



**NUS**  
National University  
of Singapore

---

# School of Computing

CS3219 Software Engineering Principles and Pattern  
AY21/22 Semester 1  
Project Report  
Group 2

Team Members	Student No.	Email
Lim Wei Quan, Ernest	A0201835M	e0415644@u.nus.edu
Anikesh Bhuvaneshwaram	A0200603A	a.b@u.nus.edu
Ambrose Liew Cheng Yuan	A0204750N	e0424673@u.nus.edu
Chan Yong Soon, Kendrew	A0197077N	e0392519@u.nus.edu

## **1. Background and purpose of project**

It is difficult to learn a new language. Even with the help of beginner modules in NUS, students tend to struggle while learning a new language. There is not only the need to understand and familiarise yourself with many new vocabulary words but also to understand how the language is structured.

Due to the overwhelming breadth of content, it is challenging for learners to organise their learning through methods such as traditional rote memorisation techniques. With the advent of technology, tools such as online flashcards have become commonplace to help reduce the difficulty in memorising vocabulary or learning grammar. There is no consolidated platform which allows for learners to have quick access to sample sentences along with vocabulary words. Often, learning a new language is an individual and arduous pursuit, and we hope to leverage the social aspects of online sharing platforms to help in this fruitful endeavour.

Our website builds on this pain point of people trying to learn new languages by providing a platform for people to create flashcards as well as to play vocabulary games against each other. LanguageLearners allows users to create flashcards in Korean/Japanese and play vocabulary games against other users of a similar rating. Our project, in essence, is built around making the process of learning a new language less tedious and more enjoyable for users.

## 2. Individual contribution

Name	Contribution
Ambrose	<ul style="list-style-type: none"><li>- Make website mobile responsive</li><li>- Set up login authorization</li><li>- Set up frontend testing</li><li>- Finish up profile webpage</li></ul>
Anikesh	<ul style="list-style-type: none"><li>- Set up matchmaking capabilities</li><li>- Set up game functionality</li><li>- Make frontend for the game</li><li>- Integrate frontend with backend for game</li><li>- Set up boilerplate code for flashcards/profile microservice</li><li>- Set up socket testing</li></ul>
Ernest	<ul style="list-style-type: none"><li>- Set up frontend architecture</li><li>- Set up docker containers</li><li>- Set up kubernetes locally</li><li>- Add boilerplate for frontend</li><li>- Create flashcard and profile pages for frontend</li><li>- Integration of flashcard and profile pages with backend</li><li>- Deployment to Google Kubernetes Engine</li><li>- Load Testing with Locust</li></ul>
Kendrew	<ul style="list-style-type: none"><li>- Setup backend architecture</li><li>- Worked on login, profile, flashcard, matchmaking microservice backend database and routes</li><li>- Worked on login, profile, flashcard, matchmaking microservice backend unit tests</li><li>- Worked on profile details and flashcard on frontend</li><li>- Worked on create/edit profile page</li><li>- Worked on leaderboard page</li></ul>

### 3. Specified and prioritized requirements of the application

#### Functional Requirements

Sorted in descending order of priority, uses the same numbering convention as the Project Proposal. The functional requirements below are the ones that are implemented by our team.

#### 1) Login System

- I. The system should allow the user to create an account (L1.0)
- II. The system should authenticate the user when the user enters their login credentials (L2.0)
- III. The system should login the user when the user is authenticated. (L3.0)
- IV. The system should authorize actions by authenticated users. (L6.0)
- V. The system should inform the user if the user is not authenticated. (L2.1)
- VI. The system should allow the user to logout. (L3.1)
- VII. The system should recognise users and remember their login information (L5.0)

#### 2) Flashcards

Stated below are the original functional requirements for flashcards that were modified to suit the flashcard set requirement, refer to New FR section for more details

- I. The system should allow the user to create flashcards for themselves (F1.0)
  - II. The system should allow the user to edit their own flashcards (F1.1)
  - III. The system should allow the user to delete their own flashcards. (F1.2)
  - IV. The system should allow the user to view their own flashcards. (F1.3)
  - V. The system should allow the user to search for flashcards by title of flashcard. (F5.1)
  - VI. The system should allow the user to sort the flashcards by date. (F5.3)
  - VII. The system should allow the user to sort the flashcards by alphabetical order. (F5.4)
  - VIII. The system should allow the user to type in many different languages. (F6.0)
- 
- I. The system should allow the user to create a flashcard set (FS1.0)
  - II. The system should allow the user to add flashcards to their own flashcard set (FS1.1)
  - III. The system should allow the user to delete flashcards in their own flashcard set (FS1.2)
  - IV. The system should allow the user to view their flashcard sets (FS1.3)
  - V. The system should allow the user to view their flashcards in their own flashcard set (FS1.4)
  - VI. The system should allow the user to search for flashcard sets by title of flashcard set (FS1.5)
  - VII. The system should allow the user to sort the flashcard sets by date created (FS1.6)
  - VIII. The system should allow the user to sort the flashcard sets by alphabetical order (FS1.7)

