School of Computing

CS3219

Software Engineering Principles and Patterns

Project Report: PeerProgram

Group 33

Student Name	Teh Hao Rui Marcus	Chia Wei Hao David	Chua Hua Lun	Bruce Ong
Matriculation Number	A0182338B	A0183642A	A0185471Y	A0182333L

Code repository link:
https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-project-ay2122-2122-s1-g33
Website link:
https://onlyduh.com

1. Background and purpose of project

Today, cloud-based hosting services such as Github allow development teams to work on the same code base asynchronously. However, such systems do not support real-time code collaboration. In some cases, such real-time collaboration can be more effective in writing code as team members need to discuss and make changes on the spot. A platform that allows multiple users to edit the same piece of code at the same time and execute it quickly without needing any set up in each users' machine would be useful. Furthermore, the emphasis we place on being able to share code in real-time with multiple users conveniently allows our tool to be used for educational purposes as well. (via links)

Purpose

- Users can write and run their code online together with other users in real time.
- Users can share their code with others via a link. Anyone with the link can access the code and make changes to it.

2. Individual contributions

Contributions	Technical	Non-technical
Teh Hao Rui Marcus	- Set up Kubernetes on GCP - Implement Sessions Service - Frontend and Sessions Service Integration	- Requirements documents - Project report
Chia Wei Hao David	- Implement frontend and Docs microservice - Frontend and Sessions Service Integration	- Webapp design - Requirements Documentation - Project report
Chua Hua Lun	- Unit Testing - Functional Testing - Setup CI/CD	- Requirements Documentation - Project report
Bruce Ong	- Implement Code executor microservice - Kubernetes load-balancing + auto-scaling for code executor	- Requirements Documentation - Project report

3. Functional Requirements

3.1 User Management

ID	Functional Requirement	Priority	Timeline
UM-1	The users will be able to sign up/login/logout using their Github account.	Low	Week 7
UM-2	The users will be able to sign up/login/logout using their email account.	High	Week 7
UM-4	The users will be able to save the docs they created and shared.	High	Week 8

3.2 Session System

ID	Functional Requirement	Priority	Timeline
SS-1	Users should be able to join the same session with a code editor	High	Week 7
SS-2	Code edits in a session should be observed in real time by all users in the session	High	Week 7
SS-3	The system should show the output after the code has been executed.	High	Week 7

SS-4	The Session System should start a new session linked to a particular user document if a user opens a document.	High	Week 7
SS-5	The Session system should delete an existing session if there are no users currently accessing it.	High	Week 8
SS-6	The system should block the users from editing when the code is being executed.	Medium	Week 8

3.3 Frontend Client

ID	Functional Requirement	Priority	Timeline
FC-1	The application client will provide a UI for users to enter in signup/login information	High	Week 7
FC-3	In each session, the client should display the results of each code execution	High	Week 7
FC-4	When logged in, the application client should provide a dashboard for users to view the share link for each of their docs	High	Week 7
FC-2	When logged in, the application client should provide a dashboard for users to view a list of their created docs	Medium	Week 8
FC-5	In each session, the client should show	Low	Week 8

	syntax highlighting in the editor.		
FC-6	When not logged in, the application UI consists of a landing page to introduce new users to the purpose of the application	Low	Week 9

3.4 Code Executor

ID	Functional Requirement	Priority	Timeline
CE-1	The code executor microservice should be able to take in a code file and run it, sending back the output.	High	Week 7
CE-2	The code executor should be able to execute Python code.	High	Week 7
CE-3	The code executor should run code in separate containers.	Medium	Week 8
CE-4	The code executor microservice will time out after 10 seconds of not receiving an output.	Low	Week 9

4. Non-Functional Requirements

Our NFRs are listed in order of priority:

1. Security
2. Performance
3. Availability
4. Scalability

We prioritize security first because we believe there is a need to set a strong foundation for security from the beginning. Performance comes next because it is important for the user to have a good user experience when using our platform. We need to ensure there is minimal delay during the game process as the game is time-sensitive. We placed availability in third place because we don't foresee a need to support other less popular browsers/operating systems in the foreseeable future. Finally, we prioritize scalability last because we do not expect to have high traffic during our early stages of deployment.

4.1 Security

ID	Non-Functional Requirement
S-1	All microservices will only allow relevant IPs addresses (e.g frontend) to access the GCP instance (backend)
S-2	All microservices will be configured such that only relevant backend services should be able to access and make modifications on the databases
S-3	We utilize Auth0 to handle user authentication and authorization, so that we can offload the burden of storing user passwords: Auth0 is a flexible, drop-in solution to add authentication and authorization services to your applications. Your team and organization can avoid the cost, time, and risk that come with building your own solution to authenticate and authorize users.
S-4	Code should only be executed within a container to prevent remote code execution cyber-attacks.

4.2 Performance Requirements

ID	Non-Functional Requirement
PR-1	The application will ensure at most 3 seconds response delay during live coding when there is a change in the content in the collaborative notepad.
PR-2	The application will not take more than 5 seconds to change from one screen to another.
PR-3	API calls from frontend to backend should have a response time of <3s during peak periods

4.3 Availability Requirements

ID	Non-Functional Requirement
AR-1	Ensure system is usable during peak hours, 12pm - 8pm
AR-2	The system will support the most browsers e.g. Chrome, Firefox, Edge
AR-3	The system will be available on both MacOS and Windows operating systems.

4.4 Scalability Requirements

ID	Non-Functional Requirement
SR-1	The system can host 100 concurrent sessions at any point in time
SR-2	The system can support up to 1000 users logged in at once
SR-3	The system can run different code files within separate containers.

5. Architectural Design

5.1 Architecture Diagram

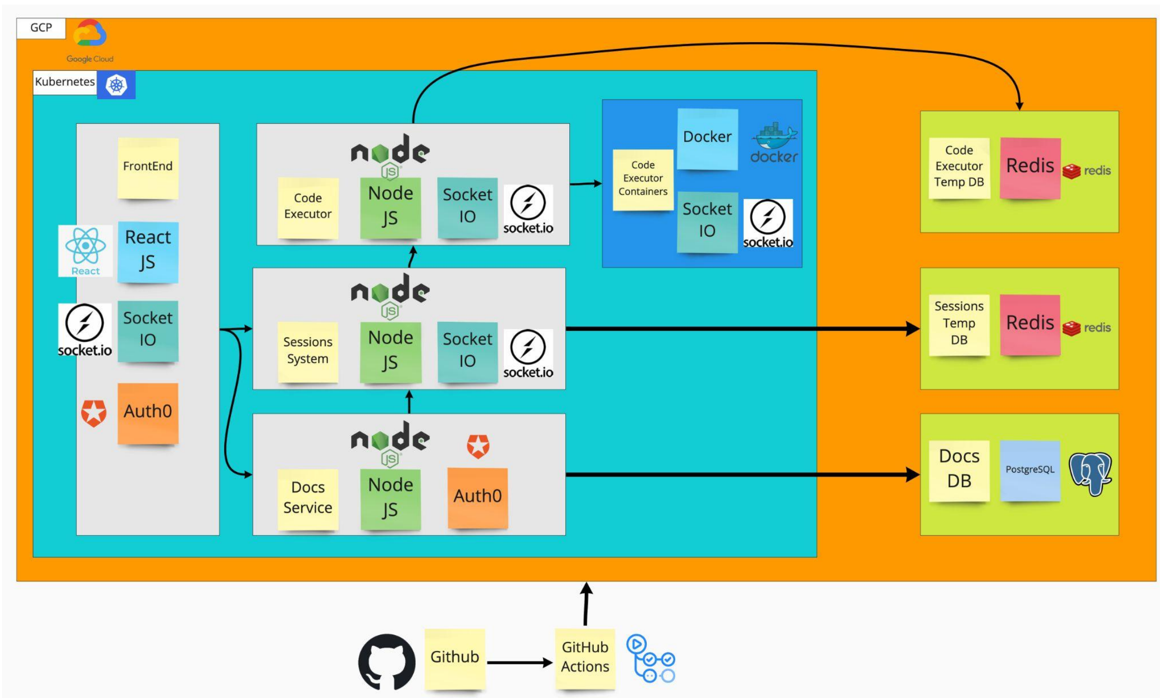

Figure 5.1.1: Architecture Diagram Of PeerProgram

We decided to use a microservices architecture consisting of 3 microservices: Code Executor, Sessions system and the Docs service. Each microservice is in charge of a single responsibility, the Sessions system handles user sessions, the Docs service handles code document-related operations with authentication, and the Code Executor handles the live execution of Python code.

The Kubernetes Ingress Gateway sits between our frontend and our microservices, performing load-balancing and horizontal pod auto-scaling between service pods as well as service discovery operations for the client.

Real-time communication between the frontend and the Sessions system is done via the use of websockets protocol, while microservices talk to each other using http protocols.

Each service instance is being launched in Docker container(s) from within a Kubernetes pod, with all replicas of a service being part of a single Kubernetes service. Using containerization and container-orchestration allows us to tap into benefits such as greater portability, improved security as well as high scalability.

5.2 Architectural Decisions

Architecture Decision: Microservice Architecture	
Concerns	1. Large codebase takes time to understand 2. Tightly-coupled components make it more constraint 3. Difficulty in horizontally-scaling monolithic application as the entire code base has to be replicated on other servers 4. Difficulty in scaling monolithic database
Decision	A Microservices Architecture allows us to subdivide a large application into loosely-coupled modules/services communicating via APIs. Each service can be deployed on Docker container(s) within a Kubernetes pod/replica. Kubernetes supports horizontal-scaling and load-balancing between pods/replicas. (each shipping only one service) This allows us to scale each individual service up and down independently of other services depending on traffic and spin up a suitable number of replicas for each individual service. (e.g. one particular service might need more pods for the application to run smoothly) Also, each member can develop and test services in isolation to other services, ensuring continuous

	integration and deployment of individual services.
Assumptions	None
Alternatives	Monolithic Architecture
Reasons	With a microservice architecture, each component is loosely coupled. This means that it can be easily maintained and tested. Since the Docs and Sessions services will have more traffic than Code Executor service, a microservice architecture will allow us to scale up Docs and Sessions microservices independently from the Code Executor microservices. Good for small teams. Each group can work on an individual component and does not affect others. Each service can be deployed on it's own making it easy to make changes without having to redeploy the whole application.

