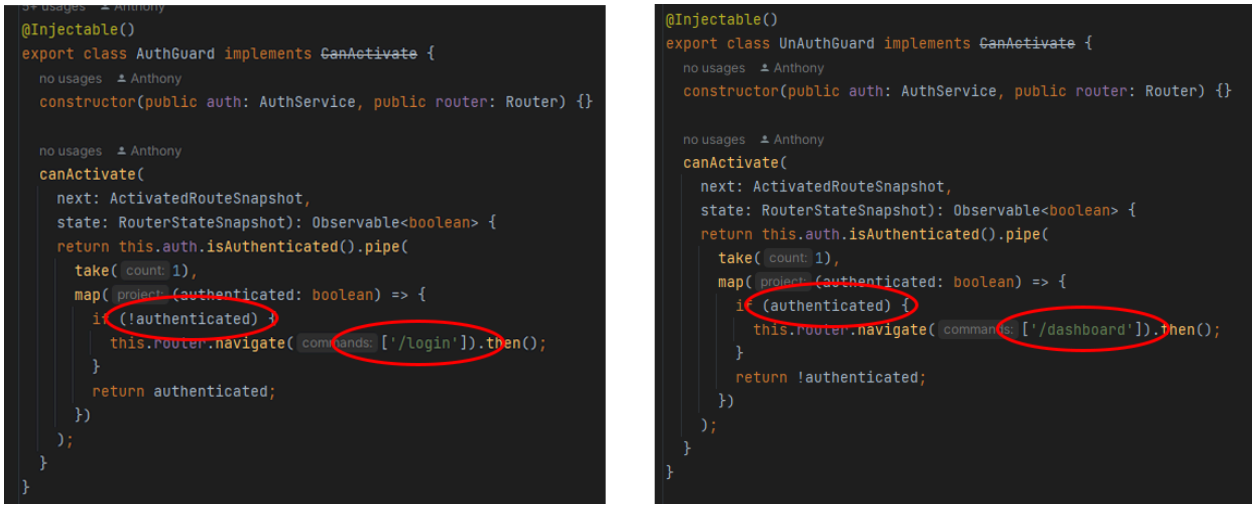


Figure 19 - Front-end route guards

As seen in *[figure 19]*, the authGuard and unAuthGuard are identical, apart from wanting opposite results, these guards are then included within the app's routing module. They make a call to the auth_service's isAuthenticated function, which returns a boolean value of whether the user is authenticated.

This function performs complex but important logic, and due to the number of calls to this function, relying on the API to verify the user's authentication status would be a great waste of resources and time. Therefore, caching was implemented to store recently results of api calls for certains api call. The first thing this function does is check if the cached response is less than 10 seconds old, if so the function uses this cached response. Otherwise, it will ensure a token is stored within the client browser, and if there isn't one, of course, the user isn't authenticated and will return false. Finally, a call is made to the API, which validates the authenticity of the access token, but instead of returning a 401 if it's not valid, will return a boolean value. The response from this request is tapped into and stored within the cache, before the result is returned as an observable ready for the guard to subscribe and utilise the response.

Planning ahead, I knew the project would also incorporate specific routes which depend on a connected GitHub account, for this reason, I also implemented the same guard and unGuard for Github routes. Both perform in the same way as the auth guards, checking the cache for a valid response and if not it will call an API endpoint to verify the user has a connection with GitHub. The endpoint to compute this is straightforward, it uses the provided JWT token to try to find the user's GitHub access token in the database and return the boolean value of this check.

4.7 App Flow and Navigation

After a large amount of back-end work and the groundwork for the front-end development complete, the focus shifted towards the usability of the application's interface. The routing module is the centre of the interface's routing, the module contains an array of Route objects,

each of which defines a separate route within the application. As mentioned in the previous chapter, these routes can be configured with parameters such as CanActivate using the guards to restrict access *[Figure 20]*.

```
{ path: '', redirectTo: 'dashboard', pathMatch: 'full'},
{ path: 'login', component: LoginComponent, canActivate: [UnAuthGuard]},
{ path: 'register', component: RegisterComponent, canActivate: [UnAuthGuard]},

{ path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard, GithubGuard]},
```

Figure 20 - App routing using guard protection

By default, the application attempts to route the user to the dashboard component *[figure 20]*. However, this component is protected with both the authGuard and githubGuard therefore to reach the dashboard page both these guards must return true. As seen in *[figure 19]*, the authGuard if not authenticated will redirect the user to the login page. The same process occurs within the GitHub guard, however, this time sending the user to '/github-connect' if the authenticated user hasn't connected their GitHub. This approach ensures no matter the state of the user, they will always be directed to the appropriate page.

Another attribute which can be included within a Route object is another array of Route objects, called child routes. These child routes share the default path of the parent route *[figure 21]*. The repository-specific routes all begin with '/repository' and then include both the repoOwner and repoName parameters; these URL parameters are used within the component's functionality and will be discussed later. Each subsection of the repository pages are defined *[figure 21]*, with the default being the repository overview page. Importantly, none of the child routes need to be protected by guards, due to the parent component being protected.

```
{ path: 'repository/:repoOwner/:repoName',
  canActivate: [AuthGuard, GithubGuard],
  children: [
    { path: '', component: RepositoryOverviewComponent},
    { path: 'commits', component: RepoCommitHistoryComponent},
    { path: 'commits/:sha', component: CodeFileChangesComponent},
    { path: 'issues', component: IssuesPageComponent},
    { path: 'reports', component: ReportPageComponent}
  ]
},
```

Figure 21 - Nested child routes within the parent

4.8 GitHub Authentication

To continue development, GitHub integration needed to be completed. As stated earlier, the basic setup of the Oauth application was already complete, however, a major obstacle was still present, how does the authentication flow work. Learning and understanding much of GitHub's API proved challenging, due to my lack of experience and the complexities of the Oauth system. Fortunately, GitHub provides excellent documentation for how to set up and grant access tokens to users. The authentication flow is as follows and can be visualised in *[figure 22]*.

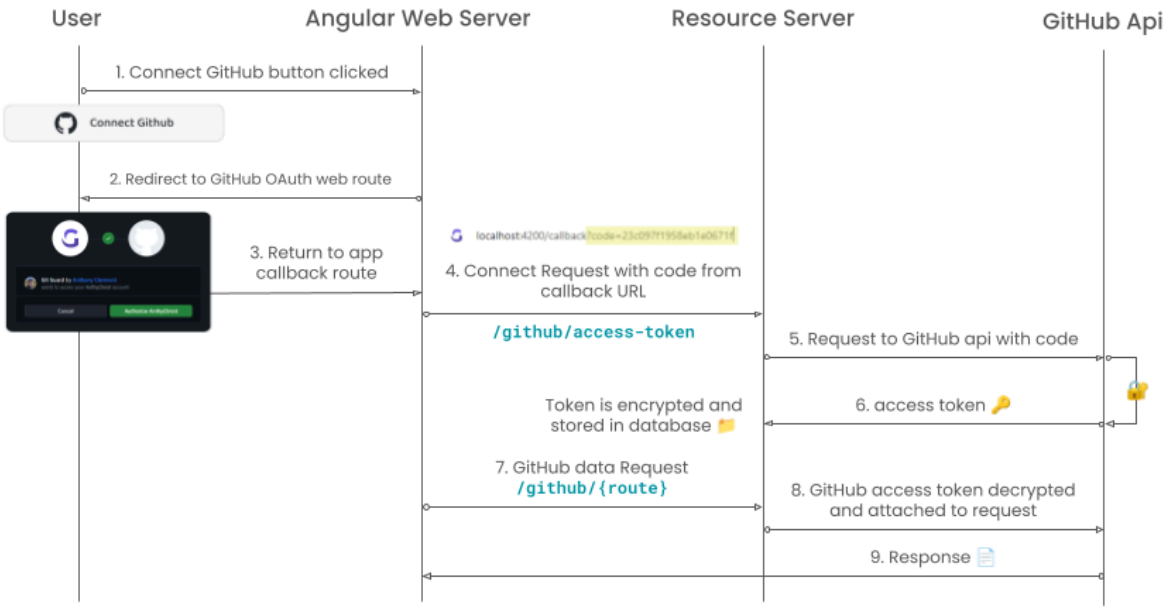


Figure 22 - GitHub connection and authentication flow

The first step is to redirect the user to GitHub and the project's Oauth connection webpage, from here the user can either accept or deny the connection, however, this explanation will demonstrate a successful authentication. Once authorised by the user, GitHub returns to the application at a specific callback route, with a query parameter contained. This code can be given to GitHub to grant a full access token.

To discuss the actual implementation, the first step was to create the API endpoint to request an access token from GitHub using the temporary code. Making this call from the back end ensures no sensitive codes are passed over to the front end which would compromise the tokens integrity.

Then the callback route was created, this component doesn't contain any HTML or CSS. The logic works within the initialisation of the component, first utilising the loading service to set a loading view to the user *[figure 23]*, and then subscribing to the route parameters in attempt to pull out the temporary code returned from GitHub's authorisation page. From here, if a code is present *[figure 23]*, it will make a call to the API endpoint previously discussed, stop the loading visual and finally redirect the user to the dashboard page. Alternately, if no code is present or an

invalid code is presented the loading state is stopped and the user is redirected back to the github-connection page.

```
1 usage   1 Anthony
ngOnInit() {
  this.loadingService.setLoading({
    active: true,
    text: 'Establishing GitHub Connection...',
    overlay: true
  });
  this.route.queryParams.subscribe( observerOrNext: (params :Params ) => {
    const code = params['code'];

    if (code) {
      this.userService.connectGitHub(code)
        .subscribe( observerOrNext: () => {
          this.authService.clearCache( key: 'githubConnected');
          this.loadingService.deactivateLoading();
          this.router.navigateByUrl( url: '/dashboard')
        })
    }
    else {
      this.loadingService.deactivateLoading();
      this.router.navigateByUrl( url: '/github-connect')
    }
  })
}
```

Figure 23 - Extracting the temporary code from URL parameters

The final aspect of this implementation was the encryption of the access tokens. As discussed in my research, application-level encryption would be implemented to securely store the GitHub access tokens. To achieve this, another Python package was installed, 'Fernet'. Utilising a new utils file within the API, it contains two functions which handle the encryption and decryption of data, using a 32-bit secret pre-generated and stored in the .env file. The encryption function is called just before the tokens are stored in the API, and then the decrypt function when queried from the database. Due to this SQL command being executed regularly, the performance of the encryption/decryption was greatly considered, and after testing the query time of different solutions this Fernet package proved to be the best option.

4.9 Front-end Structure and Shared UI Components

Thus far, this text has glossed over a large amount of work which was carried out for the front-end, HTML page structure and CSS styling for instance. Nonetheless, it was imperative development work which consumed a great deal of time and resources. Therefore, this chapter will dive into some of the key front-end design issues and how they have been addressed.

To begin with, understanding the UI's structure is important. Angular apps are technically single-page applications. They run off a single root page, the content is rendered through the 'router-outlet'; a placeholder that Angular dynamically fills based on the current router state. This root file is also where the navigation and application bar are located, this approach ensures these components are only declared once yet included regardless of the route.