#### 3) Game

- I. The system should allow users to be able to submit their answers to the question (G1.0)
- II. The system should check correctness of answers submitted by user (G1.1)
- III. The system should store the users points (G3.0)
- IV. The system should allow the user to view their points (G3.1)

- V. Users should be able to gain leaderboard points upon winning (G2.1)
- VI. Users should be able to lose leaderboard points upon losing (G2.2)
- VII. Points gained or lost should depend on speed and correctness of completion (G2.3)

#### 4) Matching

- I. The system should be able to match users with other users based on language. (M1.0)
- II. The system should be able to match users with other users based on rank similarity. (M1.1)
- III. The system should match one user to another user (M3.0)
- IV. The system should inform users if it is unable to match them with suitable opponents. (M2.0)

#### 5) Leaderboard

- I. Users should be able to see their leaderboard points. (L1.0)
- II. Users should be able to see their ranking on the leaderboard. (L1.1)
- III. Users should be able to sort users based on name. (L2.0)
- IV. Users should be able to sort users based on ranking. (L2.1)
- V. Users should be able to search for other user's names. (L2.2)

#### Unfulfilled Functional Requirements

Functional Requirement	Priority	Reason for not fulfilling
L4.0 The system should allow the user to reset his password	Low	<p>There are two possibilities for implementing this feature:</p> <ol style="list-style-type: none"> <li>1. Allowing users to just reset passwords as long as they know their usernames/emails (unauthenticated)</li> <li>2. Making sure of the user identity using security questions or email verification</li> </ol> <p>Option 1 was rejected as it would mean any user could reject any other user's password without being authenticated which is not at all desirable.</p> <p>Option 2 required integrating either email verification or implementing a security questions option which is a tedious and unnecessary process given our relatively low security prioritization.</p> <p>Although our team initially developed option 1 of the reset password feature, it was eventually removed due to the above reasons. However, should we decide to improve the security of our application it can be extended from the current implementation.</p>
F2.0/2.1: The system should allow users to share/view flashcards with/shared by other	Low	Implementing these features presented its own set of challenges with regards to creating a new model or changing the current flashcard models to handle shared flashcards.

users		These features required increasing the complexity of our flashcard microservice significantly and as a result, we chose to prioritize work on our game since it was of higher priority.
F4.0/5.1/5.2: Specification of tags on flashcards and searching/filtering using these tags	Low	This feature is not particularly difficult to implement. Nevertheless, during our weekly discussions, we came to the consensus that tags would be useful only if flashcards were to be shared around by users. Without implementing the sharing feature, searching or filtering by tags becomes an unnecessary requirement.
F3.0: The system should retrieve definitions of words in English	Low	Over the course of the development process, this was considered to be an extremely low priority feature especially considering the ease with which google translate can already be used.
G2.0: The system should display a timer to the user about the amount of game time left	Medium	Upon reassessing our priorities during one of the weekly meetings, it was decided that this as a feature is not of medium priority but in fact much lower. In order to focus on more important priorities like fixing bugs in the game itself, this feature was replaced by a functional requirement to make the number of correct answers viewable to players..

### New Functional Requirements

FS1.0-1.6: Functional Requirements pertaining to flashcard set	Medium	<p>During the development process, our team discussed the requirements more deeply and we realised that it would be more convenient for the user to be able to group their flashcards into a flashcard set, mimicking closely a textbook style, where the vocabulary lists are divided into different chapters. This allows for more easy organisation of the users' flashcards and better separates the users' learning to manageable divisions.</p> <p>Moreover, it would be more convenient for our team to use such flashcard sets for the game, where it greatly reduces the query time of retrieving the flashcards.</p> <p>The new requirements roughly correspond with the previous requirements (F1.0 - F6.0)</p>
----------------------------------------------------------------	--------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Project Management

Our team used the Github Project tool for Project management. The use of the Kanban Style board helped our team to organise the deliverables that need to be completed. Our team adopts

an Agile style SDLC, where weekly agile meetings are held to consolidate our progress and make any necessary adjustments to the issues and project plan.

Agile style SDLC was used to handle the workloads of all group members as well as to reprioritize requirements. Daily updates on code progress were given by group members and weekly meetings were held to consolidate all of these details and split the tasks for the next week. Meetings were held during the week as well in situations where a task proved too troublesome to implement or bugs needed to be discussed about.

GitHub Project was not used to record the bugs found, as we felt that it would be simpler and clearer to pass the reproduction steps of the bug to the developer in charge through a meeting. Using this approach we managed to fix most bugs found within 3 days.

Our general workflow can be summarised as follows:

1. Meet on Sunday to
  - Discuss about last week's progress
  - Split the tasks for the upcoming week
  - Set up the GitHub Project issues and assign each to the member who is assigned to the issue
2. If a task is completed:
  - Create a Pull Request on GitHub and once it passes CI request for a review from a teammate.
  - Fix any comments/feedback provided by the reviewer
3. If any significant obstacles are faced while implementing a feature:
  - Try for a day to fix the feature up
  - Provide details about the obstacle and tried implementations to other group members
  - Group members meet up for brainstorming sessions to help implement the feature
4. Bugs found:
  - Contact the developer of the feature and provide the details to reproduce the bug
  - Organize a meeting for discussing how to resolve the bug if necessary
5. Meeting on Sunday

### **Non-Functional Requirements**

#### **Reliability:**

- R1.0: The system should be able to maintain connection between the 2 users while in game.
  - So that matches users do not end up losing game progress
- R2.0: The system should not crash due to any invalid input from users.
  - So that the user experience remains positive while using the system
- R3.0: The system should reliably perform any actions made by the user and respond with the appropriate response.
  - So that the user experience remains positive while using the system
- R3.1: The system should reliably retrieve any data from the database and reliably write to the database.