Architecture Decision: Redis for Caching	
Concerns	1. Slow return of results due to large data (slow API response) 2. As database scales, the response time is reduced 3. No user interaction session data stored greatly

	affects performance (slower)
Decision	The sessions and code executor database uses Redis Caching. The data are frequently accessed. As such, caching it will reduce lookups and improve performance. With caching, API calls executed will be faster.
Assumptions	The data cached is set up to improve speed.
Alternatives	Memcached
Reasons	Without a caching layer, the average latency for getting posts is roughly 100-300ms (for small datasets). By caching with Redis, this time is reduced to 10-100ms, ensuring minimum latency. Redis also supports persistence unlike memcached, with a point-in-time snapshot of all the datasets allowing the data to be restored on startup, in case of system failure.

Architecture Decision: Auth0 For Authorization	
Concerns	Security: 1. Taking in password input from user requires sanitation implementation in the frontend 2. Need to store user passwords with proper encryption

<table>
<tr><td>Decision</td><td>Use Auth0, a third party authorization and authentication service. More specifically, we utilised the Auth0 React SDK for the frontend service.

Auth0 provides a log in/sign up page that our application redirects to. Users that login will be redirected back to our app. This allows us to offload the following workload to Auth0:
1. The implementation of log in/sign up UI components
2. The implementation of password sanisation logic in the frontend
3. The implementation of user authentication logic and password requirements
4. The implementation of password encryption logic in the frontend
5. Storage of user’s encrypted password and user ID in the backend</td></tr>
<tr><td>Assumptions</td><td>None</td></tr>
<tr><td>Alternatives</td><td>Implementation and Storage of User Authentication</td></tr>
<tr><td>Reasons</td><td>Since we offload the aforementioned workload to Auth0, we are able to save on development time and man-hours while still being able to deliver user authorization requirements. [UM-1] [UM-2]

This results in lesser testing needed as we now do not need to build and deploy a Users microservice that accesses an additional database of user data.</td></tr>
</table>

Architecture Decision: Ingress Gateway	
Concerns	1. Security (Clients have direct access to microservice endpoints) 2. Manual redirection of traffic to pods 3. No HTTPs without Ingress
Decision	The Ingress gateway sits between the client and the microservices and is the single point of entry for the client to all microservices. By doing this, we obtain the following benefits: 1. Hiding our microservices (endpoints) from potential attackers 2. Monitor traffic to each of the microservices from a single place 3. Scaling up by adding new microservices or modifying existing microservices only requires us to change the ingress gateway configuration. The frontend code does not have to change as it only has to be aware of a single endpoint
Assumptions	None
Alternatives	Direct client-to-microservice communication
Reasons	We have moved the logic of directing requests to the appropriate backend/microservice from the client-side/frontend code to the Ingress gateway.

	This results in less coupling on the client-side as they are only aware of the ingress gateway and are no longer coupled to every microservice. (Even if microservice endpoints/servers change, client-side code does not change)

6. Design Patterns

6.1 SPA Pattern

We employed the SPA pattern for our frontend and Docs service:

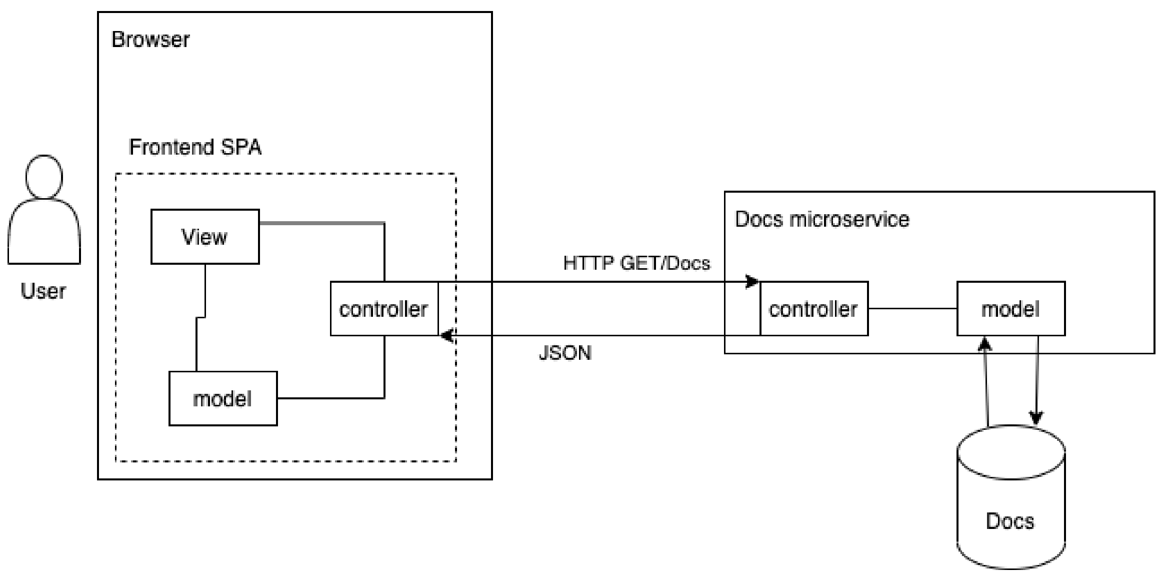

Figure 6.1.1: SPA Design Pattern Between Frontend and Docs Microservice

The client or frontend handles all the visuals presented to the user and the Docs service handles the model of the Doc data. The frontend will communicate with the Docs service via Restful API requests, and the Docs service has a controller that handles incoming requests from the frontend.

This is done so that we are able to separate concerns where the frontend service will only deal with interface components or views while the Docs service will contain the business logic and access to the database. This also allows the frontend to be more responsive and design oriented.

6.2 Pub-Sub Pattern

Real-time document editing is being implemented via the use of Socket.io, an implementation of the Pub-Sub pattern, in the sessions microservice. In Socket.io, the frontend and backend can emit and listen for certain events, much like publishing and subscribing to messages. Below is an example flow of the pub-sub pattern in our application:

1) A user publishes a message containing changes made to the document, without needing to be aware of how many users are listening/subscribing to changes for that document.
2) The sessions system will receive the message, and broadcasts that message to all users subscribed to that document
3) The users will receive the message and update their state and view of the document accordingly

7. Microservices

7.1 Docs

The Docs microservice handles creation and retrieval of code documents [UM-4] It is built with Express and Node.js.

It interacts with the frontend service by creating a new document each time the user clicks “Create” on the frontend interface, and returns a list of a user’s documents when the user signs in and loads the user dashboard.

It also interacts with the sessions service by patching a document’s text every 3 seconds from the last change in a session.