The largest difficulty during implementation was the CSS, how to place the different components in the right place on the client's screen. To solve this, the root file uses a grid display, specifically grid-template-areas *[figure 24].* The solution to how each part only uses the correct spacing is using "auto 1fr", this makes the element in the auto column/row take up only the space it requires and then the next element to fill the remaining space. For this application, the nav bar takes up its required space, and then the router occupies the remaining area (minus the height of the app bar).

```
display: grid;
grid-template-areas:
"navbar appbar"
"navbar content";
grid-template-columns: auto 1fr;
grid-template-rows: auto 1fr;
```

Figure 24 - Root application pages CSS grid layout

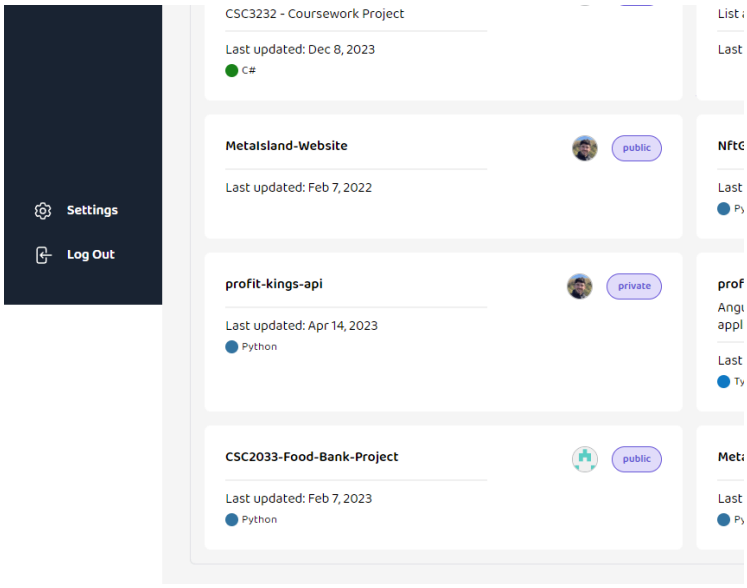


Figure 25 - Navigation bar overflow scrolling issue

With this however, the router outlet caused an unusual issue. As seen in *[figure 25]*, if the current route's height was larger than the height of the client's browser the navigation bar would not respect scrolling, and the app bar would not be present. To resolve this, the router outlet is contained within a div which ensures the height of the outlet is restricted to the viewport's height minus the height of the app bar, with overflow (scrolling) restricted to only show for the div which contains the router outlet.

Furthermore, some routes shouldn't contain either the navigation or app bar, for instance, the login page. To resolve this, the component uses the auth service and checks whether the user is authenticated, hiding the component if not.

Lastly, before discussing some of the actual components, many of the components within the application utilise the loading component due to many of them making API calls during initialisation. For this to occur, the HTML uses conditional expressions to show the loading component until the data it needs has been received *[figure 26]*.

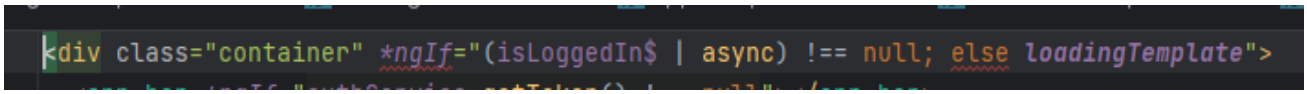


Figure 26 - Observable subscription within HTML and loading component usage

Beginning with the application's navigation bar. The design of the navigation bar evolved throughout this project, each iteration improving in both appearance, functionality and interaction experience. Depending on the current route of the user, the links required change. For instance, if within a repository, all of the child routes within the repository need to be shown but outside a repository hidden. To achieve this, a small function, was made which uses the current route and checks if it begins with 'repository', again using conditional HTML expressions to hide or show the links *[figure 27]*.

Figure 27 - Function to show/hide repository-specific navigation routes

Each link, when active, is also styled to show if it's the active link. This is achieved with Angular's routerLinkActive, which tracks the current route to see if it matches the routerLink of the 'a' tag *[figure 28]* and applies the 'active' class if true. This feature is purely visual but is common practice within web development, and increases application usability.

```
<a *ngIf="hideLinksOnGithubPages() && !hideLinks()" [routerLink]="['/dashboard']" routerLinkActive="active">
  <ng-icon name="heroSquares2x2"></ng-icon>
  <h4>Dashboard</h4>
</a>
```

Figure 28 - Applying active CSS styling to navigation routes

The application bar is the other major shared component. This component is designed to provide a better understanding to the user of the current navigation, and is important due to the subroutes within the application. For instance within a repository, a user may also be within the commits page and even on a specific commit, for this structure a breadcrumb menu seemed most appropriate *[figure 29]*.

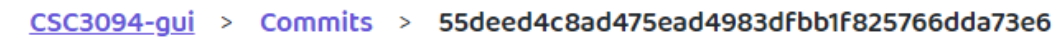


Figure 29 - Application's breadcrumb menu in the app bar

The implementation of the breadcrumb menu proved difficult. To achieve this, a function was made called 'parseRouteSegments', which runs during the page initialisation *[figure 30]*. It takes the URL as a function parameter and begins by splitting the string by forward slashes to construct an array of each element in the URL. The issue was a repository link is followed by the repoOwner and repoName which aren't specific routes within the application, and therefore shouldn't be separate crumbs in the menu.

To solve this the function loops through the separated links checking if one matches 'repository', if true, it will take this element and the next two in the array, combine them to be one part of the breadcrumb, before incrementing the count by 2 to skip the repoOwner and RepoName. This loop also is responsible for constructing the menu objects. This object contains the display text and the routing link *[figure 31]*. This display text is just capitalising the first letter in the string, but for the specifically joined repository link, only the repository name is included, as the repoOwner is the user of the application and is unnecessary to include. The HTML then uses this list of constructed objects to create the menu, with the display name shown as text and the link included in the 'a' tags route *[figure 29]*.

```
1 usage  Anthony
parseRouteSegments(url: string): RouteSegment[] {
  const segments = url.split( separator: '/').filter(segment => segment !== '');
  let routeSegments: RouteSegment[] = [];

  for (let i = 0; i < segments.length; i++) {
    if (segments[i] === 'repository') {

      let displayName = `${this.capitalizeFirstLetter(segments[i+2])}`;
      routeSegments.push({
        link: '/' + segments.slice(0, i + 3).join('/'),
        displayText: displayName
      })
      i += 2;
      continue;
    }

    let displayName = this.capitalizeFirstLetter(segments[i]);
    if (i < segments.length - 2 && segments[i + 1] !== 'repository') {
      displayName += ' ' + this.capitalizeFirstLetter(segments[i + 1]);
      i++;
    }

    routeSegments.push({
      link: '/' + segments.slice(0, i + 1).join('/'),
      displayText: displayName
    });
  }

  return routeSegments;
}
```

Figure 30 - Route separation and breadcrumb menu link creation

```
3 usages  Anthony
interface RouteSegment {
  2 usages  Anthony
  link: string;
  2 usages  Anthony
  displayText: string;
}
```

Figure 31 - Model used for breadcrumb menu links

4.10 Dashboard Webpage

This section will explore the general structure of routes within a repository and the process of extracting information from the GitHub API. The dashboard page is the default route for

authenticated and GitHub authenticated users. Its purpose is to display all the user's repositories.

To achieve this, a new endpoint was created which fetches the requested data. A lot of the endpoints similarly interact with GitHub, thus this explanation applies throughout. It will first use the user ID extracted from the provided JWT to query the database for the user's GitHub access token and includes this token within the headers. These headers are attached to the GitHub request calling the appropriate API endpoint. In this case, the user's repositories *[figure 32]*. Importantly, the endpoint is defined asynchronously, and we await the response from GitHub. Once we receive a response if the status code isn't 'OK', the endpoint will stop and return an error. Otherwise, it will extract the JSON response and use the Pydantic model, to map the wanted data from the response, before returning the repository data.

```python
@github_router.get("/repos", response_model=List[GitHubRepo])
async def getRepos(user_id=Depends(auth_handler.authWrapper)):
    token = getGitToken(user_id)
    if token:
        headers = {"Authorization": f"Bearer {token}"}
        async with httpx.AsyncClient() as client:
            response = await client.get("https://api.github.com/user/repos", headers=headers)

        if response.status_code != 200:
            raise HTTPException(status_code=response.status_code, detail="GitHub API request failed")

        github_repos = response.json()
        mapped_repos = [GitHubRepo(**repo) for repo in github_repos]

        for i, repo in enumerate(mapped_repos):
            mapped_repos[i].commitsUrl = HttpUrl(str(repo.commitsUrl).replace('%7B/sha%7D', ''))

            if mapped_repos[i].language is not None:
                mapped_repos[i].languageColour = get_language_colour(mapped_repos[i].language)

        return mapped_repos
    else:
        raise HTTPException(status_code=400, detail="Github not connected")
```

Figure 32 - API's get repositories endpoint implementation

Most of the processes the front-end performs with this are extremely similar to what has previously been discussed. A service was created, to contain functions capable of making requests to the individual API endpoints, with some functions implementing caching. The dashboard component's typescript file uses the 'getRepos' function to store the observable response *[figure 33]*, this response can be subscribed to within the HTML. This approach, when possible, is considered better than subscribing within the typescript as it prevents memory leaks due to automatic unsubscribing after page destruction. *[figure 34]* shows this implementation,

and how the loading component is shown while we wait for the API to give the client the requested information.

```
2 usages   Anthony
userRepos$: Observable<GitHubRepo[]> = this.userService.getRepos();
```

Figure 33 - API call stored as observable ready to be subscribed

```
<section *ngIf="userRepos$ | async as userRepos; else loadingTemplate">
```

Figure 34 - Subscription of observable within HTML

The interface is constructed using a grid approach. The challenge was how to get the repository cards within a responsive grid. To solve this, the project uses auto-sizing grid columns, the min-max CSS declaration defines the minimum space for each card '450px' and for the grid to auto-fill the number of columns depending on the client screen width, *[figure 35]* showing the grid on a large device and *[figure 36]* on a smaller device.