**Performance:**

- PE1.0 The website should be fast and responsive (respond within 10 seconds)
  - So that users can have a positive user experience and focus on their language learnings
- PE2.0: The system should match users of similar ranking within 20 seconds of request.
  - So that users do not have to spend too much time ambiguously waiting for a match
- PE3.0 The system should be able to update user's requests within 5 seconds.
  - To improve responsiveness as close to real time as possible
- PE4.0 The system should be able to load pages within 5 seconds.
  - So that the user experience would be great
- PE5.0 Users should be able to view another flashcard within a second.
  - So that users are able to quickly and easily move from one flashcard to the next
- PE6.0 The system should be able to login users within 5 seconds of inputting their username and password
  - So that users don't spend too much time waiting for their profiles to load up after inputting their passwords and usernames.
- PE7.0 The system should update the leaderboard in real time.
  - So that users will be able to know the updated information as accurately as possible

**Portability:**

- PO1.0: The system should support popular browsers including but not limited to Google Chrome, Mozilla Firefox and Safari.
  - So that users are not limited to using only one type of browser while using the system
- PO1.1: The website should be mobile responsive.
  - So that users can use their mobile devices to view our website and not be restricted to only desktop view.

**Usability:**

- U1.0 The system should be intuitive and easy to follow for new users.
  - So that the learning curve required to use our website would be low and attract more users.

**Security:**

- SE1.0: The user's password should be encrypted.
  - So that their credentials are protected from untoward access
- SE2.0: The system should allow the user to remain logged in if the user has the access token.
  - So that users do not have to keep inputting their usernames and passwords every time they want to use the website

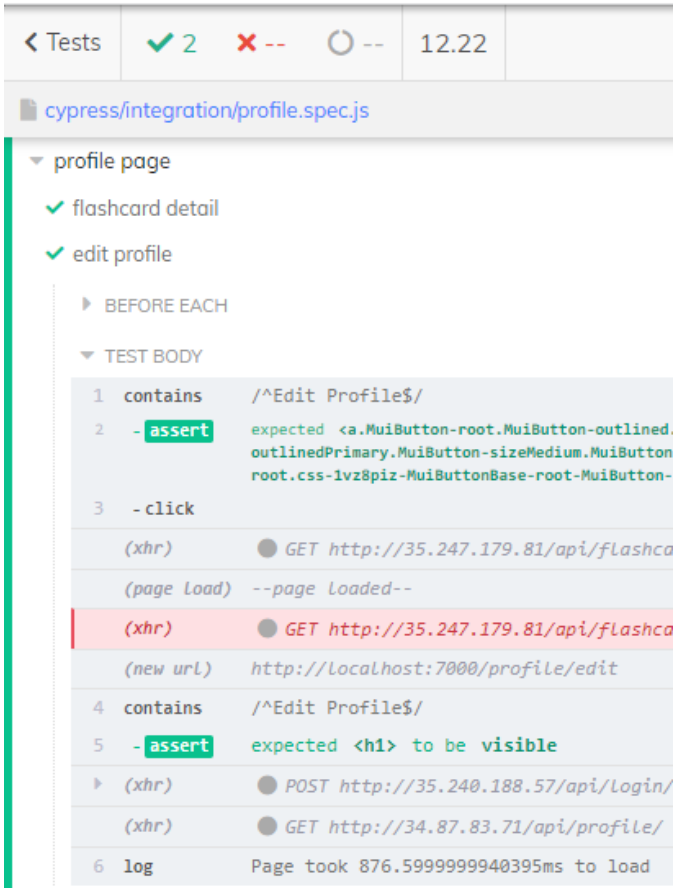
**Scalability:**

- SC1.0 The system should be able to tolerate high user capacity.
  - So that our system can be responsive and serve at least 50 users concurrently at a time.



### Justification for NFR

NFR ID	Description	Justification
Reliability		
R1.0	The system should be able to maintain connection between the 2 users while in game.	Socket IO Testing using Jest
R2.0	The system should not crash due to any invalid input from users.	Frontend Testing using Cypress
R3.0	The system should reliably perform any actions made by the user and respond with the appropriate response	Unit Testing on Backend and Frontend Testing using Cypress
R3.1	The system should reliably retrieve any data from the database and reliably write to the database.	Unit Testing on Backend
Performance		