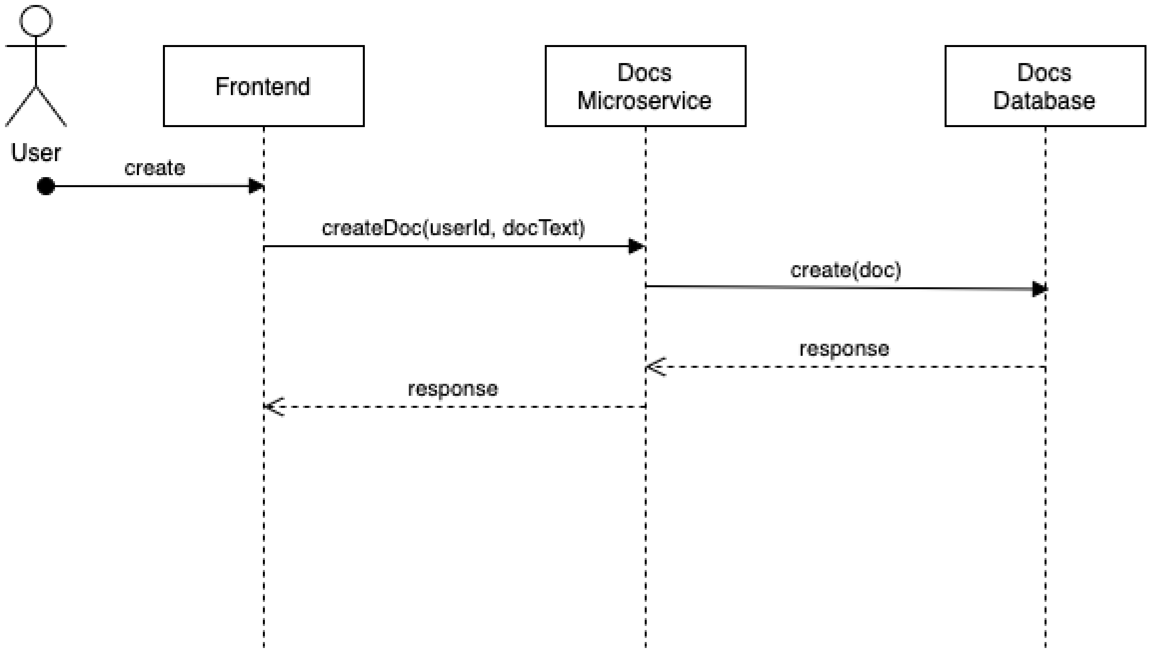

Figure 7.1.1: Sequence Diagram for Create Doc Interaction

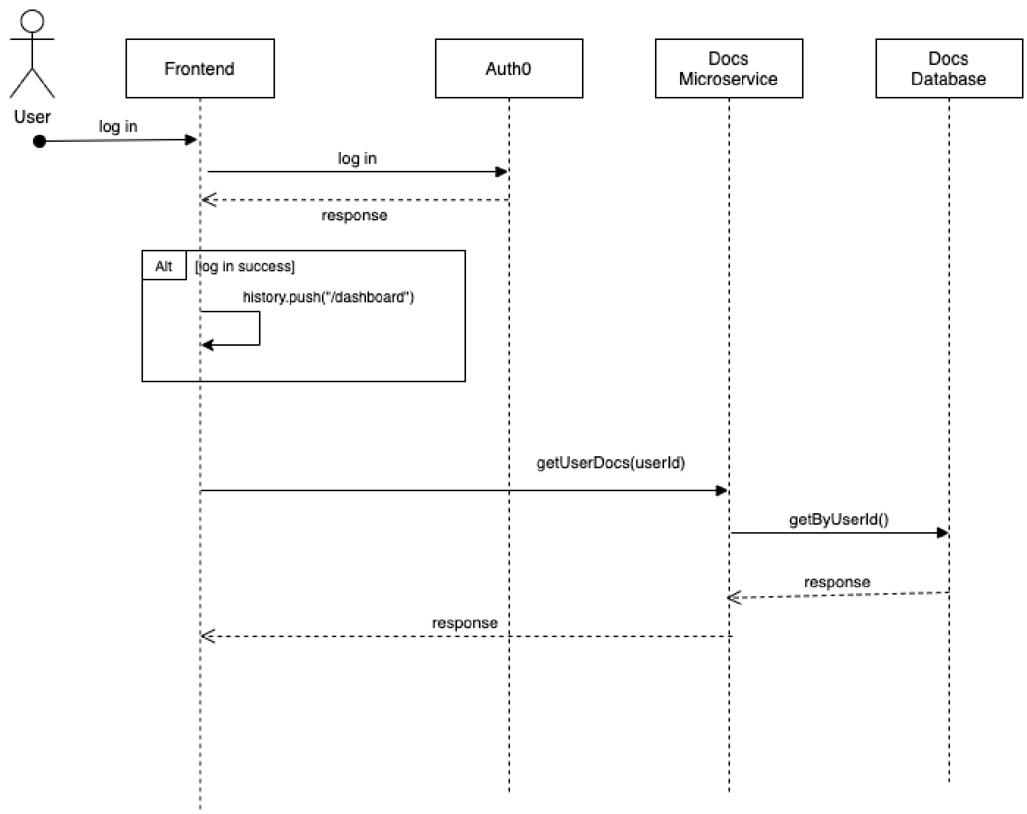

Figure 7.1.2: Sequence Diagram for Get User's Docs Interaction

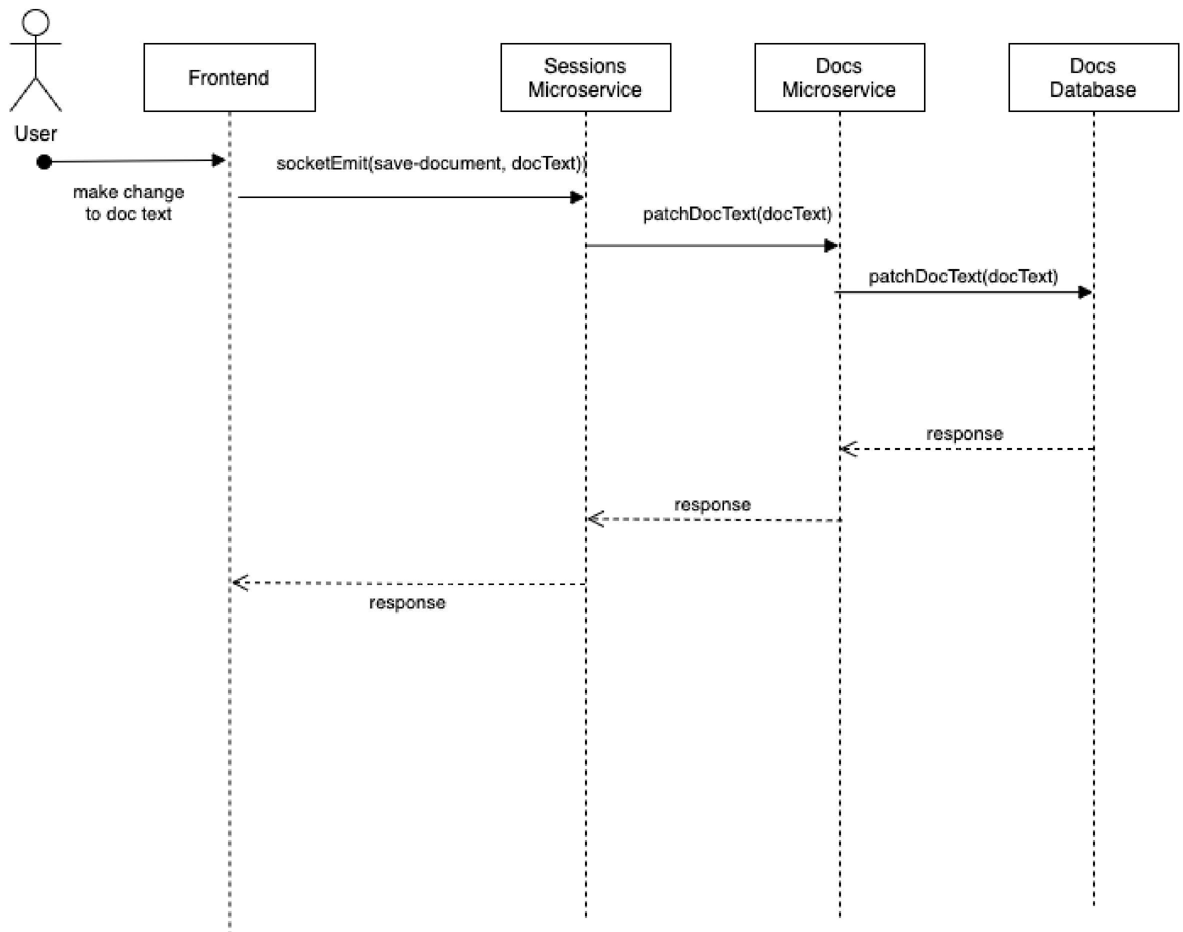

Figure 7.1.3: Sequence Diagram for On Doc Text Change Interaction

Supported Endpoints:

Create Doc

onlyduh.com/createDocs - HTTP POST Request

Creates a document resource

Sample success response: JSON

```
{
    "docId": "p46bss6m",
    "docTitle": "Doc Title 2",
    "userId": "12345",
    "docText": "code code  code code"
}
```

Get Doc By Doc Id

onlyduh.com/getDoc/:docId - HTTP GET Request

Retrieves the document given the doc id

Sample success response: JSON


```
{
    "docId": "hun3WpQ6",
    "docTitle": "Doc Program",
    "docText": "# type your python code here\nprint(1 + d)",
    "userId": "auth0|615bcb7ac69eb200704b4c2e"
}
```


Get Docs By User Id

onlyduh.com/getUserDocs/:userId - HTTP GET Request

Retrieves the all documents belonging to a specified user id

Sample success response: JSON


```
[
    {
        "docid": "hun3WpQ6",
        "doctitle": "Doc Program",
        "userid": "auth0|615bcb7ac69eb200704b4c2e",
        "doctext": "# type your python code here\nprint(1 + d)"
    },
    {
        "docid": "MTzoyss2",
        "doctitle": "Doc Program",
        "userid": "auth0|615bcb7ac69eb200704b4c2e",
        "doctext": "x = 1\ny = 2\nprint(x - y)"
    },
    {
        "docid": "usJ7b6yk",
        "doctitle": "Doc Program",
        "userid": "auth0|615bcb7ac69eb200704b4c2e",
```

```
        "doctext": "# type your python code here\nwhile True:\n    x = 1\n
"
    }
]
```

Patch Doc

onlyduh.com/patchDocText - HTTP PATCH Request

Update the document text of a document resource

Sample success response: no body

7.2 Code Executor

The Code Executor microservice handles the execution of Python code passed to it from the Sessions microservice, returning any output or errors [CE-1] [CE-2]. It is built with Express and Node.js.

Upon receiving a request containing the code, the Code Executor will first write the code to a temporary python file. It then spawns a child process which will execute the python file, returning the results of the execution to the Sessions microservice as a response. It is set to timeout after 10 seconds in order to prevent server overload. [CE-4]

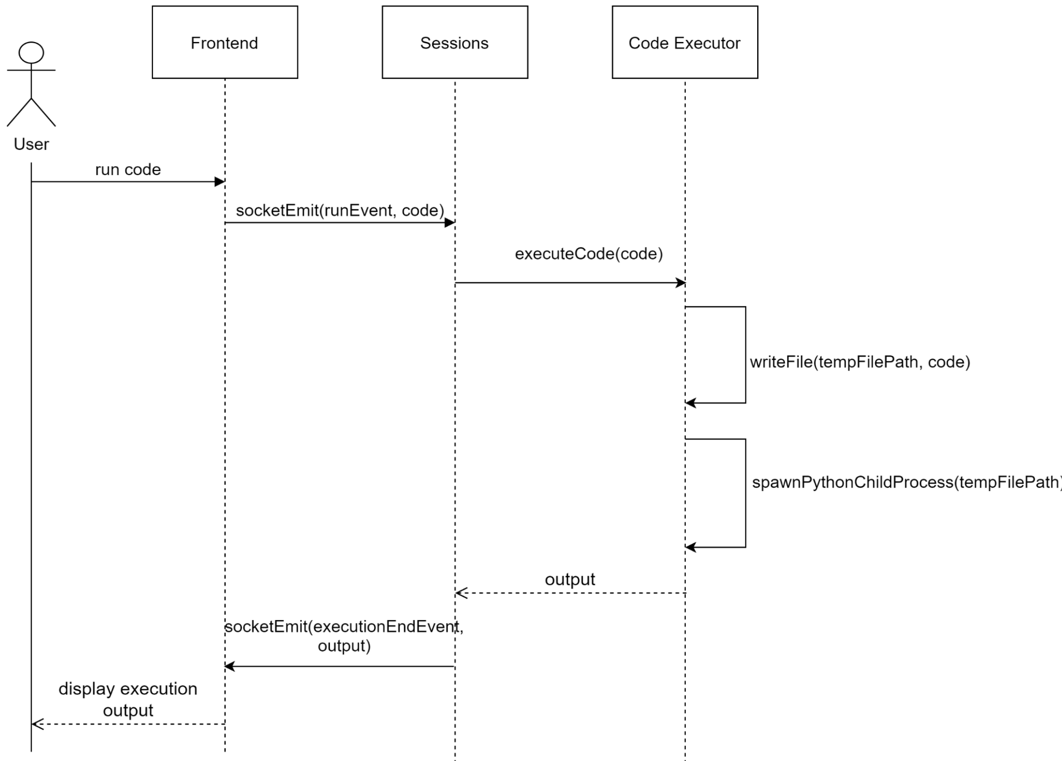

Figure 7.2.1: Sequence Diagram for code execution

In the sequence diagram above, the entire flow of the code execution is depicted as follows:

1) The user clicks a button to run the code
2) A ‘run’ socket event is being emitted by the Frontend using Socket.io (over websockets), along with the code data
3) The Sessions service, which is listening for this ‘run’ event, proceeds to make a post HTTP request to the Code Executor service, passing along the code data
4) The Code Executor writes this data to a temporary file, then spawns a child process to execute the code. The output is returned in the HTTP response to the Sessions service
5) The Sessions service emits a ‘executionEnd’ event to the Frontend, along with the output/errors that were the result of code execution
6) The Frontend listens from this ‘executionEnd’ event and proceeds to display the output/errors received to the user

}

Supported Endpoints:

Execute Code

<kubernetes internal IP for code executor service>/code-executor

- HTTP POST Request

Executes the Python code

Sample body (no errors):

```
{
    "code": "a, b, c = 1, 2, 3\nprint(a + b + c)",
}
```

Sample success response: JSON

```
{
    "output": "6\n",
    "message": "Code ran without errors."
}
```

Sample body (syntax error):

```
{
    "code": "a + +",
}
```

Sample failure response: JSON

```
{
    "error": "File <temporary file path>, line 1\r\n\ta + +\r\n
^\r\nSyntaxError: invalid syntax",
    "message": "Errors were found while executing the code."
}
```

Sample body (loops forever, server timeout):

```
{
    "code": "while true:\n\tprint('Looping forever')",
}
```

Sample failure response: JSON

```
{
    "error": "Time Limit Exceeded: Code took too long to finish
executing.",
    "message": "Errors were found while executing the code."
}
```

7.3 Sessions

The Sessions service is in charge of maintaining the state of active code docs for everyone currently accessing it. It is also in charge of calling the Code Executor API and preventing further edits whenever the code is being executed. [SS-1] [SS-2] [SS-4] [SS-5] [SS-6]

It primarily communicates with the Frontend through socket.io pub-sub messaging.

7.3.1 Subscribed Events

Here is the list of events that the Sessions service subscribes to.

Event	Data	Description
get-document	documentId : string userId: string	Open a currently active document session, or create a new session. A load-document event will be emitted if a document can be found. A document-not-found event will be emitted if no document with the specified document ID can be found in the main docs db, and the user will be disconnected from the socket.
send-changes	delta: Delta	Update the document for everyone, with the change described by a Delta JSON.

run-code	userId: string data: string	Run the current code. Editing will not be possible while the code is being run.
save-document	data: string	Save the document in the cache, which will be written to the docs database every 10s.
disconnecting	reason: string	This event is automatically sent by the socket whenever the user disconnects.

7.3.2 Published Events

Here is a list of events that the Sessions service emits to the socket connection..

Event	Data	Description
document-not-found	docId: string	The requested document cannot be found in the docs database.
receive-changes	delta: Delta	Update the document with changes made by another editor.
code-execution-start	userId: string data: string	The code document is currently being executed by the code executor. No changes can currently be applied.
code-execution-end	data: { output: string, message: string } | { error: string, message: string }	The code execution has ended, providing either a good or bad output.
code-still-running		The code doc you are trying to edit is still running a code execution.
user-disconnecting	reason: string	A user has disconnected from the session

8. Frontend

8.1 Tech Stack

1. React
2. Ant-design

React is a Javascript library for building frontend user interfaces. It is component-based, and maintains a virtual Domain Object Model (DOM). For efficient update and rendering on each interface change, a differing algorithm is used to select parts of the DOM that actually change.

Ant-design is a React UI library that contains enterprise-class UI designed React components. Ant-design allows us to use high-quality and well designed React components out of the box, so that we do not need to reinvent the wheel for commonly used interface components like buttons. The styling of components from ant-design is also customisable so we are able to change the style to fit that of PeerProgram.

9. Application Screenshots

9.1 Landing Page

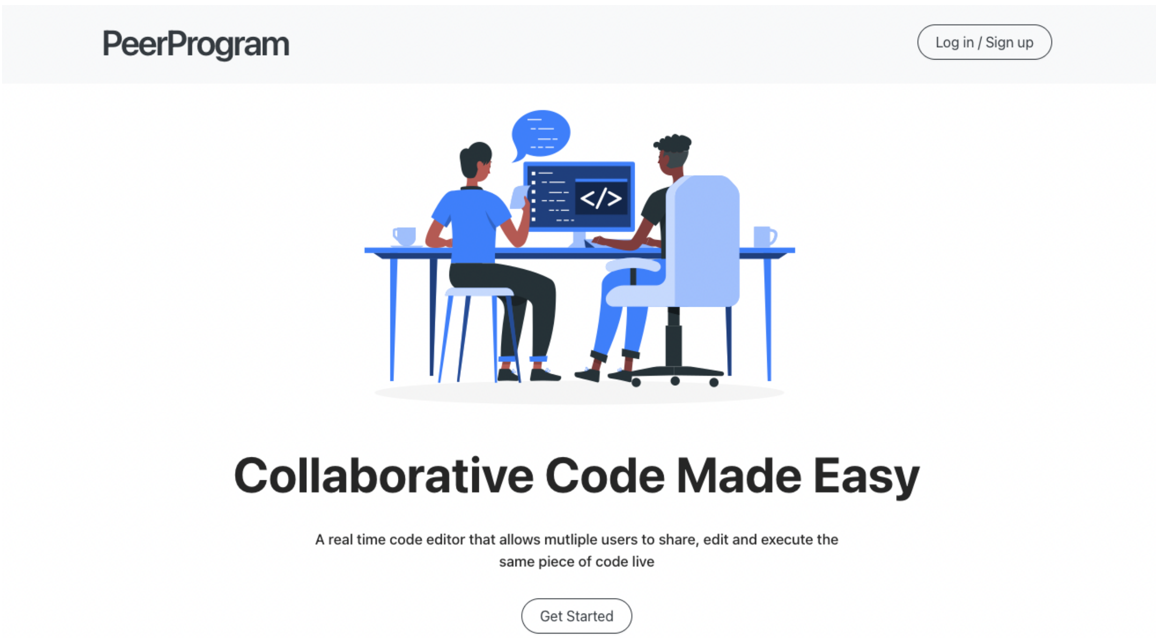

Figure 9.1.1: PeerProgram Landing page

Beforing logging in, users will be shown the landing page, which has a brief description of what PeerProgram is. [FC=6]

9.2 User Authentication

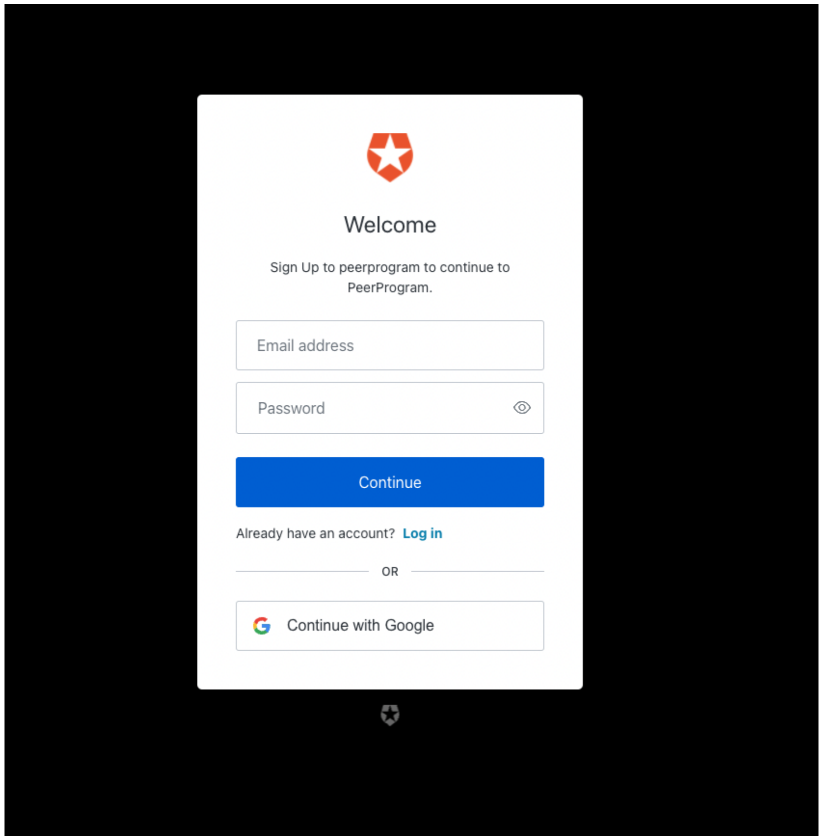

Figure 9.2.1: Auth0 User Login Page

Upon clicking “Log In / Sign Up”, the user will be redirected to Auth0’s sign up page. Existing users can click on “Log In” to be redirected to a similar page. [FC-1]