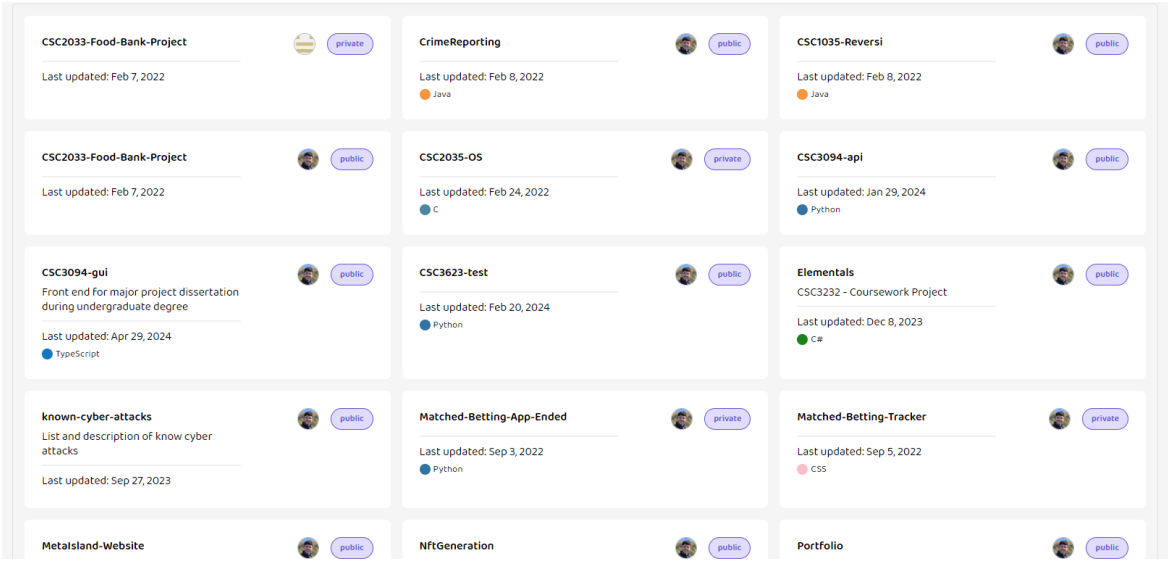


Figure 35 - Larger screen grid layout

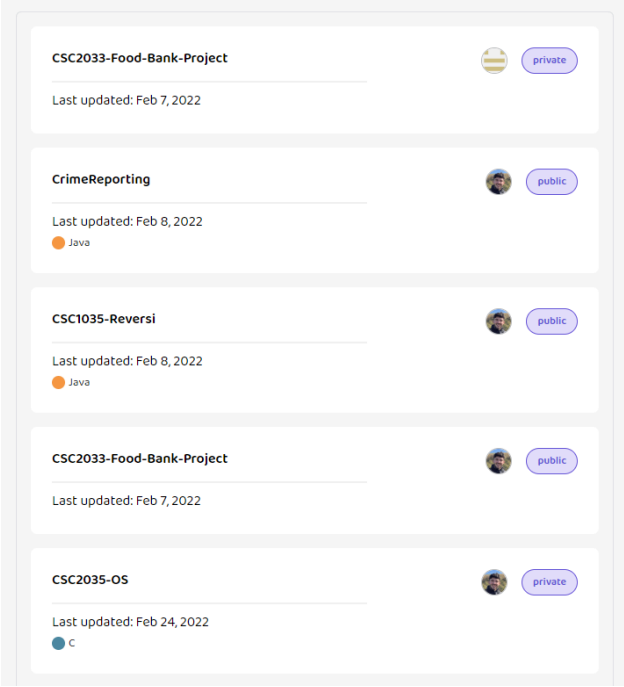


Figure 36 - Smaller screen grid layout

4.11 Repository Overview Webpage

The repository overview page is the home root when within a repository, and the development of this page caused considerable issues. This section will explore these issues and explain how they were addressed.

The first obstacle was how the repository pages could understand which repository they needed to gather data for. At first, I considered using a service along with local storage to store the current repository. However, if you recall chapter 4.7 (App Flow and Navigation), the repository routes are defined in such a way as to include the repository owner and name *[figure 37]*.

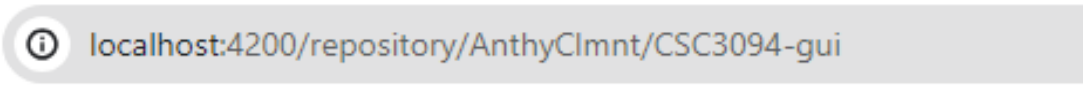


Figure 37 - Example URL route within a repository

```
ngOnInit(): void {
  this.repoOverview$ = this.route.params
    .pipe(
      switchMap( project: (params :Params ) => {
        if (!params['repoOwner'] || !params['repoName']) {
          this.router.navigateByUrl( url: '/')
          return of( value: null);
        }

        this.repoOwner = params['repoOwner'];
        this.repoName = params['repoName'];

        return this.userService.getRepoOverview( params: {
          repoOwner: this.repoOwner,
          repoName: this.repoName
        })
      })
    );
}
```

Figure 38 - NgOnInit function to extract repository information

This allows the component to extract these parameters from the URL during component initialisation. As seen in *[figure 38]*, the current route observable is piped, and the code ensures both parameters are contained within the URL. If not, the observable containing the overview is returned to be null. This ensures the loading component is deactivated, preventing a never-ending loading screen. Assuming these two parameters are within the URL, the function makes a call to the user service which in turn makes a request to the API. The observable returned by the request is then stored ready to be used within the HTML.

The next challenge was what should be shown on the overview screen. The implementation of the analysis would be included later, but there is a lot of useful information which could be extracted from GitHub. The endpoint created for the repository overview had many iterations throughout development. Originally, the overview page contained information such as the contributors within the repository, contributor commit count and some general information about the repository.

However, due to the future development of the code analysis, the page seemed too crowded, with salient information not being obvious. Therefore, most of the data provided on the overview page was mainly collected from the database, and only the general information of the repository was requested from GitHub *[figure 39]*.

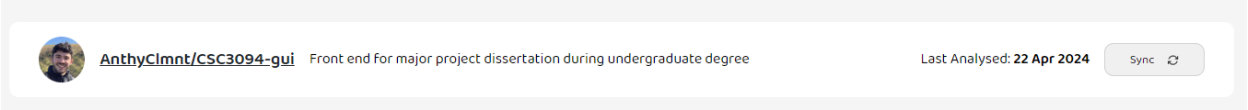


Figure 39 - Basic repository information gathered directly from GitHub

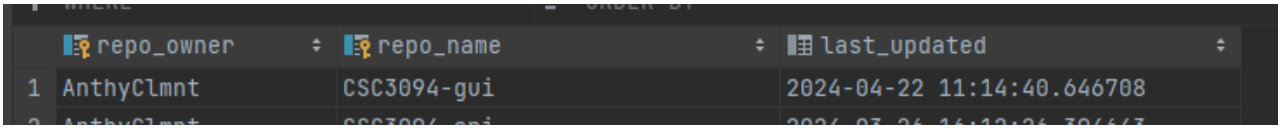
4.12 Sync Repository APIs

This section will explore the beginning of the application's code analysis, and the groundwork undertaken for it to be possible.

One of the largest issues during development was preventing wasteful requests. As previously mentioned, a part of this issue was addressed through caching responses, but another huge aspect of this issue was wasteful requests to GitHub. To perform the code analysis, the back-end needs access to the actual code, and what was changed within commits. If the application constantly updated and ran the repository analysis, gathering the commits and code changes from GitHub's API, it would severely impact the application performance and waste resources.

Therefore, the project's solution to this issue was to control which commits the requests would gather. It achieves this by utilising a new table within the database, 'RepoLastAnalysed', a primary combined key of both repository owner and name that ensures the integrity of the last_updated timestamp, which stores when the repository was last analysed *[Figure 40]*.

GitHub's endpoint to get commits from a repository can also include a timestamp of what datetime to fetch commits from, using this allows the project to only gather the commits which have not already been analysed *[figure 41]*.



	repo_owner	repo_name	last_updated
1	AnthyClmnt	CSC3094-gui	2024-04-22 11:14:40.646708

Figure 40 - Example repository last analysed database entry

```
if since:
    since_datetime = datetime.datetime.strptime(since, "%Y-%m-%d %H:%M:%S.%f")
    since_isoformat = since_datetime.strftime("%Y-%m-%dT%H:%M:%SZ")
    params['since'] = since_isoformat

async with httpx.AsyncClient() as client:
    response = await client.get(
        f"https://api.github.com/repos/{repoOwner}/{repoName}/commits",
        params=params,
        headers=headers
    )
```

Figure 41 - Commit request function along with optional since date in the request params

This updating of the repository is controlled by the user through the front end. Within the repository overview screen, a new button was included which, when clicked, makes a request to the API to update the repository. As well, the last analysed time was also included within the data passed to the overview page, this way could it can be included within the page to show the user when the repository was last analysed *[figure 39]*.

Angular contains a date pipe, which is used to format timestamps to be more user-friendly. However, this project utilises its own, custom pipeTransform to better show this updated time. The pipe can be included within the HTML, automatically formatting the timestamp. The transform function handles the actual logic *[figure 42]*. Firstly, it ensures the given date is of type timestamp, and then it calculates if this timestamp is either today's or yesterday's date. If so, instead of returning the date and time it will return 'Today, 11:40' for instance *[figure 39]*. This small improvement allows for user interaction to be more clear and easier to read. Furthermore, the overview page also calculates whether the last sync was more than a week ago, and if so an alert will be shown to the user, indicating they should update the repository.

```
2 usages  Anthony
transform(inputDate: Date | string): string {
  const today = new Date();
  const dateObj = typeof inputDate === 'string' ? new Date(inputDate) : inputDate;

  const yesterday = new Date(today);
  yesterday.setDate(yesterday.getDate() - 1);

  if (dateObj.getDate() === today.getDate() &&
    dateObj.getMonth() === today.getMonth() &&
    dateObj.getFullYear() === today.getFullYear()) {
    return `Today, ${new DatePipe( locale: 'en-UK').transform(dateObj,  format: 'HH:mm')!}`;
  }

  else if (dateObj.getDate() === yesterday.getDate() &&
    dateObj.getMonth() === yesterday.getMonth() &&
    dateObj.getFullYear() === yesterday.getFullYear()) {
    return `Yesterday, ${new DatePipe( locale: 'en-UK').transform(dateObj,  format: 'HH:mm')!}`;
  }

  else {
    return new DatePipe( locale: 'en-UK').transform(dateObj,  format: 'dd MMM yyyy')!;
  }
 }
}
```

Figure 42 - Custom pipe transform for better date readability

The endpoint which updates the repository also returns the updated overview data. Therefore the front-end utilises the data returned to update the stored repository data. As this data is stored as an observable, the overview page automatically updates once the data has been received.

4.13 Code Analysis Calculations

Development now focused on the implementation of the code analysis calculations. This section will first re-visit the project's key metrics and then how these metrics have been implemented into the code base.

To summarise the project's research. The application intended to implement three metrics: Cyclomatic Complexity, Lines-To-Comment Ratio and Maintainability Index. The actual implementation of the formulas for these calculations was straightforward to implement. However, to have the correct data for these calculations to work, some very challenging obstacles needed to be overcome.

Firstly, to have the actual code which has been changed within commits. When the request for repository commits is returned, the JSON data is structured, with each commit being a different object. Within a commit object, some general information is stored, the hash number, author data, and parent commit hash, skipping over this data to the end, is a nested object which contains all the files within this specific commit. *[Figure 43]* shows the response schema of this part of a commit. Each file contains some metadata, but for this project, only the filename and importantly the patches are extracted. The filename is used to determine which programming language has been used, and the patch contains the actual code which has changed.


```
"files":[
  {
    "filename":"file1.txt",
    "additions":10,
    "deletions":2,
    "changes":12,
    "status":"modified",
    "raw_url":"https://github.com/octocat/Hello-World/raw/7ca483543807a51b6079e54ac4cc392bc29ae284/file1.txt",
    "blob_url":"https://github.com/octocat/Hello-World/blob/7ca483543807a51b6079e54ac4cc392bc29ae284/file1.txt",
    "patch":"@@ -29,7 +29,7 @@\n....."
  }
]
```


Figure 43 - Example response schema of GitHub commit GET request

The patches are formatted to show a snippet of the file, with lines added starting with a '+' and lines removed starting with a '-'. The challenge is how the code can understand which lines should be included in the analysis calculations. To solve this, the patches are parsed through a function, splitting the lines using the special new line character '\n'. With this array of lines, the code loops through and checks if the string begins with the plus, indicating it's a new line which has been added, if so, the line is added to the final array ready to be returned when the loop ends. Importantly, the code removes the '+' from the line, ensuring this character is not included in the calculations.

With the code now parsed, the calculations can be performed. As mentioned the filename is used to extract the file extension, knowing the language of the file is beneficial in two ways.