PE1.0	The website should be fast and responsive (respond within 10 seconds)	 <p>Tests: 2 passed, 0 failed, 0 pending, 12.22s</p> <p>cypress/integration/profile.spec.js</p> <p>profile page</p> <ul style="list-style-type: none"> <li>flashcard detail</li> <li>edit profile</li> </ul> <p>BEFORE EACH</p> <p>TEST BODY</p> <pre> 1 contains    /^Edit Profile\$/ 2 - assert    expected &lt;a.MuiButton-root.MuiButton-outlined.outlinedPrimary.MuiButton-sizeMedium.MuiButton-root.css-1vz8piz-MuiButtonBase-root-MuiButton- 3 - click  (xhr) GET http://35.247.179.81/api/flashca (page load) --page Loaded-- (xhr) GET http://35.247.179.81/api/flashca (new url) http://localhost:7000/profile/edit 4 contains    /^Edit Profile\$/ 5 - assert    expected &lt;h1&gt; to be visible (xhr) POST http://35.240.188.57/api/login/ (xhr) GET http://34.87.83.71/api/profile/ 6 log         Page took 876.5999999940395ms to load </pre> <p>Time taken is found at the bottom</p>
PE2.0	The system should match users of similar ranking within 20 seconds of request.	Matchmaking in the backend uses the concept of promises and timeouts with a timeout set at 20 seconds ensuring that users of similar ranking get matched within 20 seconds.
PE3.0	The system should be able to update user's requests within 5 seconds.	Manual testing

PE4.0	The system should be able to load pages within 5 seconds.	<div> <div> Tests 4 -- -- 11.60 </div> <div> cypress/integration/pages.spec.js </div> <div> page latency check </div> <div> profile page </div> <div> BEFORE EACH </div> <div> TEST BODY </div> <div> 1 contains /^Proficiency\$/ </div> <div> 2 - assert expected :not(script,style):cy-contains( '/^Pr [value~= '/^Proficiency\$/ ' ] not to exist in tl </div> <div> 3 contains /^My Flashcards\$/ </div> <div> 4 - assert expected :not(script,style):cy-contains( '/^My [value~= '/^My Flashcards\$/ ' ] not to exist in </div> <div> get [data-testid=AccountCircleIcon] </div> <div> 6 - assert expected [ &lt;svg.MuiSvgIcon-root.MuiSvgIcon-for MuiSvgIcon-root&gt;, 1 more... ] to be visible </div> <div> (new url) http://localhost:7000/home </div> <div> (xhr) POST 200 http://35.240.188.57/api/Lo </div> <div> 7 - filter :visible </div> <div> 8 - click </div> <div> (new url) http://localhost:7000/profile </div> <div> (xhr) GET --- /api/profile/ </div> <div> (xhr) GET --- /api/flashcard </div> <div> 9 contains /^Proficiency\$/ </div> <div> 10 - assert expected &lt;h5.MuiTypography-root.MuiTypography- to be visible </div> <div> 11 log Page took 1182.7000000029802ms to load </div> </div> <p>Time taken is found at the bottom</p>
-------	-----------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