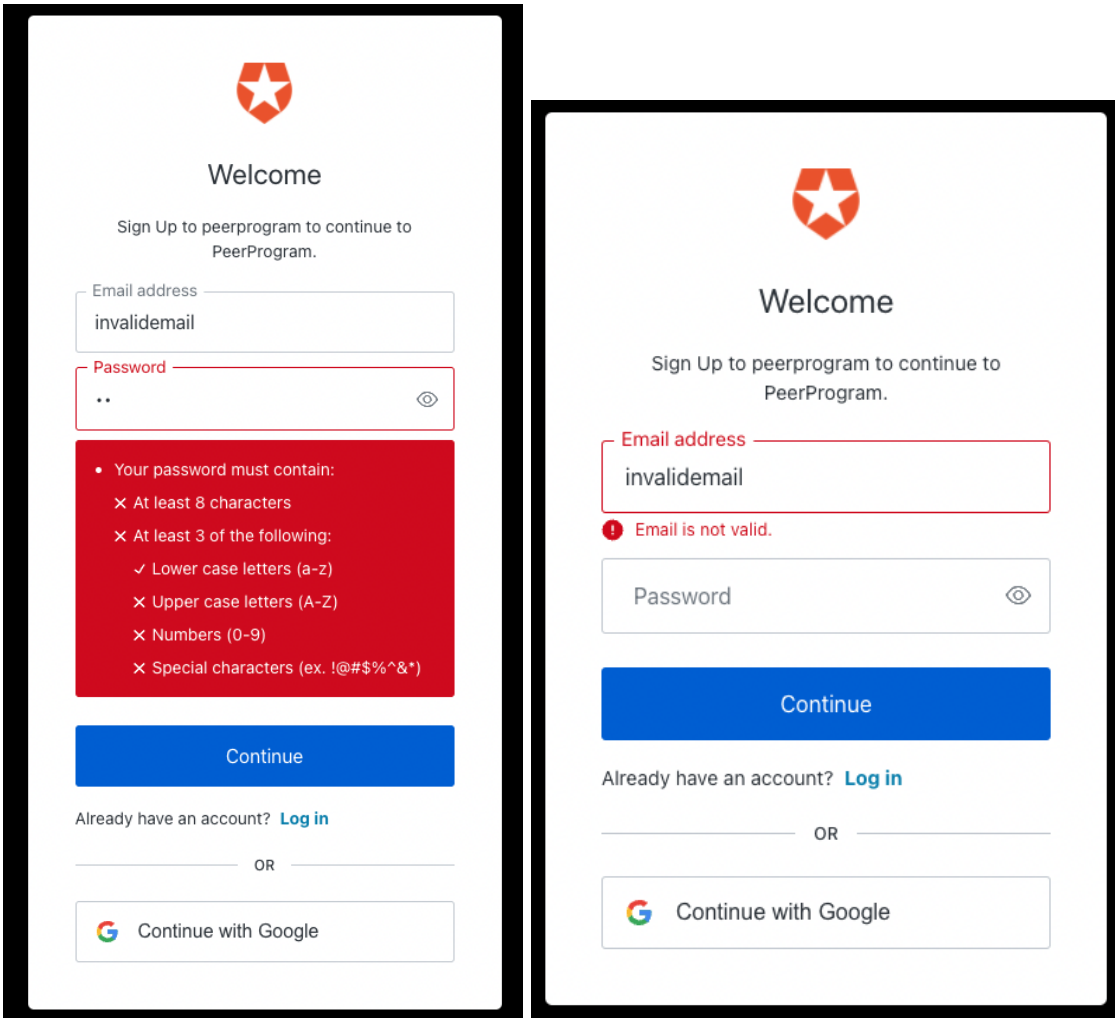

Figure 9.2.2: User is unable to proceed if fields are invalid

The Auth0 components handle basic data validation. The user will be alerted if there are any invalid fields or password strength does not meet the security requirements (for signing up).

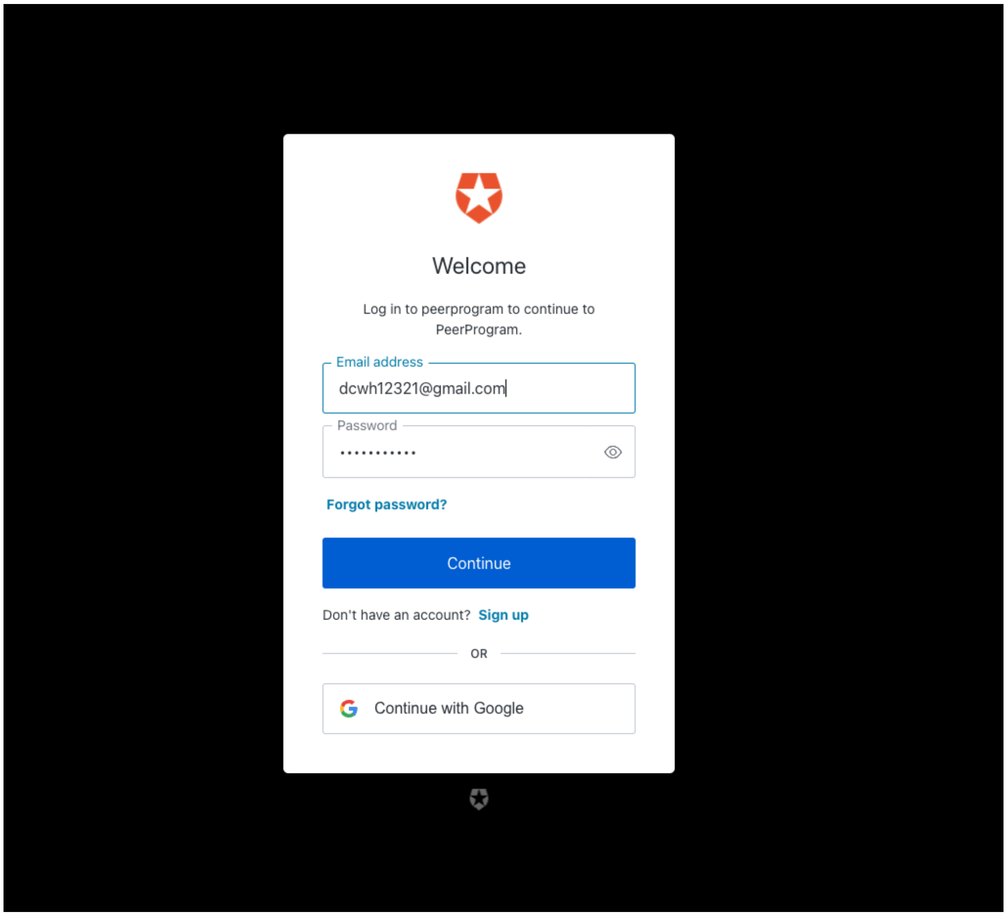

Figure 9.2.3: User is unable to proceed if fields are invalid

The user is able to continue with the login process if the fields are valid.

9.3 User Dashboard

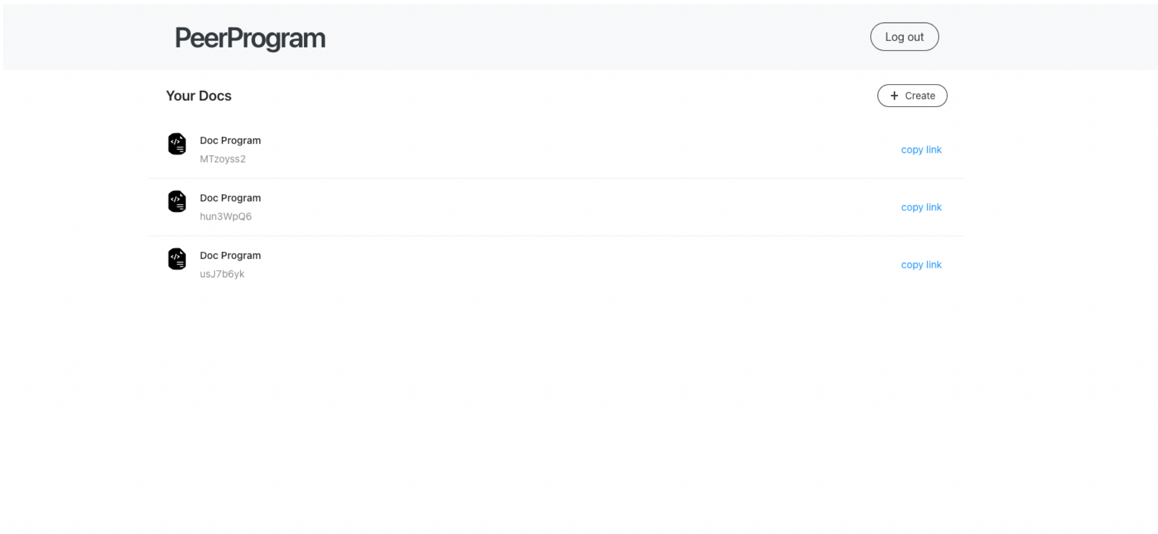

Figure 9.3.1: PeerProgram User Dashboard page

Once logged in, the user can see the list of code documents that they have created (if any). Here, they can choose to create a new document, view an existing document or copy the link of the document to share with others for code collaboration. [FC-4] [FC-2]