Firstly, if it's a language the analysis doesn't or shouldn't support, the file can be voided and computation wouldn't be wasted on unsupported programming languages. Secondly, and more importantly, it can define language specific flags and declarations which should be picked up by the calculations.

```
match file_extension:
    case "py":
        comment_identifiers += ["#", "'''", '"""']
    case "java":
        comment_identifiers += ["//", "/*"]
    case "js":
        comment_identifiers += ["//", "/*"]
    case "ts":
        comment_identifiers += ["//", "/*"]
    case "html":
        comment_identifiers += ["<!--"]
    case _:
        return None

lines = code.split('\n')
total_lines = len(lines)
comment_lines = sum(
    1 for line in lines if any(line.strip().startswith(comment) for comment in comment_identifiers))
```

Figure 44 - Comment-To-Line ratio metric implementation

To discuss the easiest metric to implement, the Line-To-Comment Ratio. This formula is very simple, how many lines of code are there and how many comment lines are present? *[figure 44]*, shows the code implementation of this metric. Using a switch statement with the language type to know the character(s) which define a comment, and then using list comprehension, to go through each line in the code, and each comment identifier to know the number of comments within the change. I also considered the fact that some languages support multi-line comments, and this solution already handles this. As, if a multi-line comment is present, the analysis can still just pick out the starting identifier allowing the calculation to still be correct.

```
TokenInfo(type=63 (ENCODING), string='utf-8', start=(0, 0), end=(0, 0), line='')
TokenInfo(type=5 (INDENT), string='    ', start=(1, 0), end=(1, 4), line='    public static void main(String[] args) {\n')
TokenInfo(type=1 (NAME), string='public', start=(1, 4), end=(1, 10), line='    public static void main(String[] args) {\n')
TokenInfo(type=1 (NAME), string='static', start=(1, 11), end=(1, 17), line='    public static void main(String[] args) {\n')
TokenInfo(type=1 (NAME), string='void', start=(1, 18), end=(1, 22), line='    public static void main(String[] args) {\n')
TokenInfo(type=1 (NAME), string='main', start=(1, 23), end=(1, 27), line='    public static void main(String[] args) {\n')
TokenInfo(type=54 (OP), string='(', start=(1, 27), end=(1, 28), line='    public static void main(String[] args) {\n')
TokenInfo(type=1 (NAME), string='String', start=(1, 28), end=(1, 34), line='    public static void main(String[] args) {\n')
TokenInfo(type=54 (OP), string='[', start=(1, 34), end=(1, 35), line='    public static void main(String[] args) {\n')
TokenInfo(type=54 (OP), string=']', start=(1, 35), end=(1, 36), line='    public static void main(String[] args) {\n')
TokenInfo(type=1 (NAME), string='args', start=(1, 37), end=(1, 41), line='    public static void main(String[] args) {\n')
TokenInfo(type=54 (OP), string=')', start=(1, 41), end=(1, 42), line='    public static void main(String[] args) {\n')
TokenInfo(type=54 (OP), string='{', start=(1, 43), end=(1, 44), line='    public static void main(String[] args) {\n')
```

Figure 45 - Example output of tokenize function

The major issue with the other two metrics was how much more complex the calculations are, therefore the solution to how they can work with multiple languages, and not be Python specific like many packages already available, required more work and research.

Tokenerizer is a Python package which tokenizes code into token objects, containing metadata about each section of the provided code. For instance *[figure 45]*, shows how a patch of code has been converted into an array of these tokens, some of which types are indent, name or op (operator) to name a few. We then can use these tokens in the calculations.

Cyclomatic complexity utilises the token strings to match them with key language identifiers for complexity. For instance, python code may look for token strings like if, else, def, break, continue etc. Each time one is found, the patch's complexity is incremented by one.

The most challenging metric to implement was the maintainability index, which itself comprises Cyclomatic Complexity, Halstead volume and Lines of Code. LOC (lines of code) was easy to implement and the challenges for Cyclomatic Complexity were overcome. The Halstead Volume calculation also had its challenges. The Halstead volume also uses the output from the tokenizer function but picks out the number of operands and operators using *'token.type === tokenize.OP'* and *'token.type === tokenize.NUMBER'*.

An interesting bug which was uncovered had to do with how the result of the analysis was stored within the database. Originally, I didn't know the commit hash (commit sha) wasn't unique. Within a repository, the hashes are unique but there is no guarantee the hashes in different projects are. Therefore the table which stores the analysis originally only has a primary key on the commit sha, which needs to be altered to have a combined primary key on the commit sha, the filename, the repo owner and repository name. This table is then queried to get information out, such as the updated repository overview page extracting the needed information from the database.

Also, to ensure the metric calculations are accurate, the results of the implementation solutions were compared with existing Python package results, specifically radon. Which provides such metrics from source code. The project's implementation was tweaked to get results as close as possible to existing solutions.

Lastly, from competitor research and my own experience using similar software, SonarCloud, the grading system proved a great feature to include. Therefore the results from the analysis are run through a grading system to determine a visual indicator of quality.

4.14 Contributor Report Webpage

The last major part of the development was the contributor report page. Utilising the analysis stored in the database, individual contributor data could be extracted to provide alternative analysis.

Before implementation, the major aspect I wanted this feature to be able to show was how a contributor's quality has changed over time. Due to how the analysis table was set up, this was not possible, as the table didn't store the commit date. Therefore a migration script was written to alter the table, to include the timestamp of the commit.

33	46.8	0	2024-03-24T17:22:46Z
30	36.8	0.09	2022-01-18T14:46:51Z
24	<null>	0	2024-03-12T15:44:26Z

Figure 46 - Example database entry of analysis

The second issue this task faced was how to efficiently calculate and process the raw data into grouped averages of the different metrics. *[figure 46]* demonstrates an example of a piece of analysis stored in the database. Initially, the project simply extracted all the information from the database for the requested contributor and specified repository. With the back end performing the all of the computation. This approach to processing the raw data this solution wasn't optimal, as the time it took to process all this data seriously impacted application performance.

```
def get_repo_contributor_analysis(repo_owner: str, repo_name: str, author: str):
    try:
        conn, cursor = connect_db()
        cursor.execute("""
            SELECT
                DATE(commit_date) as commit_date,
                AVG(maintain_index) AS avg_maintain_index,
                AVG(ltc_ratio) AS avg_ltc_ratio,
                AVG(complexity) AS avg_complexity
            FROM
                commitFileAnalysis
            WHERE
                repo_name = ?
                AND repo_owner = ?
                AND author = ?
            GROUP BY
                strftime('%Y-%m', commit_date)
            ORDER BY
                commit_date ASC
        """, (repo_name, repo_owner, author))
        contributor_data = cursor.fetchall()

        close_db(conn)
        return contributor_data
```

Figure 47 - Improved repository analysis SQL query

To solve this issue, the computation was moved from the back end into the database query itself. *[figure 47]* shows the improved query. To understand how it works, the explanation will go from back to front. Grouping by a formatted commit date ensures the data selected will be

grouped into months, and sorting by this ensures the correct order: “Jan, Feb…”. The query takes in the repository information as well as the author to gather the data for and filters the table using these parameters. Finally, the query makes use of SQL’s AVG function, which calculates the average of the filtered results. The returned query data contains the commit date, along with the three calculated averages of the metrics. The only logic left for the back end to perform is rounding the values.

The front-end implementation begins with a familiar pattern. Within initialisation, a call is made to the API to get all the contributors on the given repository, with the returned observable being subscribed within the HTML. From here, when the user selects a contributor to generate a report for *[figure 48]*, a call is made to the endpoint which runs the average SQL query. The returned data is passed to a function which populates the empty HighCharts charts. Highcharts is a javascript library that enables this project to display powerful and excellently designed charts.

As a result of myself never having used this library, the learning curve was fairly steep and was a substantial obstacle to overcome. Thankfully due to the good documentation, a specific reason this chart library was chosen, I was able to set up the graphs for the three calculated metrics, showing a contributor’s progression or regression over time *[figure 48]*.

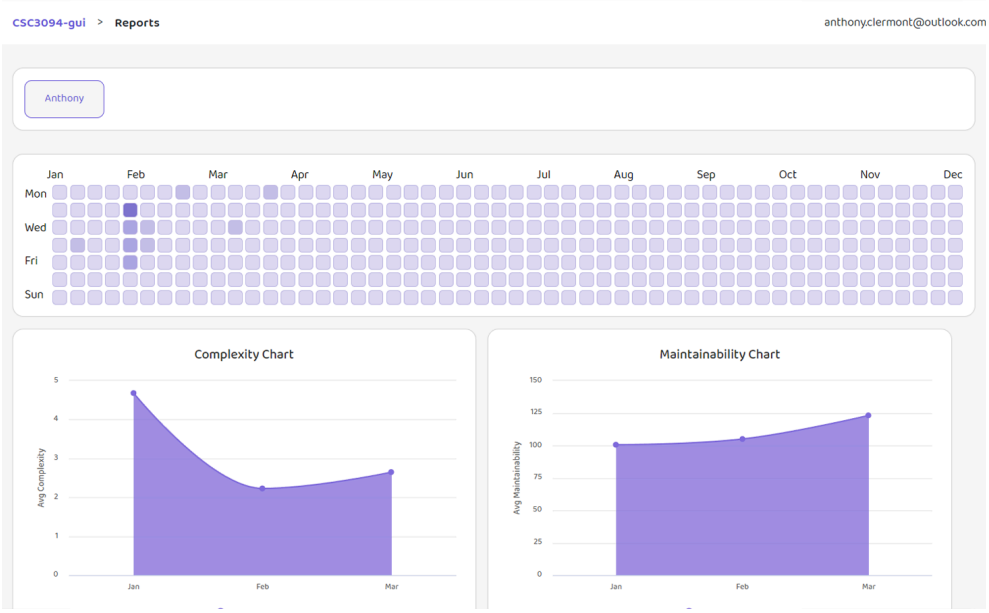


Figure 48 - Outcome of the contributor report page

5. Testing

This next chapter will explore and discuss the application’s testing practices and evaluate the outcome of the implemented testing.

5.1 Unit Testing

Although not mentioned, unit testing was continuously integrated alongside application functionality to achieve one of the project's key objectives. A major aspect of development for any software engineering project is testing. Regardless of the functionality implemented, if not properly tested, it will affect the quality of the final product.

Unit tests are beneficial in two ways. Firstly, bugs are inevitable, these tests can identify whether the expected outcome of functions are correct eliminating programmer errors. Secondly, although not as applicable to this project, if a part of the code is thoroughly tested, even if another developer changes something in the code, the tests can ensure what has changed doesn't break other parts of the system.

For the back-end system, a Python package: Pytest was used for testing. These tests focused on two main aspects. Firstly, endpoint testing. Tests which mock a call to an endpoint, ensuring the expected behaviour occurs. This includes both successful and unsuccessful calls. Both valid and expired tokens are tested to ensure the auth handler behaves appropriately, also improving security. Secondly, database testing. Tests which ensure the SQL queries are correct. To maintain data integrity for the application's live database, a cloned database was used for testing. This also provides more control over the data, as even if something went wrong during testing the database could be restored without affecting the live data.

The front-end unit testing is more complex. Due to the asynchronous nature of the project and the dependencies on the client's system, a more powerful testing solution was required. Therefore, Jasmine was used as the BBD framework (behaviour-driven development) for writing the test cases and Karma as the test runner, capable of running the unit tests across multiple browsers.

Isolation is a major element of unit testing. Testing components separate from other sections of the code base ensures any failures or issues found during test runs can be identified to a specific component. Thus, allowing easier debugging and a more robust development process. Therefore, each component within the front end contains a testing file.

Each testing file must be set up appropriately, using a testbed. This testbed must provide the testing suite with all of the component's dependencies. For instance, say a component uses the authentication service making a call to isAuthenticated, the test bed would need a mocked authentication service and a spy on isAuthenticated to mock the return value.