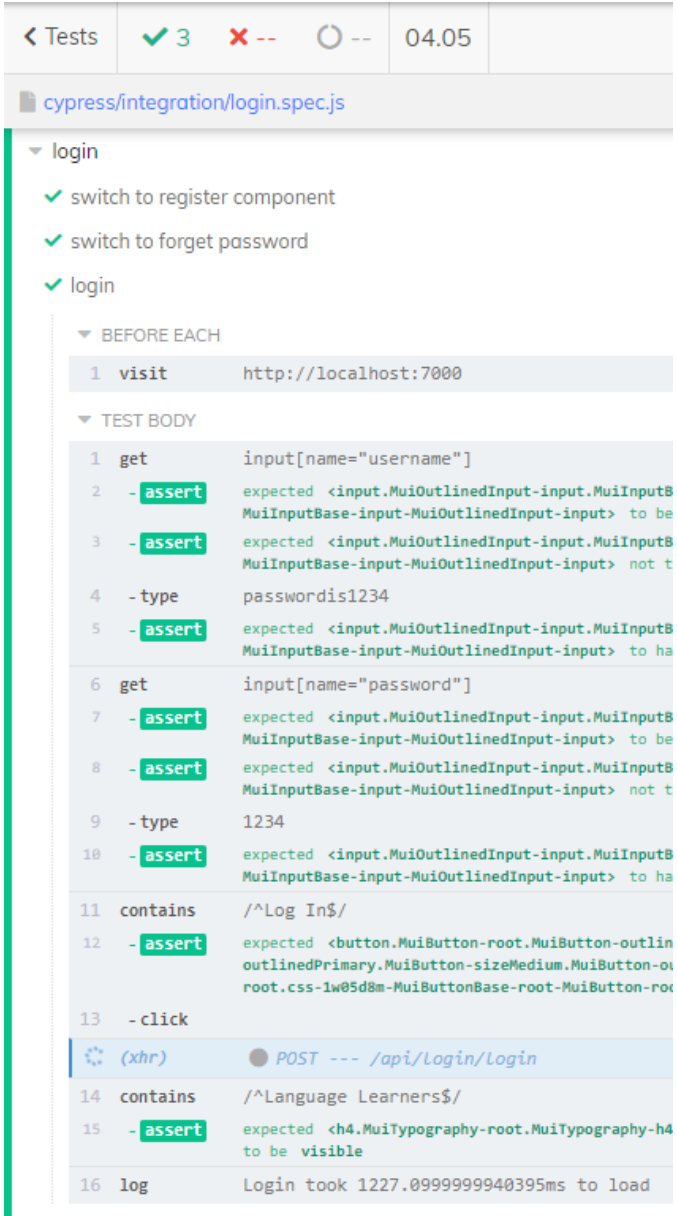
PE5.0

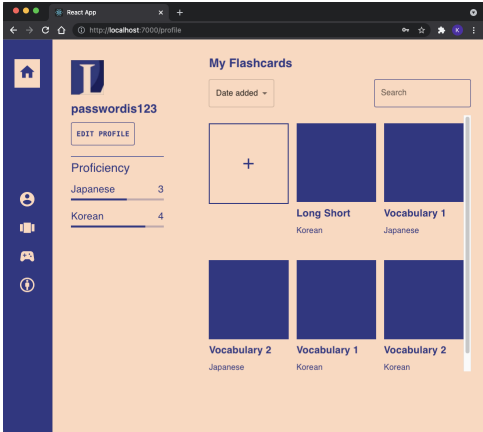
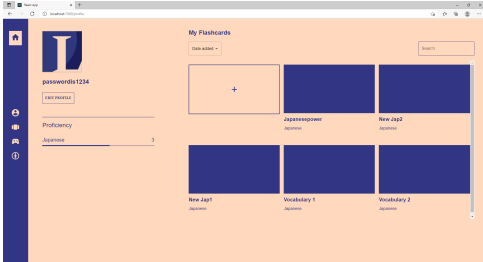
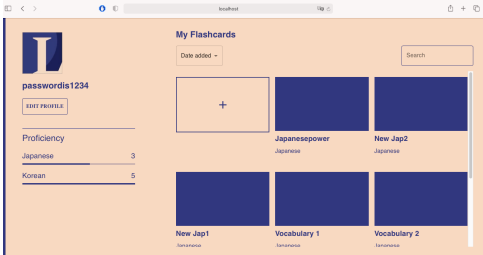
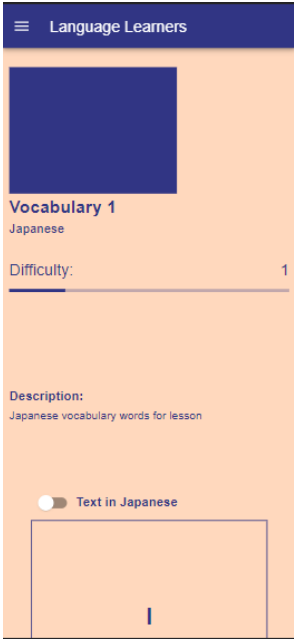
Users should be able to view another flashcard within a second.

The screenshot shows the Cypress test runner interface. At the top, it indicates 4 tests passed, 0 failed, and 0 pending, with a total time of 10.54 seconds. The test suite is 'cypress/integration/pages.spec.js'. The specific test is 'page latency check', which includes a 'profile page' sub-test. The test body consists of several steps: 1. 'contains' assertion for '/^Proficiency\$/'. 2. '- assert' step with an expected error message. 3. 'contains' assertion for '/^My Flashcards\$/'. 4. '- assert' step with another expected error message. 5. 'get' command for '[data-testid=AccountCircleIcon]'. 6. '- assert' step with an expected error message. 7. '- filter' command with ':visible'. 8. '- click' command. 9. '(new url)' command with 'http://localhost:7000/profile'. 10. '(xhr)' command with 'GET --- /api/profile/'. 11. '(xhr)' command with 'GET --- /api/flashcard'. 12. 'contains' assertion for '/^Proficiency\$/'. 13. '- assert' step with an expected error message. 14. 'log' command with the message 'Page took 631ms to load'.

```
< Tests 4 0 0 10.54
cypress/integration/pages.spec.js
page latency check
  profile page
    BEFORE EACH
    TEST BODY
      1 contains /^Proficiency$/
      2 - assert expected :not(script,style):cy-contains( '/^P
[value= '/^Proficiency$/' ] not to exist in
      3 contains /^My Flashcards$/
      4 - assert expected :not(script,style):cy-contains( '/^M
[value= '/^My Flashcards$/' ] not to exist in
      5 get [data-testid=AccountCircleIcon]
      6 - assert expected [ <svg,MuiSvgIcon-root.MuiSvgIcon-fi
MuiSvgIcon-root, 1 more... ] to be visible
      (new url) http://localhost:7000/home
      (xhr) POST http://35.240.188.57/api/Login
      7 -filter :visible
      8 -click
      (new url) http://localhost:7000/profile
      (xhr) GET --- /api/profile/
      (xhr) GET --- /api/flashcard
      9 contains /^Proficiency$/
      10 - assert expected <h5.MuiTypography-root.MuiTypograph
to be visible
      log Page took 631ms to load
```

Time taken is found at the bottom

PE6.0	The system should be able to login users within 5 seconds of inputting their username and password	 <p>Time taken is found at the bottom</p>
PE7.0	The system should update the leaderboard in real time.	Functions as expected
Portability		
PO1.0	The system should support popular browsers including but not limited to Google Chrome, Edge	SS of chrome