9.4 Document

9.4.1 Editor

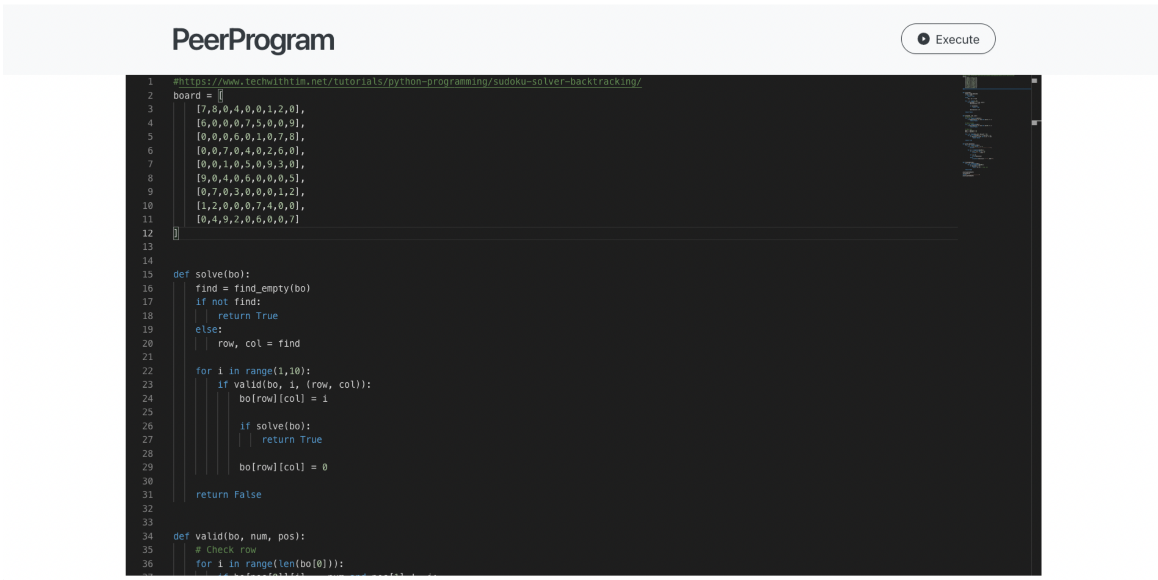

Figure 9.4.1.1: PeerProgram Code Editor Page

After choosing a document from the dashboard, users will be directed to the document’s code editor page. In this page, users are able to write Python code and execute it. [FC-5]

9.4.2 Executing State

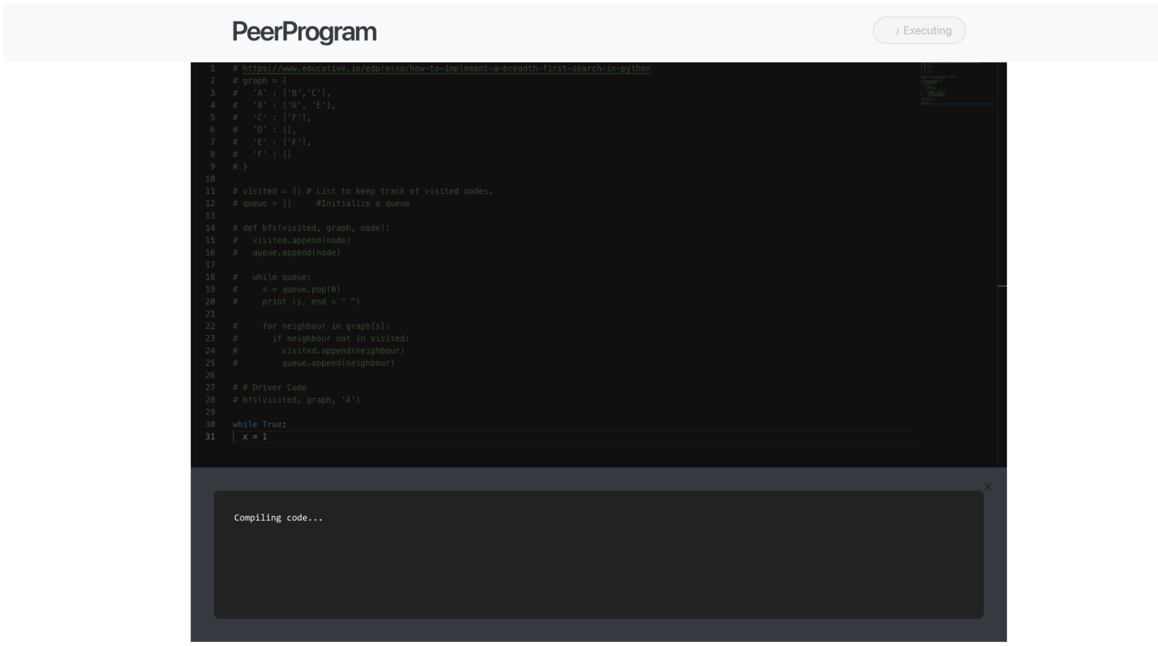

Figure 9.4.2.1: Console view after code execution

When any user clicks “Execute”, a bottom view will overlay the editor, preventing a user from editing the code. “Compiling code…” will be displayed.

9.4.3 Execution Results Shown

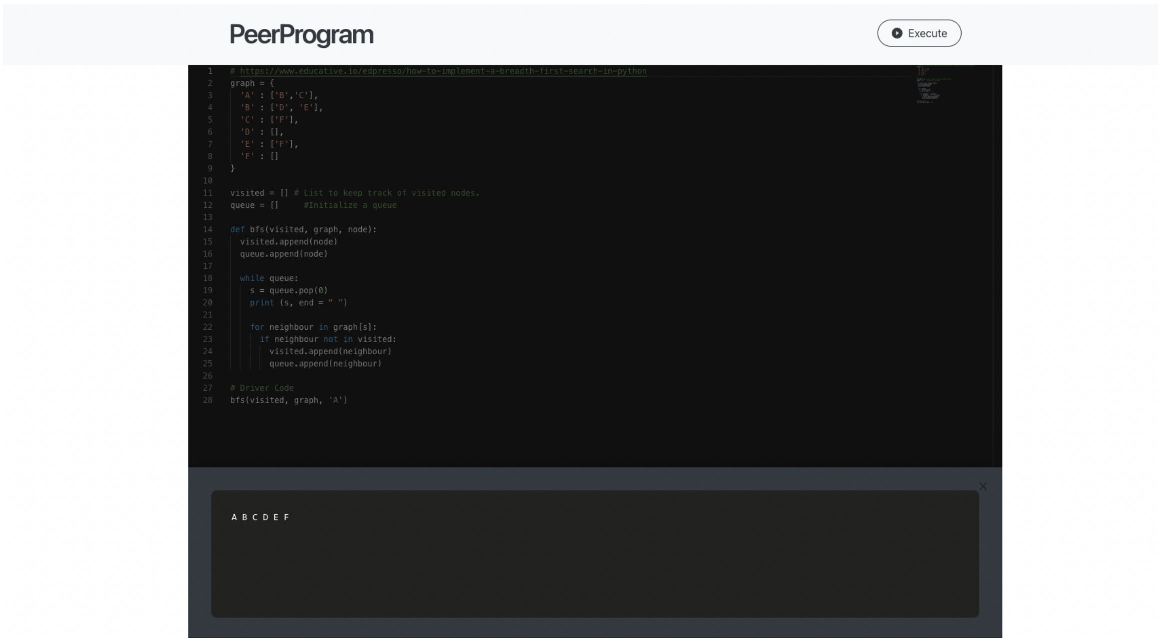

Figure 9.4.3.1: Console view after execution is complete

After execution is complete and the client-side receives the execution results, the bottom overlay will display the code execution output. Users will then be able to close the bottom overlay and edit the code again. [FC-3]

10. Testing Plan

Test cases were designed and written to ensure the application is working as expected. In this project, a mixture of unit testing, functional testing and system testing were done.

10.1 Unit Testing

Unit tests were written for the Code Executor function - `runPython`. This particular function is important and ensures the usability of the application.

The Jest testing framework was used.

Test ensures the `runPython` parses the input correctly and is able to execute python code successfully.

No.	Test Case	Rational
1	Basic 1 line python code that prints “hello world!”	Ensure function works for a 1 line code
2	Simple arithmetic	Ensure function can perform arithmetic operations
3	Wrong indentation causes errors	Python code requires correct indentation for it to work
4	Correct indentation, no errors	As a follow up 3, a correct example was written
5	Use of import statements - standard module (math)	For standard Python modules, import statements are supported and can be used
6	Use of import statements - external module (pandas)	For external Python modules, import errors are expected
7	Multiple lines of code with a written function, fibonacci, no errors (small input value)	Ensure function works for multiple lines of code. A `def` is included to ensure the function can parse it correctly.
8	Multiple lines of code with a written function, fibonacci, causes errors (large input value)	As a follow-up of 7, large input values take longer execution time. Since a 10s limit was given in the requirements, we expect this test case to produce a timeout.

10.2 Functional Testing

In order to check if the API is running properly, we used Postman to test our endpoints. Each individual services API was tested. There were 2 things we can find out from this test. Firstly, ensuring that our endpoints are working. Secondly, making sure the response time is well within our requirements.

In this document, we showcase some of the test results for the Docs service.