Figure 49 - Example unit-test using fakeAsync

Most of the components utilise at least some asynchronous functionality, this causes another issue with testing. Before comparing the result of the test with the expected outcome, the test must be told to wait for the return value. *[figure 49]* shows one such test. Its purpose is to ensure the user is directed to the dashboard after a successful GitHub connection. This functionality depends on the code URL parameter and it being processed during component initialisation. Therefore, the test is set up with fakeAsync and the tick function allows us to control the simulated time.

5.2 Manual Testing

The other aspect of application testing is manual testing. Writing hundreds of unit tests is great for ensuring robust code but this can't test the usability of the application. Throughout the development of this application, I would perform manual tests as a user of the application. This might have been following the registration process or connecting my GitHub account. This testing helped to inspire both new features and improvements. The project's agile approach meant it was well suited to make these improvements. One notable improvement was the inclusion of the loading component. Through manual testing, it became obvious that a visual indication of the application loading data was needed, which, once implemented, greatly increased usability.

5.3 Testing Results

As expected, ensuring correct testing procedures and practices produced higher-quality code and a more robust development process. The unit tests were able to prevent new development from breaking older code, as before committing code to the repository I ran the tests to ensure none had been broken. Furthermore, Karma with its inbuilt test report, analyses the code to ensure it is tested thoroughly. Allowing myself to see which functions or sections of code hadn't been tested, and therefore improve the test coverage.

This project's solution is not without its flaws. Although good test coverage was achieved and proved beneficial during development, testing could still yet be improved with end-to-end testing. This type of testing would remove the notion of mocking API calls, testing the actual communication between the back and front end. Furthermore, utilising actual application testers, and mocking the interaction of real users would also have been beneficial. Being the test user and the developer of the application provides me with the unique advantage of

knowing exactly how the application works. Unfortunately, due to this project's time constraints, this wasn't a possibility.

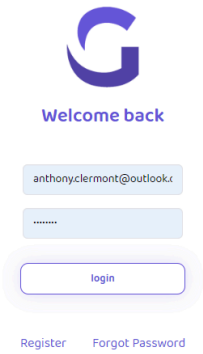
6. Results and Evaluation

This chapter will summarise the project's outcome, explore the completed application and evaluate the project's aim and objectives.

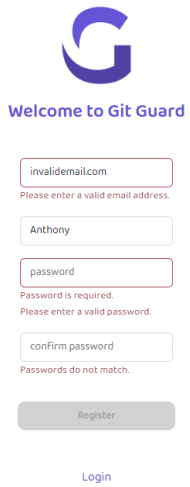
6.1 System Walkthrough

This section will show the process of using the system. From user registration, connecting a GitHub account and analysing a repository.

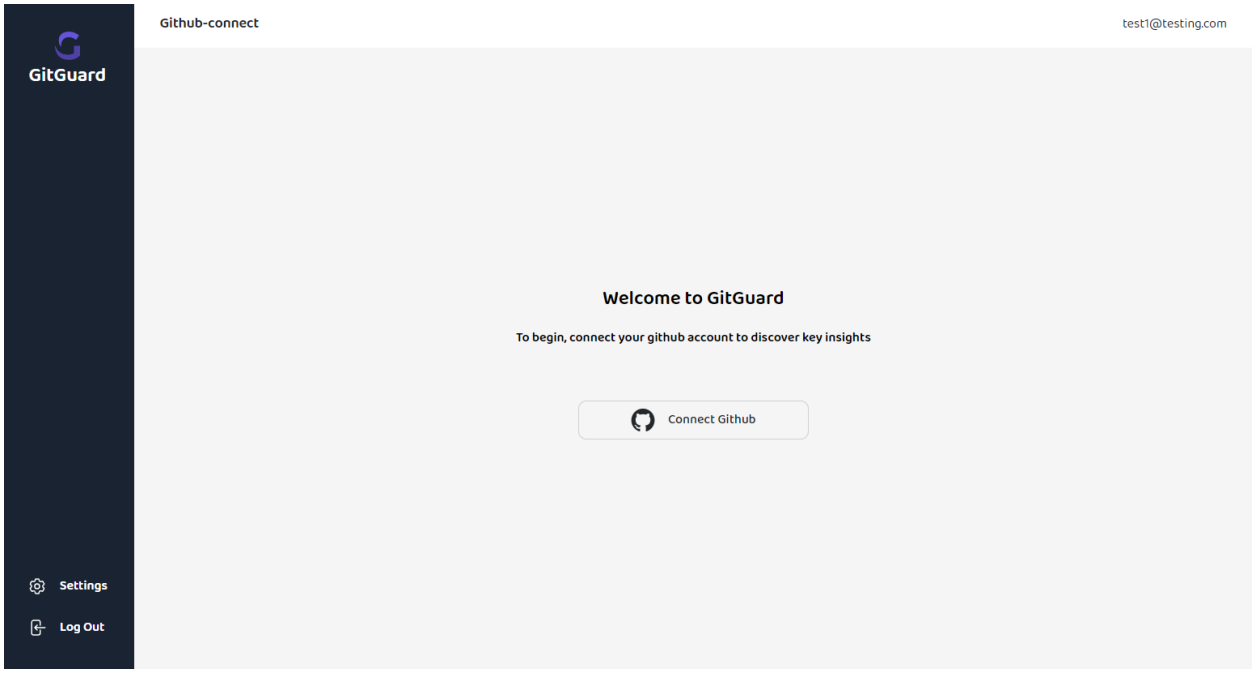
Login Page - This is the default root for an unauthenticated user accessing the application, the only routes which can be accessed are this page and the registration page.



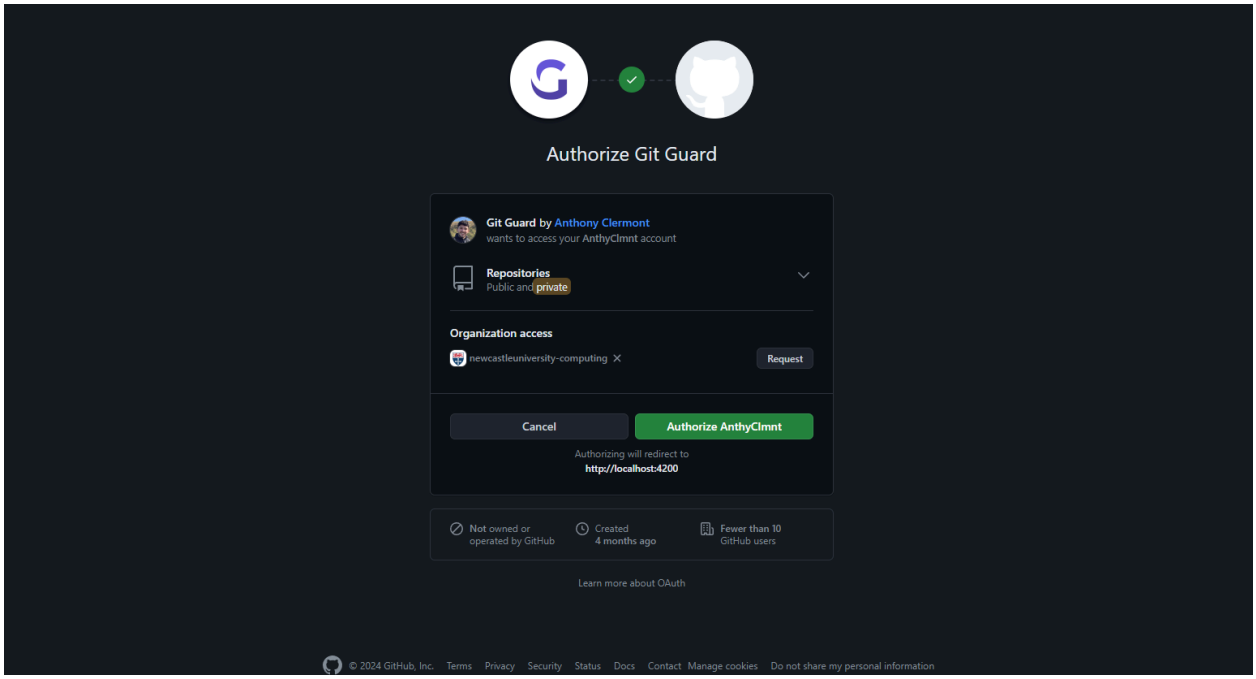
Registration Page - In the screenshot we can see the error validation for form fields, and the disabled login button while the form is not valid



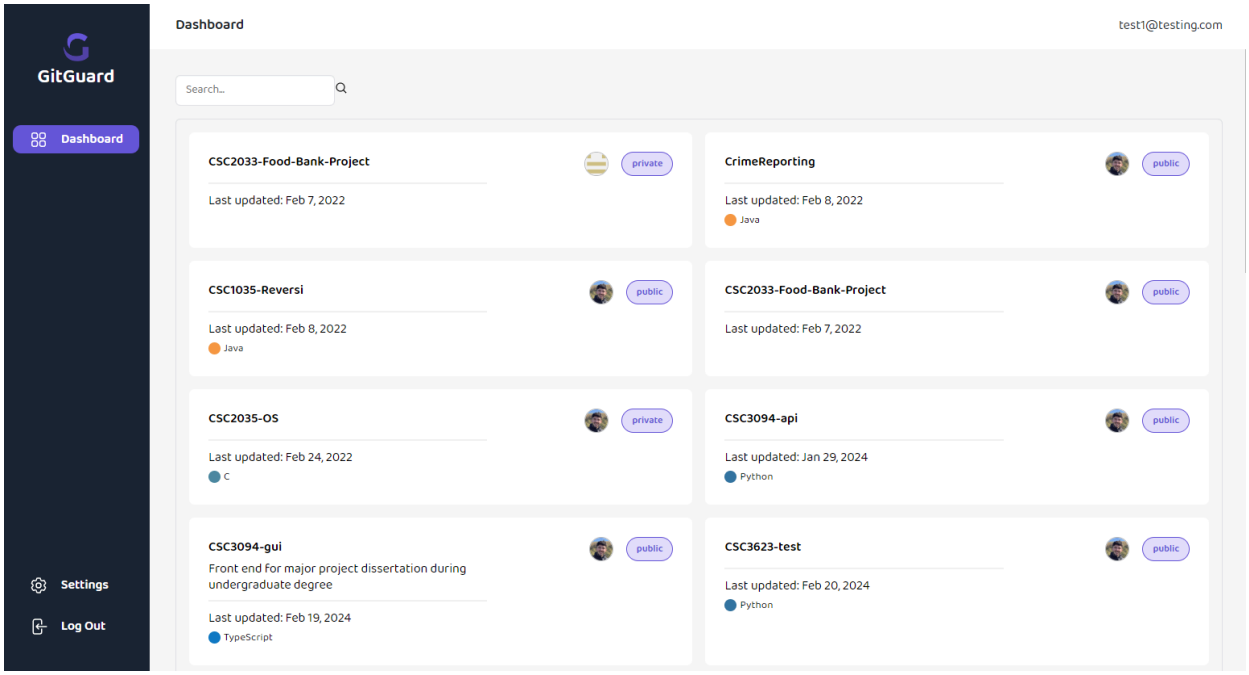
Connect GitHub - If a logged-in user has not connected their GitHub account, they will be directed to the connect page. All other routes are restricted when a user has not got a connected Github account.