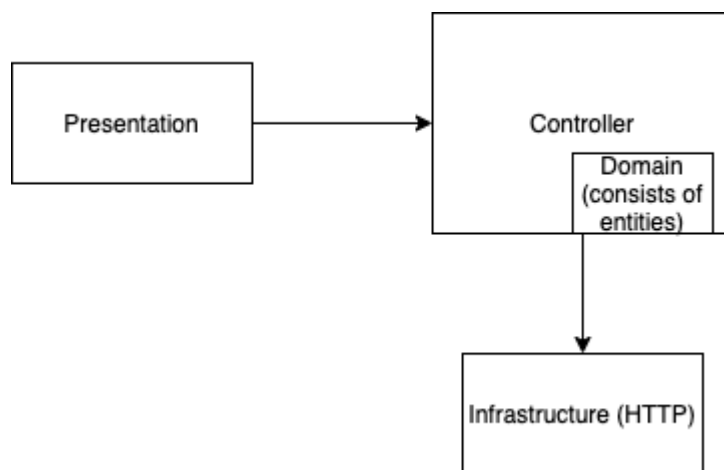
		<div data-bbox="716 210 1200 640"></div> <div data-bbox="716 647 839 683"><p>SS of edge</p></div> <div data-bbox="716 685 1200 945"></div> <div data-bbox="716 952 863 987"><p>SS for Safari</p></div> <div data-bbox="716 990 1200 1243"></div>
<div data-bbox="193 1261 327 2007"><p>PO1.1</p></div>	<div data-bbox="327 1261 699 2007"><p>The website should be mobile responsive.</p></div>	<div data-bbox="716 1261 1399 2007"><p>SS of mobile</p></div>

Usability		
U1.0	The system should be intuitive and easy to follow for new users.	<p>User opinions from 15 testers unaffiliated with the website were taken and the results were shown to be largely positive with regards to the intuitiveness of the website.</p> <p>For more information refer to the user testing section.</p>
Security		
SE1.0	The user's password should be encrypted.	Bcrypt was used to encrypt the password, only the hash is stored in the database
SE2.0	The system should allow the user to remain logged in if the user has the access token.	Login session persists even after refresh
Scalability		
SC1.0	The system should be able to tolerate a high user capacity of 50 users.	Refer to Load Testing Section

## 4. Developer documentation

### Frontend Architecture

The frontend is a Single Page Application which uses the React library. The architecture for the frontend is illustrated in the diagram below. Our group chose to follow a layered architecture for the frontend, similar to Model View Controller (MVC). The presentation layer contains the User Interface, while the infrastructure layer handles http requests to the backend. However, we also used an object oriented approach to create domain models of the entities within the controller. Using the information hiding principle, only the controller has access to the domain, where entities are created. There is a controller for each of the three segments of our application, namely the Profile, Flashcard and Game.



### Design Patterns / Principles in Frontend

The factory design pattern is used by the domain. During the creation of an entity, the validation is done in the create method. Should the entity fail any validation, an error will be thrown by the create method. This allows the controller to pass the error to the presentation layer, where both components are not exposed to the implementation of the validation process, allowing the abstraction principle to be followed.

This is a suitable design pattern to use since it follows the separation of concerns, since only the entities should be responsible for the creation and validation process. The sequence diagram illustrates in greater detail an example of the create method of the flashcard class. (TO ADD)

The frontend also adheres to the DRY principle. To reduce code duplication, similar pages such as create flashcard and edit flashcard where the UI is roughly the same are implemented in the same file, with only minor differences in logic.

The game frontend also utilizes event based messaging using sockets to communicate with the backend as well as with other players. Using multiple events for each use case such as a "Match