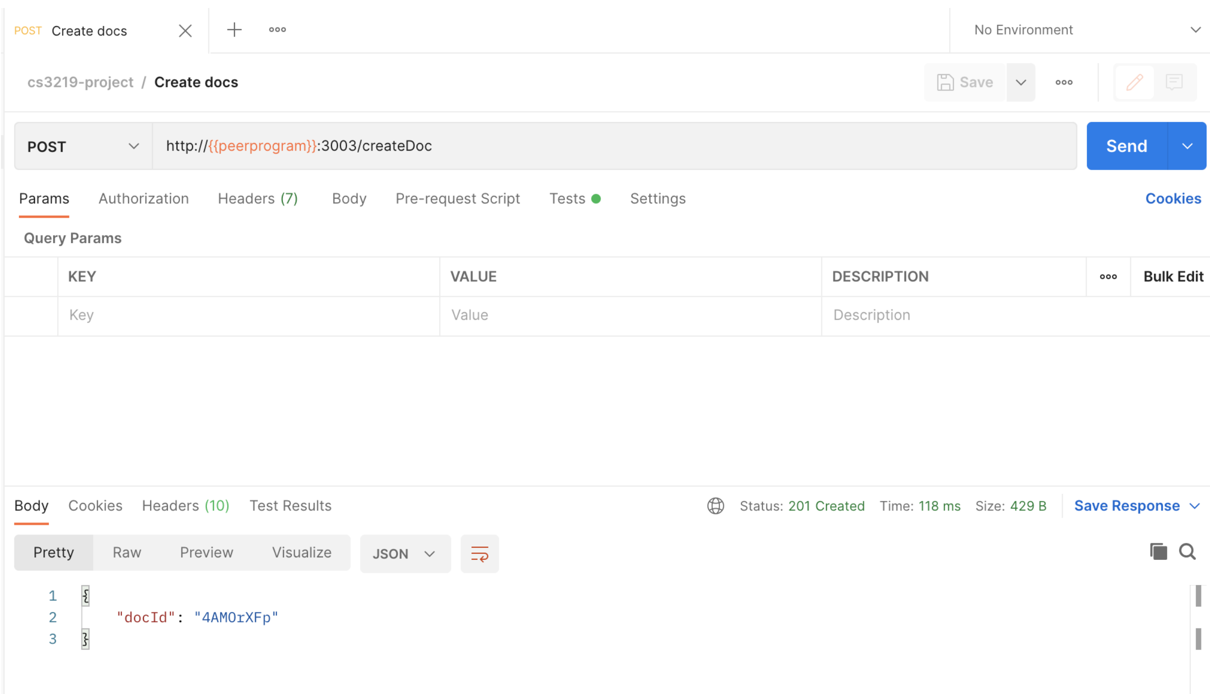

Figure 10.2.1: Postman API testing - POST: Create Docs

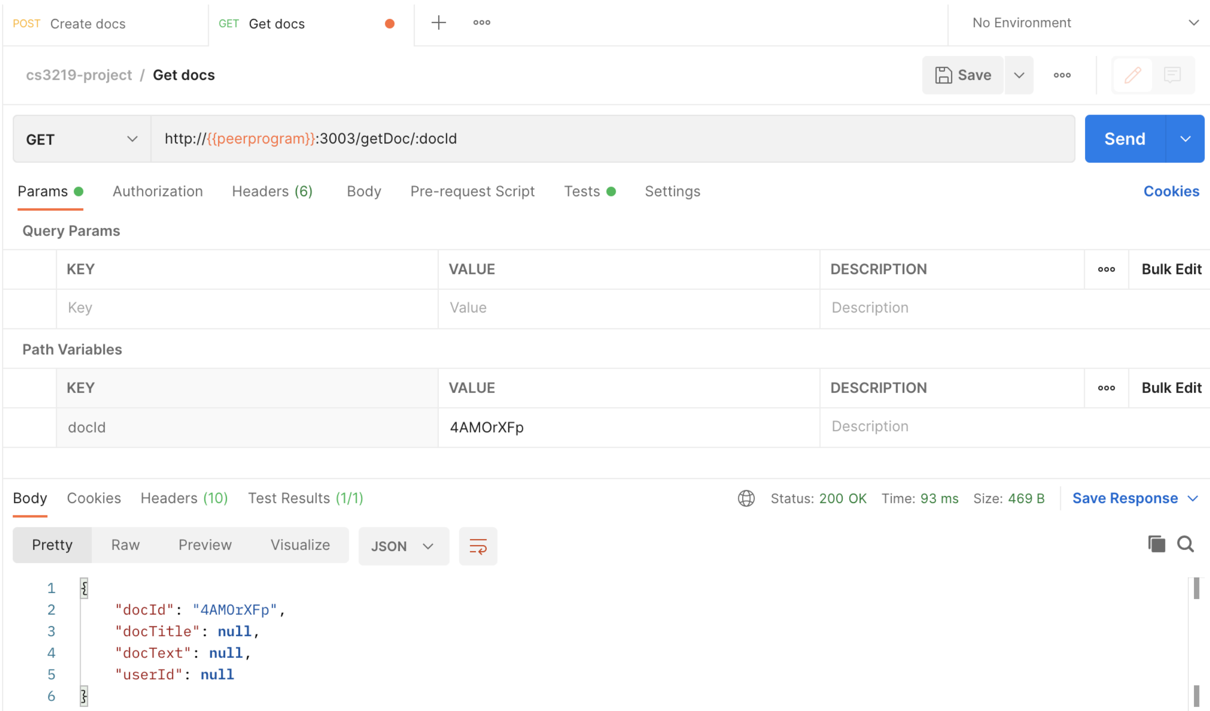

Figure 10.2.1: Postman API testing: GET: Get docs

10.3 System Testing

In order to understand the capabilities of our system, load and stress testing were conducted. These tests will be able to check if our non-functional requirements are met. In particular, we can test for PR1 and SR2.

10.3.1 Load Testing

Load testing allows us to know if there were any issues with performance. We use Apache JMeter to perform load testing.

This allows us to check if the amount of concurrent users set can be handled by the application without errors in normal conditions.

Below is a load testing test plan in Jmeter for a GET request:

Number of threads: 10
Ramp-up period (time taken in seconds for threads to be up): 100
Loop-count: 1

After executing the test, we received the results for viewing in the summary report section.

In this particular test run, 10 requests were sent with an average response time of 6 seconds. There were no errors.

This can be interpreted as our system is able to handle this load and is working as expected.

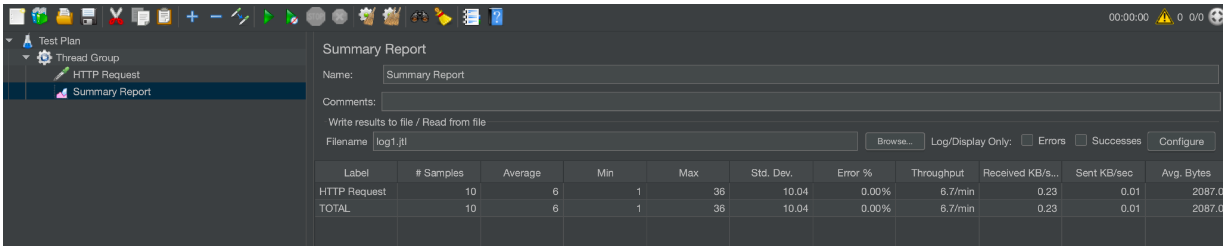

Figure 10.3.1.1: Summary Report from JMeter for a Load testing run

10.3.2 Stress Testing

Next, we test our system in extreme conditions. In these conditions, we would be able to see how our system responds and handles such requests.

Similarly, we conducted our test with Jmeter.

Below is a stress testing test plan in Jmeter for a GET request:

Number of threads: 1000
Ramp-up period (time taken in seconds for threads to be up): 600
Loop-count: infinite
Thread lifetime (in sections): 1200

This translates to: simulating 1000 virtual users in 10 minutes for a span of 20 minutes.

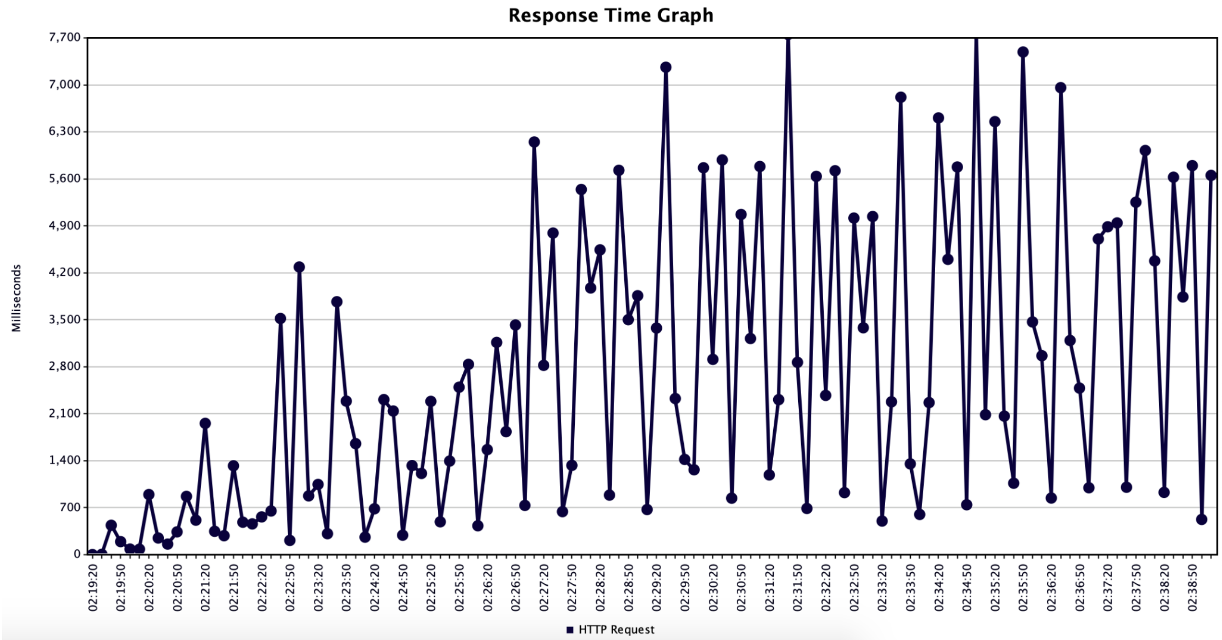

Figure 10.3.2.1: Response Time Graph from JMeter for a Stress testing run

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	644308	1405	0	30687	4840.59	25.42%	532.3/sec	1164.59	44.97	2240.5
TOTAL	644308	1405	0	30687	4840.59	25.42%	532.3/sec	1164.59	44.97	2240.5

Figure 10.3.2.2: Summary report from JMeter for a Stress testing run

After the simulation ended, we proceeded to analyse the results. From the summary report, it was reported that there was an error of 25.42%.

Moving on to the response time graph, we can understand that there is a big variation of response times throughout the test. At the start of the test, response time was fast as there were fewer virtual users. However, as more virtual users were added, the response times varied drastically.

In an ideal scenario, a good result would be a horizontal line across the graph with minimal deviations to show that response times are stable. With this simulation, we can measure our system's maximum capacity.

11. Devops

In this project, our team has decided to follow the Agile framework. With a project duration of 6 weeks, we aim to conduct 6 weekly sprints to complete our application.