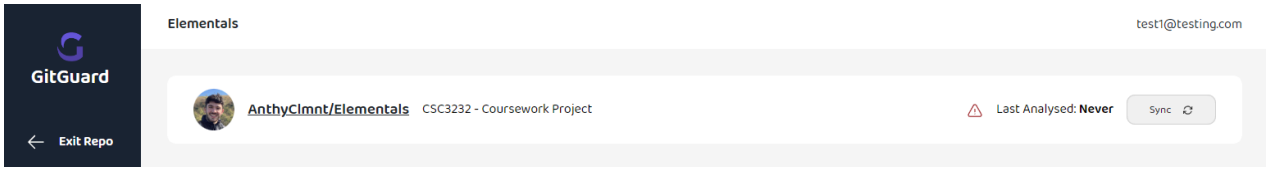
GitHub's Connection Page - Once the connect button has been clicked, the user authorises the connection before being returned to the application callback route.



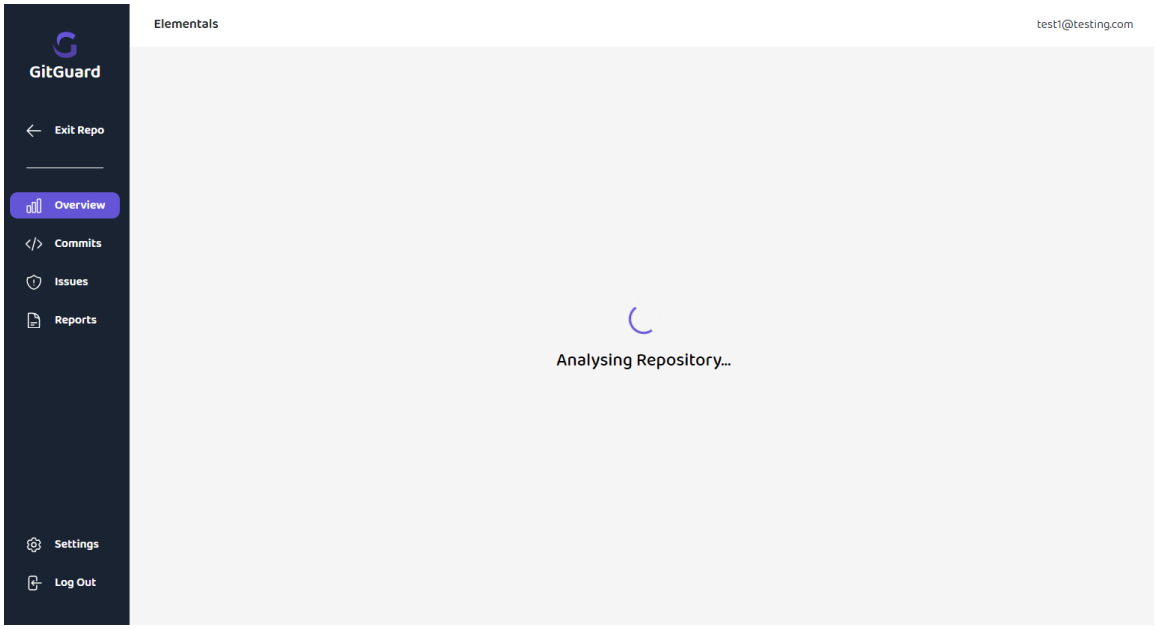
Dashboard - Once the callback route temporary code has been resolved, the user is directed to the dashboard where the GitHub connection is used to gather all of the user's repositories.



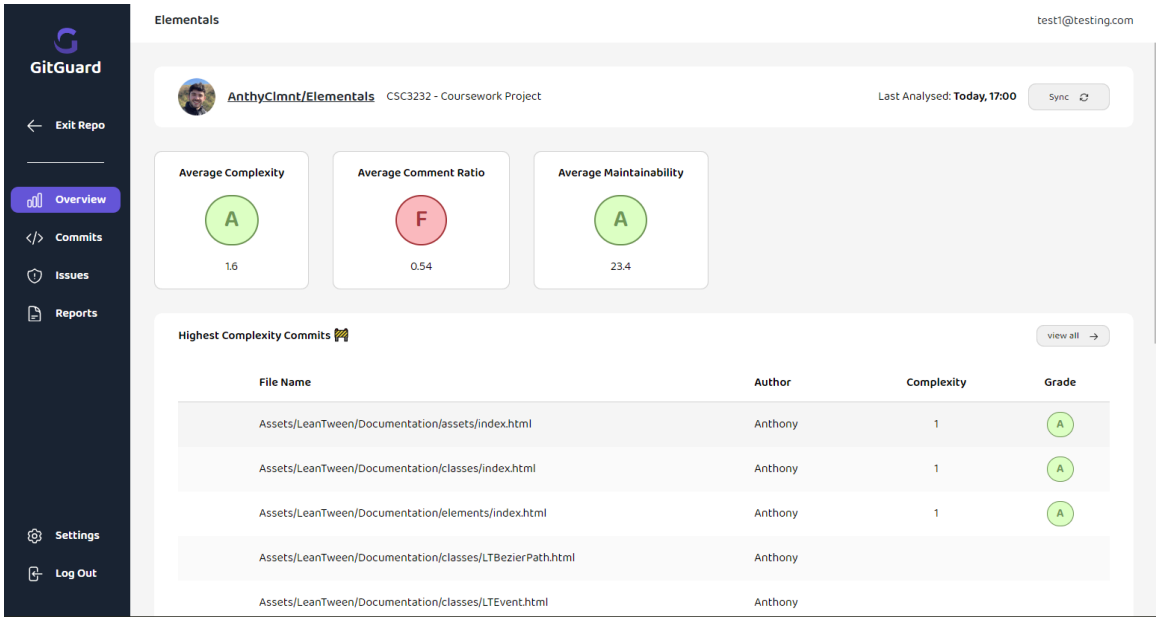
Syncing a Repository - Once a repository is selected from the dashboard grid, it will take the user inside the repository. The navigation routes change, providing repository specific routes, and seen in this image, a way to navigate out of the selected repository. This chosen repository has never been analysed, this can be seen on the right of the repository overview information. Clicking the sync button begins the process.



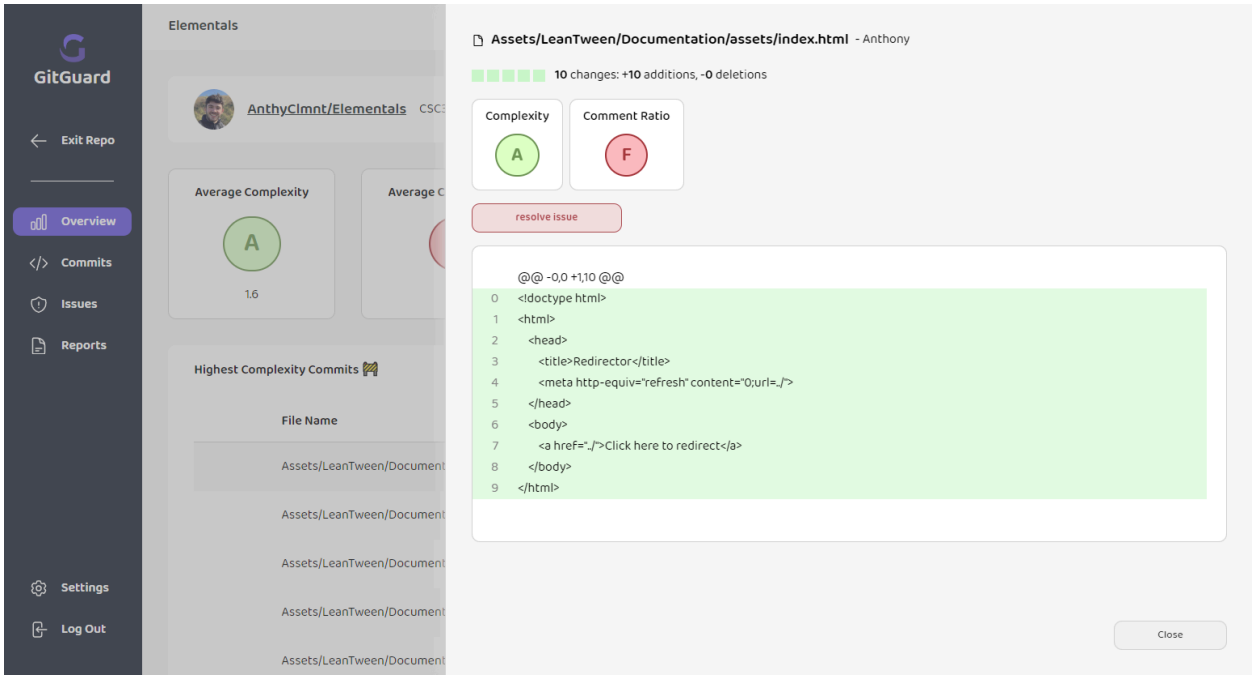
Loading State - Example of the loading state in action, here it is used to show that the chosen repository is being analysed.



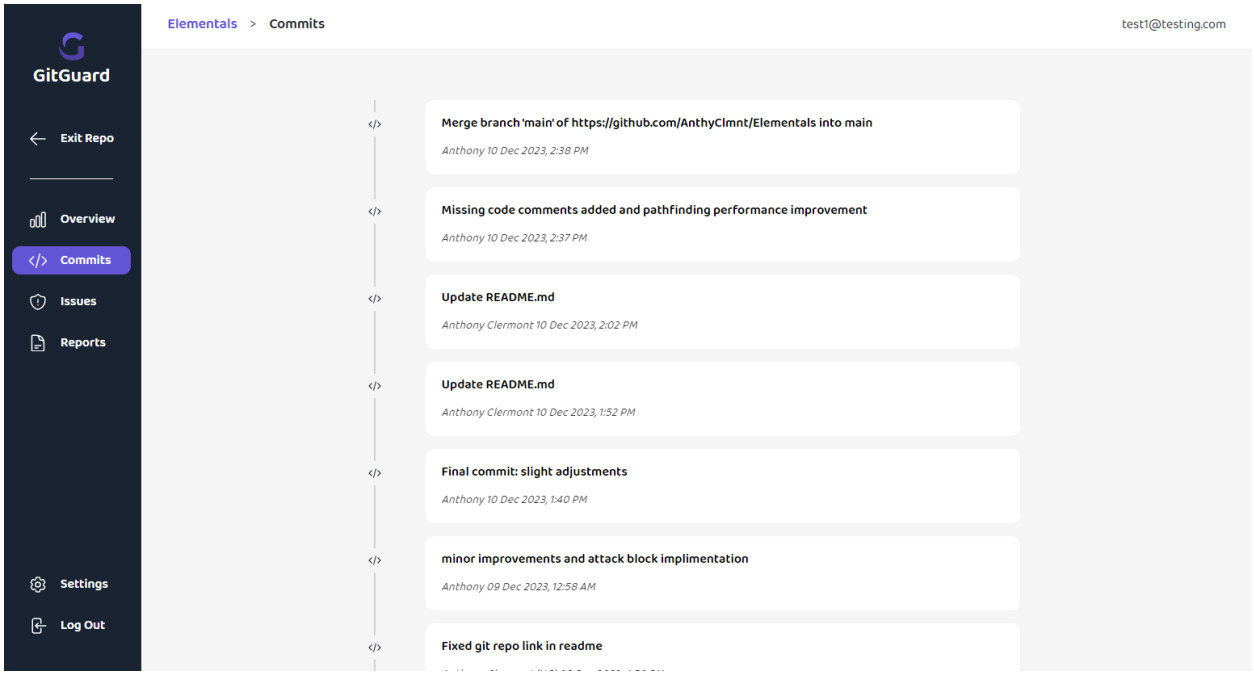
Repository Overview - Once the analysis has finished, the information is returned and displayed to the user. Providing overviews of the analysis as well as specific issues for each metric.