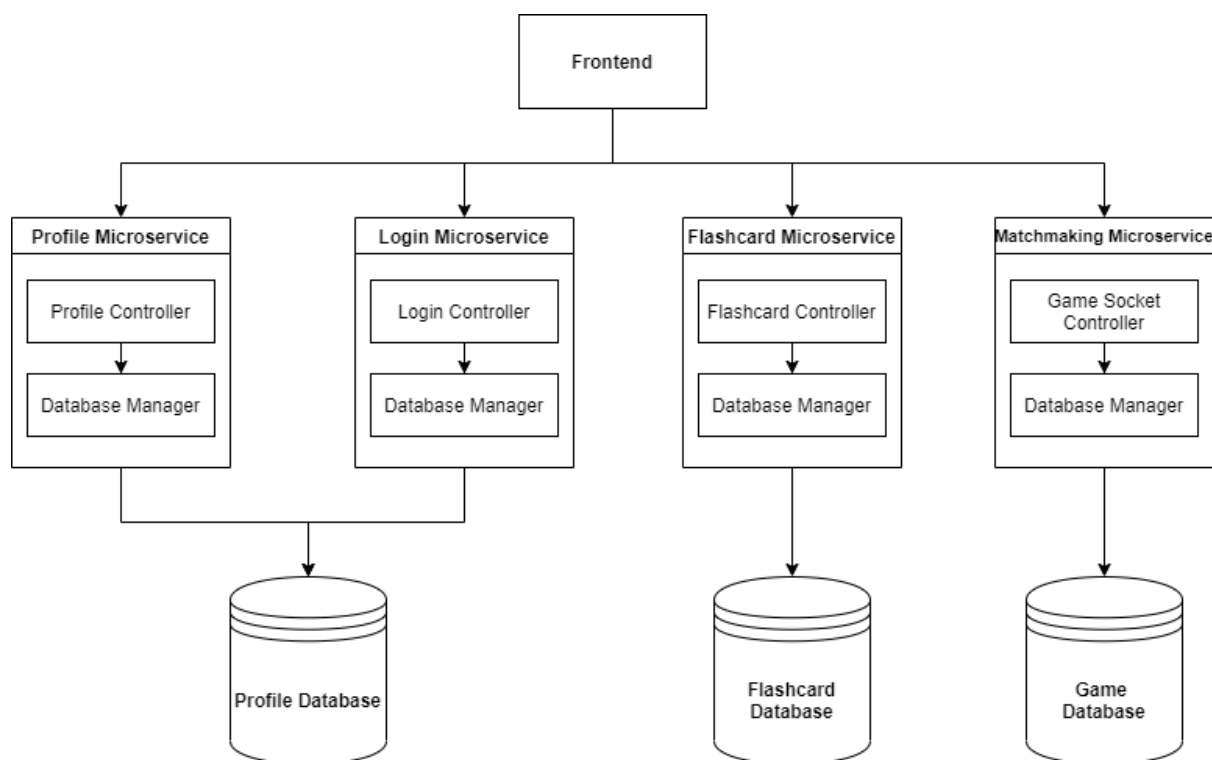


Player” event for matching players and an “answer” event to handle the user answering questions in the frontend helped us handle the various game components effectively as well as handle erroneous situations such as when a player disconnects.

For security, the frontend stores its access and refresh tokens in local storage. While on paper it looks insecure because of how easy it is for hackers to try and steal the tokens, however, we also implemented refresh token rotation. With the implementation of refresh token rotation, when the user wants to enter a private route, will exchange its refresh token with a brand new refresh token. This process will validate the refresh token and if it is valid, will exchange it with a new refresh token, while simultaneously invalidating the old refresh token. Otherwise, the user will be deemed as unauthorized and will be redirected back to the landing page. This process helps to bump up security as hackers only have a limited timeframe to try and steal the user’s access and refresh tokens, as it is constantly being updated and refreshed, which is very difficult to accomplish.

## Backend Architecture

The backend uses the microservices architecture.



The backend made use of microservices architecture with each microservice handling a specific responsibility. Each microservice in general has the same internal architecture as well with a controller handling the frontend requests and then communicating these requests to the database manager which in turn handles interactions with the respective databases.

The only exception to this internal microservice architecture is the game microservice as it uses event based messaging with sockets to communicate with the frontend as well as with other

user sockets. The reason for choosing sockets to handle the game microservice communication was the ease with which we could set up rooms for the games as well as handle user interactions with the backend as well as each other. The usage of socket rooms allowed for easy matching of players and provided the ability to broadcast the same questions simultaneously to both users matched with each other.

The profile, flashcard and game microservice all require for the authentication of the frontend requests using an access token before providing any access to the data. Refresh tokens are also implemented to allow for silent refresh which allows the user to fetch for new access tokens after their old access tokens expired. This enables the user to remain logged in and remove the constant need to re-authenticate themselves while using our website, allowing for a smoother user experience. Also, with any authenticated and authorized request sent to the backend server, the user's refresh token gets renewed in a process called 'Refresh token rotation'. This process will invalidate the user's old refresh token while also returning a brand new refresh token to the user. This reduces the threat of an illegitimate access if the refresh token was ever to be compromised. All these security practices were put in place to boost security and ensure the privacy of the user is not exposed. Any unauthenticated requests are rejected with a 401 unauthenticated error message, while any unauthorized requests are rejected with a 403 unauthorized error message, being sent to the frontend .

In general any errors caused either during connecting to the database or while communicating with the database are communicated to the frontend using a 400 error message.