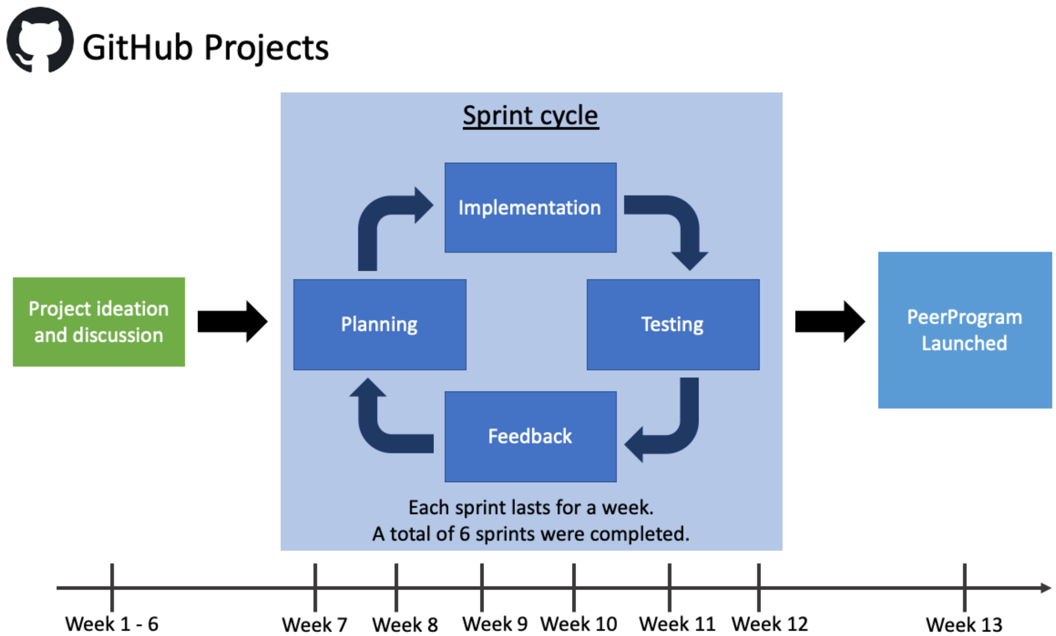

Figure 11.0: Overview Project Timeline

11.1 Tools

For program management, the GitHub Projects board was used. 5 blocks were created for each component. In a block, individual cards were created to represent a functional requirement. The cards were given a priority ranking of high, medium and low for schedule and planning purposes.

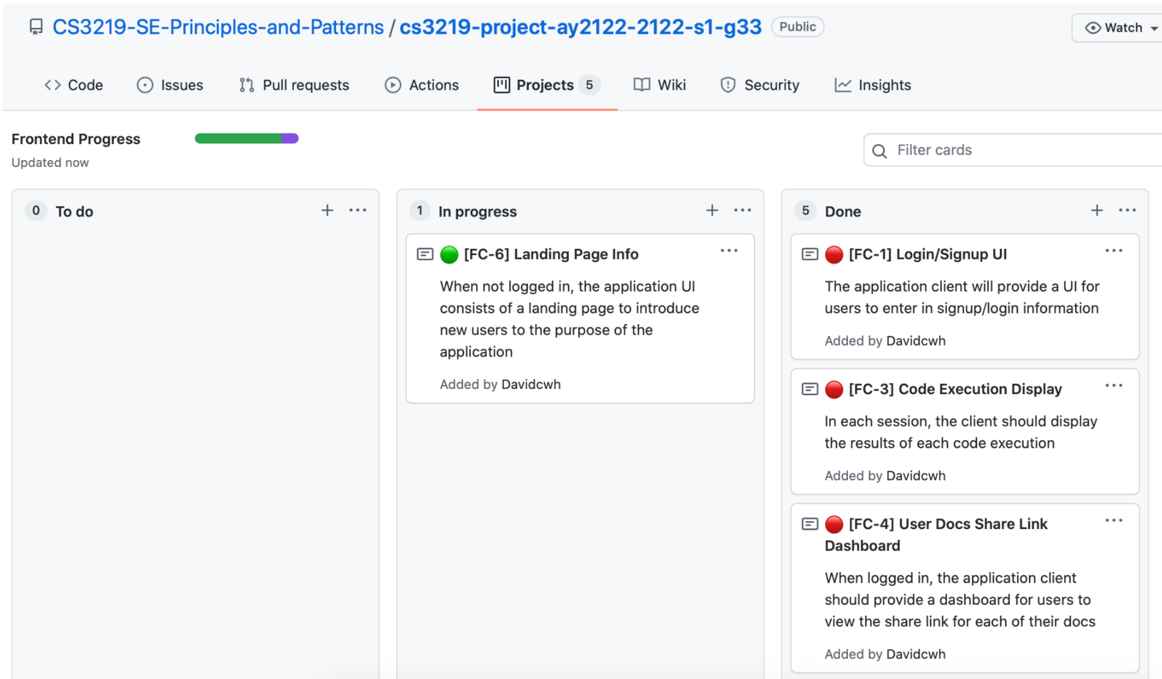

Figure 11.1.1: Screenshot of GitHub Project board for the Frontend

11.2 Sprints

Each sprint lasts for a week. Every Friday, the team will have an online meeting on Discord. Our goals for the meeting were as follows:

1. Individual progress update on tasks completed and any troubles faced
2. Resolve teammates' issues
3. Planning of tasks for the next sprint

In between a sprint, members can give an update or report an issue faced on our group's Telegram chat.

11.3 CI/CD

CI/CD was implemented with GitHub Actions. For CI, when a commit is pushed onto GitHub, a build and automated testing is executed. For CD, given a passed build, the commit will be merged, built and deployed onto GCP.

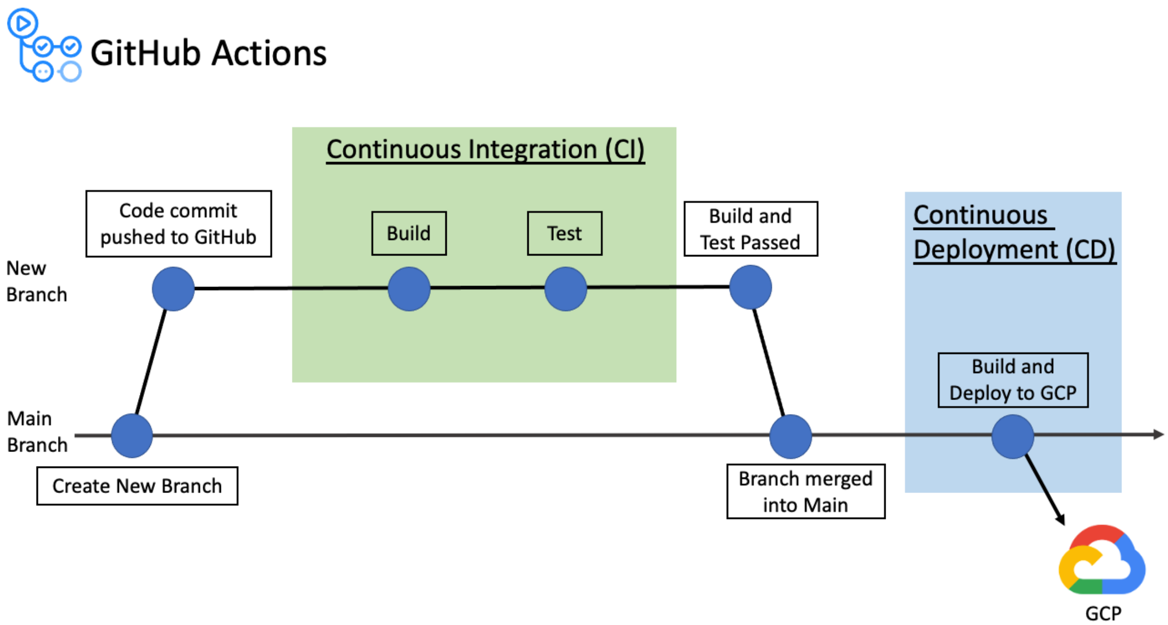

Figure 11.3.1: CI/CD flow for a passed build

12. Suggestions for improvements and enhancements to the delivered application

12.1. Support for more programming languages

The Code Executor service can support other popular programming languages such as Java, C++, Javascript. Docker containers need to install Software Development Kits (SDKs) of these languages as part of their dependencies in order to compile and run the code.

On the Frontend, the option to select the choice of language can be done in a dropdown menu. Syntax highlighting for different languages can be easily added as the React editor component used, Monaco editor, already has syntax highlighting built in.

12.2. Privacy Sharing Settings For Documents

To give more flexibility to document sharing, a good feature to add would be to allow users to set privacy options for writing and reading a document. For each of their created documents, the user will be able to indicate if the document is free to be edited or viewed by anyone with the link (current implementation), or only allow certain users to be able to do so.

We can do this by storing the privacy settings for each document in the Docs database as well. The privacy settings will contain a list of user Id’s with access to the document. When users try to access a restricted document, logged in users will send a request with a JWT from Auth0 to backend services to authorize themselves for access to the document by checking if the user’s Id is in the list of allowed users for the document.

13. Reflections and learning points from the project process

13.1 Reflections

The project was a good way to allow us to apply the points learnt in lecture into a more practical scenario. It gave us an insight to the development process of an application.

It is easy to talk about using microservices but getting the configuration correct and optimized is a much different task.

13.2 Learning points

- Docker and Kubernetes
  - Learning about container-orchestration was quite challenging.
  - Proper planning and understanding of requirements is essential in order for the components to work.
- Microservice Architecture
  - Learning about microservice architecture, the use cases and how to use it.
  - Drawing a microservice architecture diagram.
- Deploying the application to the cloud (GCP)
  - Learning about GCP and using the `gcloud` command
  - Storing secrets and getting authentication to use GCP
- CI/CD with GitHub Actions
  - Learning how to write GitHub Actions workflow file
  - Automating tests and builds before merging to the main repository
  - Automating deployment to GCP from a passed build in GitHub Actions.
- Software Testing
  - Understanding the importance of testing
  - Learning to use various testing tools: Jest, Postman, Jmeter