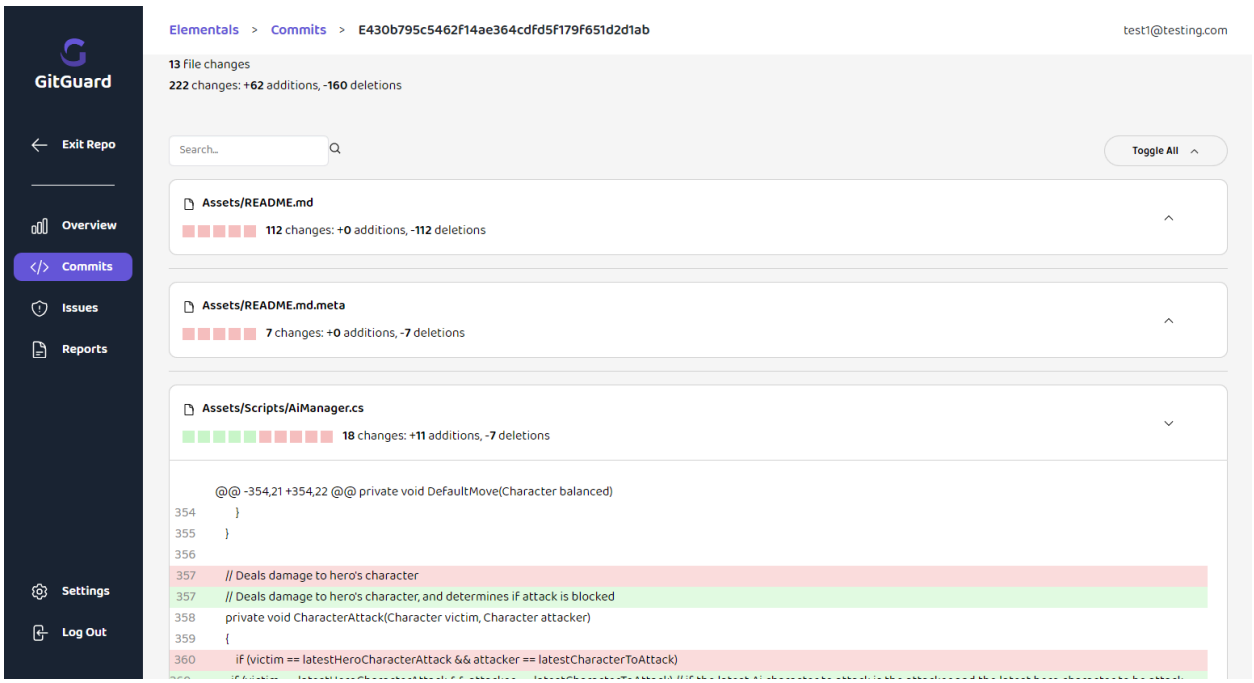
Side Draw - Clicking on a specific issue will open the application's sidebar, showing the actual code which has been flagged. In this example, we can see the comment ratio has been flagged as a major issue. The sidebar increases the applications usability, allowing salient information to be found more easily.



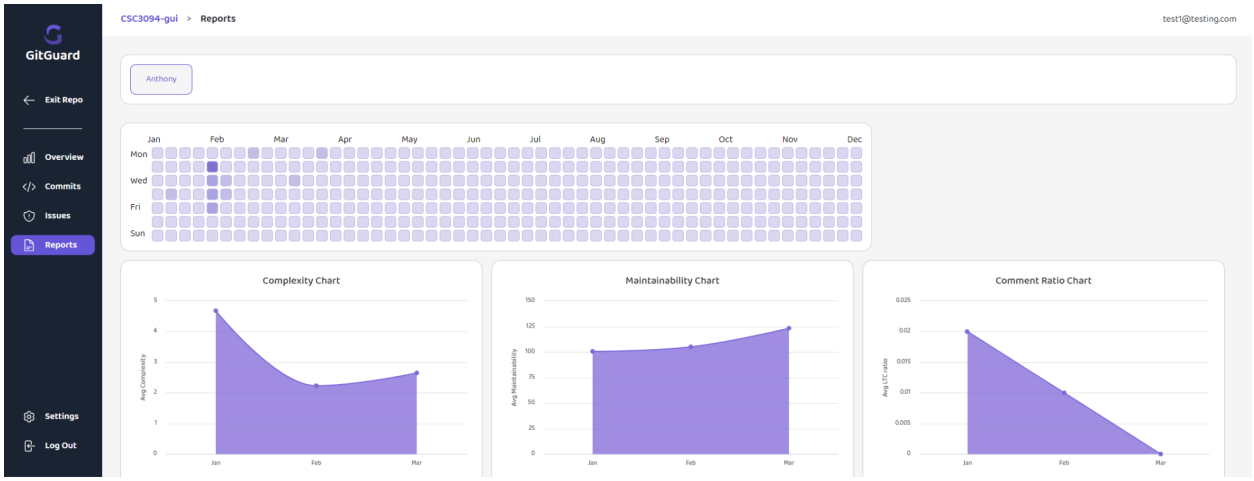
Repository Commits - A user may also see each commit within the repository.



Commit Details - Clicking on a commit shows the changes within the chosen commit. Syntax highlights has been used to show lines added or removed.



Contributor Report - clicking on a contributor will generate the chosen contributor's report data, using the analysis to show the contributor's progression or regression over time.



6.2 System Review

To provide a comprehensive review of the system, it is essential to offer thorough analysis of the results obtained from development. These results serve as evidence which support the success of the previously discussed milestones. However, it is crucial to acknowledge the limitations during the evaluation process, ensuring a nuanced understanding of the system's strengths and weaknesses.

The evaluation of the system's back end performance encompasses several key metrics, including response times, and resource utilisation. Through testing and benchmarking, it was observed that the implementation of FastAPI for the backend contributed to the system's high responsiveness, with minimal latency. However, it is important to note that the evaluation was conducted under controlled conditions, and real-world performance may vary due to factors such as network latency and hardware constraints.

In assessing the usability of the application, manual testing played a crucial role. The intuitive design of the user interface, powered by Angular, encouraged ease of navigation and accessibility. Nevertheless, limitations have been identified, particularly in terms of accessibility for users with disabilities. Partially because of the strong reliance on colour for the code analysis grading, the usability for colour-blind users would be considerably impacted.

Furthermore, to extend the review to the system's stability and reliability, focusing on potential performance bottlenecks. While the use of SQLite3 provided a lightweight solution, which was chosen due to the project’s time constraints, the potential limitations flagged during research were observed. These limitations would pose challenges during deployment due to its limited concurrent access capabilities which restricts FastApi’s performance benefits, necessitating an alternative database solutions in future iterations, PostgresSQL for instance..

One notable success of the project lies in the robust implementation of the authentication system, leveraging JSON Web Tokens for secure user authentication. The use of JWTs enhanced system security and provided a seamless authentication experience for users. However, recalling the technical constraints which arose, implementing HTML-only cookies for JWTs wasn't possible. Consequently, storing JWTs in local and session storage leaves the system vulnerable to potential Cross-Site Request Forgery (CSRF) attacks. Hence, while JWTs mitigate many security risks, such as unauthorised access, the system requires additional measures to fully address potential CSRF vulnerabilities.

6.3 Fulfilment of Objectives

The project's aim was to develop a web-based code analysis tool, utilising static code analysis to determine contributors' code quality and other performance-based metrics. This aim was deconstructed into seven key aims. In this section, I will discuss the outcome of each objective and evaluate its success.

1. ***Investigate and analyse potential solutions to determining code quality based on published research***
   The first objective was fulfilled during the research stage of this project. Identifying multiple viable ways to determine code quality. With the project deciding on a formula-based approach, further research identified which formula could determine quality the best from the opinions of developers themselves. Therefore I can conclude a successful outcome of this objective.

2. ***Research and identify the most suitable technology stack for development and implementation of this web tool***
   The second objective was also fulfilled during the research stage. Although backed by research, the majority of decisions for the technology stack were made through my own experience. Retrospectively, the chosen technology stack indeed proved to be a good choice. Fast-api and Python processed requests quickly and the Angular front-end allowed for robust and easy development. Furthermore, the original decision to use SQLite3, the lightweight database alternative, proved more than powerful enough for this project's requirements. All of which collectively result in a successful outcome.

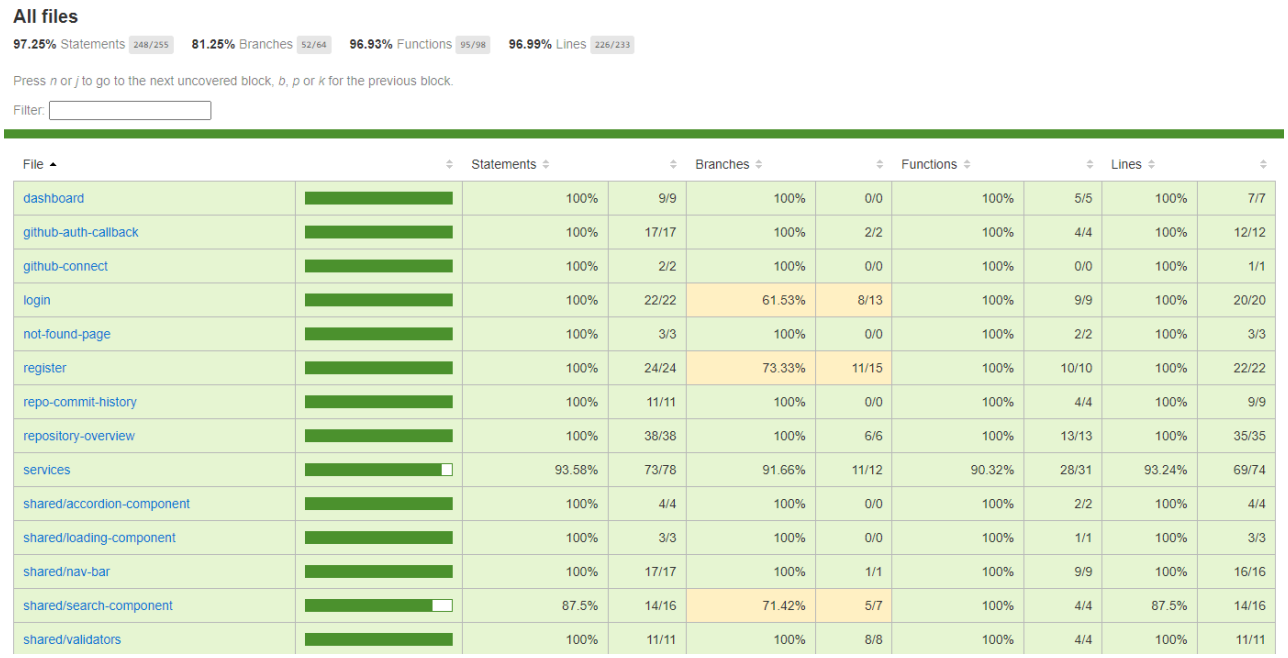
3. ***Adhere to agile methodologies specifically Scrum, for an optimal software engineering process***
   This objective ran throughout the project. And as the project evolved, so did this objective. An agile approach was utilised during development, with a continuous cycle of design, development, testing and review. However, Scrum was not used as the agile framework in the end with Kanban being preferred as a result of the project's research. This may suggest the failure of this objective, although, I would argue the modification improved the planning and management of the project, due to Kanban being much more well-suited for a sole development project.

4. ***To become familiar with GitHub's API, including the authentication process and API calls for relevant project data***
   This objective was fulfilled mainly during the early stages of development. It is a personal objective yet still crucial to the success of the project. The outcome has been a resounding success. I have not only grown accustomed to the authentication process but also to general interaction with the API. My understanding of GitHub's API enabled development to progress and complications to be resolved. Most notably, recalling chapter 4.12 Sync Repository Api's, where knowing a since date can be passed as a parameter, greatly improved application performance.

5. ***Ensure industry-standard security principles are followed, including hashing/encryption of sensitive information, an appropriate authentication flow and correct authorisation controls to restrict unauthorised access***
   This objective was fulfilled during both the research and development stages. Arguably this objective's outcome was the most successful. A huge amount of resources and time was used to implement correct authentication for both the back and front end of the application, with many of the development chapters discussing how security was improved. To address the actual objective, all of a correct authentication flow, hashing, encryption and decryption, and authorisation controls were implemented successfully.

6. **To utilise the findings from objective one and implement a suitable solution which is capable of determining contributor's code quality**
   This was the most challenging objective to achieve a successful outcome. Even with the success of objective one, the challenges faced were difficult to overcome; Supporting multiple languages was especially challenging. Although the project's implementation of this objective does warrant a successful outcome. This is by no means is a perfect solution. Ideally, with more time, the project's analysis could be greatly improved.

7. **To achieve a minimum of 80% test coverage across the application to comply with correct software engineering principles**
   The final objective ran throughout development, and the fulfilment of the objective was continuously upheld, with each new development accompanied by unit tests. Reflecting on the implementation of testing, proved arduous due to the number of tests which needed to be written. However, thankfully the project was able to implement tests whose coverage was well above the 80% aim set out in this objective, and therefore can be deemed a successful outcome *[Figure 50]*.



File	Statements		Branches		Functions		Lines	
dashboard	100%	9/9	100%	0/0	100%	5/5	100%	7/7
github-auth-callback	100%	17/17	100%	2/2	100%	4/4	100%	12/12
github-connect	100%	2/2	100%	0/0	100%	0/0	100%	1/1
login	100%	22/22	61.53%	8/13	100%	9/9	100%	20/20
not-found-page	100%	3/3	100%	0/0	100%	2/2	100%	3/3
register	100%	24/24	73.33%	11/15	100%	10/10	100%	22/22
repo-commit-history	100%	11/11	100%	0/0	100%	4/4	100%	9/9
repository-overview	100%	38/38	100%	6/6	100%	13/13	100%	35/35
services	93.58%	73/78	91.66%	11/12	90.32%	28/31	93.24%	69/74
shared/accordion-component	100%	4/4	100%	0/0	100%	2/2	100%	4/4
shared/loading-component	100%	3/3	100%	0/0	100%	1/1	100%	3/3
shared/nav-bar	100%	17/17	100%	1/1	100%	9/9	100%	16/16
shared/search-component	87.5%	14/16	71.42%	5/7	100%	4/4	87.5%	14/16
shared/validators	100%	11/11	100%	8/8	100%	4/4	100%	11/11

Figure 50 - Unit test code coverage report

7. Conclusions

This chapter will conclude the outcome of the project, from both a personal perspective and future work which could be carried out to improve the application.

7.1 Personal Development

Besides the accomplishments of the project itself, my development has progressed more than I could have hoped, both from a technical and personal perspective. This subchapter will discuss and reflect on these developments.

Firstly, from a technical perspective, although I have industry experience working within a software engineering team, I have learnt and now understand the complexities of a software engineering project. My experience was maintaining and improving existing software, the difficulties of making new software proved much more challenging. With existing software, the technology stack has been decided, and the structure of the code base and user flow has already been implemented. An authentication system, database system and connection are also implemented. All these aspects of development lay the groundwork for developing new or improving existing features, overcoming these issues highlighted much of the work I took for granted during my placement.

This also applies from a planning and project overview perspective. These systems were already implemented during my placement, tried and tested. Implementing such systems: a Trello board or commit protocols, for instance, all had to be decided on and implemented myself. Notably, although I've only highlighted the challenges of these processes, they have provided me with the

opportunity to understand how to begin a project, build an application from the ground up and the correct procedures in order to achieve this.

Although my prior experience with Fast-api, this knowledge was limited. Using this as the back-end's framework has allowed me to improve my understanding of the basics: request parameters, handling errors and model based declaration for instance, but also, begin to learn the full power of the framework utilising advanced features. This included implementing a custom built authentication method, route protection, concurrent request handling, and testing both the database system and API endpoints. This new knowledge and experience have provided me with the confidence to use this framework in the future, and I would relish the opportunity to be able to learn more, in the hope of becoming proficient with this framework.

This also applies to front-end development. The main framework I used during my placement was also Angular. For this reason, I was already more confident using this technology. Yet this doesn't mean I have not learned anything due to this project. Much of my front-end development skills have progressed further. For starters, handwriting components such as the accordion, the search component and the sidedraw were all elements implemented using pure HTML, CSS and Typescript, rather than utilising a UI library, kendoUI for example during my placement. Understanding the logic behind these components taught me the complexities of what previously I would have just used as a pre-build component. Furthermore, this approach maximised my understanding by providing the most freedom possible for styling and functionality.

Also, a major element of using Angular is understanding RxJS, and though I've previously used this package, my understanding was basic. This project has allowed me to explore varying solutions and methods, including pipe, and tap to alter the output of the emitted observable or switchmap to combine the subscriptions of two observables, to implement asynchronous code. These concepts and terminology can be applied to alternative frameworks and will benefit my engineering ability in future work regardless of the technology stack.

Finally, to evaluate my progression. This project has provided me with the freedom to explore and develop my skills in the areas I am most passionate about. This freedom caused mistakes, and in some way made this project more difficult, but from these mistakes is where I learnt the most, where I have homed in and improved my understanding of these technologies. Furthermore, I've been able to successfully manage and plan a full-scale application, which was capable of the original aim and objectives, and I'm extremely proud of the project's outcome.

7.2 Future Work

This final section will explore and discuss possible future developments, and how the project could be continued and progressed further.

Firstly, and most notably, the inclusion of artificial intelligence. As the project's research emphasised the alternative approach to determining code quality was using machine learning. I originally decided against due to time constraints on this project, but future work could greatly improve the quality and usefulness of the analysis. I would propose this project incorporates the raw metrics calculated by the formulas, to pass into a neural network, capable of providing meaningful and useful insights into contributors' code quality.

A feature which this project intended to implement but unfortunately couldn't due to time constraints was the use of role-based authorisation. The current application only utilises authentication, anyone authenticated can see their repositories. The issue is that only repository owners can see these repositories. For example: say a development team is working on a project, the repository owner could see other contributors' code quality but contributors themselves can't view their own performance. Role-based authorisation could be implemented which provides owner or contributor-level access. Owners could invite contributors to the repository team, they then would be able to view their performance but restricted to not see the entire team's performance. This role-based authorisation could be implemented with the use of the JSON web tokens, including an additional field could be included within the payload containing the user's role, this could then restrict the views and actions available to contributors. This possible work would introduce a new aspect to the application where not only individuals can benefit from the analysis but entire engineering teams.

Finally, this project intended to incorporate correct and proper software engineering practices. Although, for the most part, this has been a resounding success, a crucial aspect of any software development project is deployment. Thus far, this application has only been hosted locally. Future work could be carried out to deploy the application's systems. This includes the database, hosted with Amazon AWS or Microsoft Azure for instance, as well as, the back and front-end applications deployed with an Azure Container and GitHub pages respectively for instance. More work would also need to be carried out for a deployed application to run correctly. For example, the GitHub OAuth application would need to be reconfigured to include the URL of the live callback route, environment variables would also need to be configured for the API, to include all of the project's secret keys, and the API's CORS policy would need to be reconfigured to only accept requests from the location of the deployed front-end. These are just a few steps which are crucial to deploy the application and I'm sure if this work was carried out many more obstacles would need to be overcome.

END

References

Dunka, B., Emmanuel, E.A. and Oyerinde, Y. (2018) Simplifying Web Application Development Using-Mean Stack Technologies, ieeexplore. Available at: https://www.researchgate.net/publication/322821888_Simplifying_Web_Application_Development_Using-Mean_Stack_Technologies (Accessed: 11 March 2024).

Li, Y. and Manoharan, S. (2013) A performance comparison of SQL and NoSQL databases, ieeexplore. Available at: https://doi.org/10.1109/PACRIM.2013.6625441 (Accessed: 11 March 2024).

Singh, A. (2023) Fastapi vs. Django Rest Framework: A comprehensive comparison, Medium. Available at: https://medium.com/@abhisheksingh_50175/fastapi-vs-django-rest-framework-a-comprehensive-comparison-c55139254b7d (Accessed: 11 March 2024).

Noureddine, A.A. and Damodaran, M. (2008) Security in web 2.0 application development: Proceedings of the 10th International Conference on Information Integration and web-based Applications & Services, ACM Conferences. Available at: https://dl.acm.org/doi/pdf/10.1145/1497308.1497443 (Accessed: 11 March 2024).

Ahmed, S. and Mahmood, Q., 2019. An authentication based scheme for applications using JSON web token. In: 22nd International Multitopic Conference (INMIC), Islamabad, Pakistan, 2019. pp. 1-6. Available at: https://doi.org/10.1109/INMIC48123.2019.9022766 (Accessed: 11 March 2024).

Bouganim, L. and Guo, Y. (2011) Database encryption, SpringerLink. Available at: https://doi.org/10.1007/978-1-4419-5906-5_677 (Accessed: 12 March 2024).

Hélie, J., Wright, I. and Ziegler, A., (2018). Measuring software development productivity: A machine learning approach. In Proceedings of the Conference on Machine Learning for Programming Workshop, affiliated with FLoC (Vol. 18, p. 11). (Accessed: 13 March 2024).

Kukreja, N. (2015) Measuring software maintainability, Quandary Peak Research. Available at: https://quandarypeak.com/2015/02/measuring-software-maintainability/ (Accessed: 13 March 2024).

Börstler, J., Bennin, K.E., Hooshangi, S. et al. (2023) Developers talking about code quality - Empir Software Eng 28, 128. Available at: https://doi.org/10.1007/s10664-023-10381-0 (Accessed: 13 March 2024).

Kaur, K. and Jajoo, A. (2015) Applying Agile Methodologies in Industry Projects: Benefits and Challenges, ieeexplore. Available at: https://doi.org/10.1109/ICCUBEA.2015.166 (Accessed: 28 March 2024).

Sachdeva, S. (2016) Scrum Methodology, International Journal of Engineering and Computer Science. International Journal Of Engineering And Computer Science ISSN: 2319-7242 Volume 5 Issues 6 June 2016, Page No. 16792-16799. Available at: https://fatcat.wiki/file/3xshemd26benrdjmj5zvrtv3ji (Accessed: 28 March 2024).

Kirovska, N. and Koceski, S. (2015) Usage of Kanban methodology at software development teams, Journal of Applied Economics and Business (JAEB). Available at: https://www.aebjournal.org/article030302.php (Accessed: 28 March 2024).

Barševska, Z. and Rakele, O. (2019) Interaction of color, Interaction of Color. Available at: https://dukonference.lv/files/978-9984-14-901-1_61_konf_kraj_C_Hum%20zin.pdf#page=79 (Accessed: 29 March 2024).