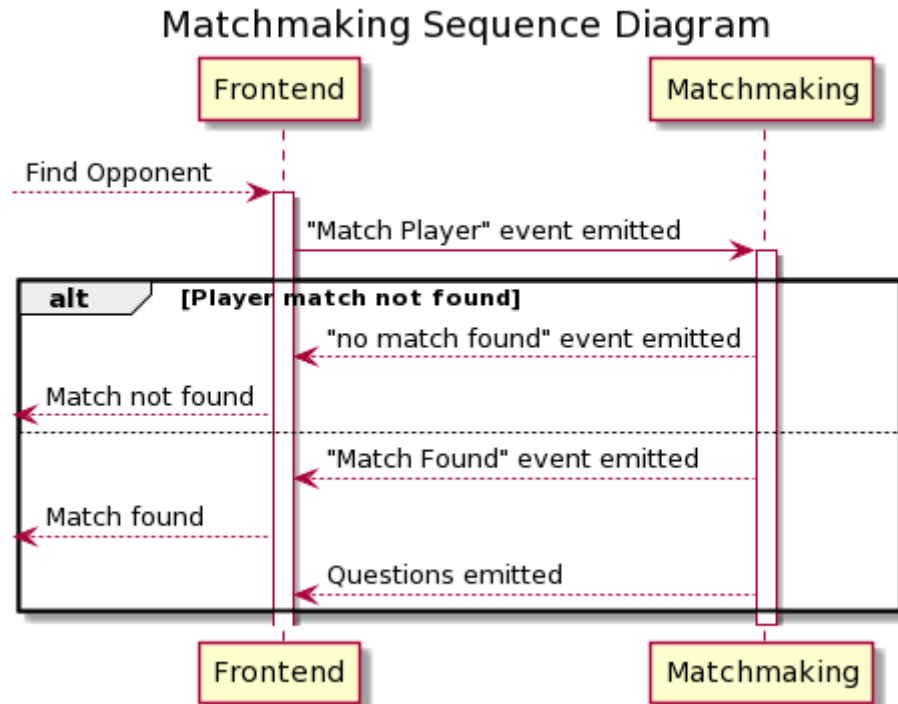
## **Kubernetes**

The backend is running on Google Kubernetes Engine's servers, where each microservice is deployed as a service. The service acts as a Load Balancer and assigns a permanent IP such that the frontend can communicate to the pods without knowing its actual IP, as the pods are ephemeral and its IP address will change once it is terminated. Google Kubernetes Engine has inbuilt support for Horizontal Pod Autoscaling, where the number of pods will increase. This ensures that even with a high number of users, the backend can continue to serve HTTP requests by creating more pods.

## **Game Algorithm**

The game and matchmaking are handled from the same microservice due to the ease of handling. The matchmaking algorithm simply makes use of recursion and timer synchronization to match players of similar ratings and languages with each other. The algorithm is explained in a step form below for easier understanding:

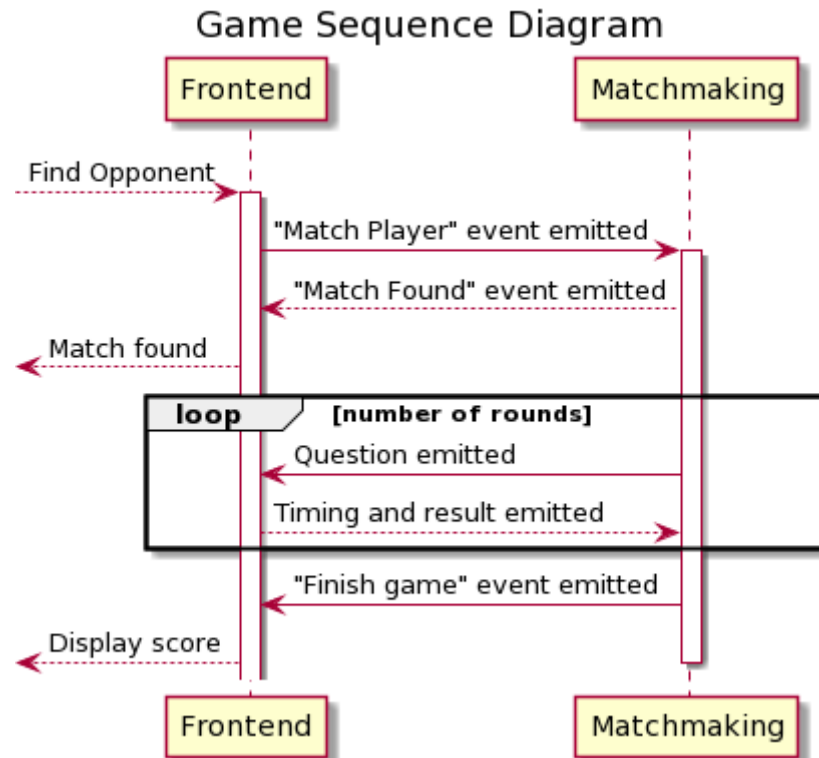
1. Player is added to a queue of players
2. Every 5 seconds, a linear scan is done of all the players in the queue to find an optimal match.
3. If no match is found, the process is repeated up to a maximum of 5 times
4. If a match is found, both matched players are removed from the queue and join a common room for themselves to play the game.



*High Level Sequence Diagram for matchmaking*

The game itself takes advantage of promises to ensure that players adhere to the timing of 20 seconds per question. The questions are initially transmitted using socket rooms to both the players simultaneously and from then on each player is individually passed the question depending on their speed of answering. The formula for calculating the player's score for a round is given by:  $Round\_Result \times \frac{(Time\ per\ round - Time\ taken)}{Time\ per\ round} \times 5$ . The player scores for each round are calculated using their accuracy and speed. The speed of answering a question is calculated in the frontend and emitted to the backend for computation of the score. When both players are done playing the game, the winner is crowned and their rankings are updated.

1. A player make a request for playing the game from the frontend triggering a "Match Player" event
2. The matchmaking microservice adds the player to a queue of players and starts a timer for them. In the event that a match is found, a "Match found" event is emitted to the players.
3. After successful matching, questions are emitted to the players depending on the speed of their answers.
4. Once players are done, a "Finish game" event is emitted to both players and their scores are displayed.



*High Level Sequence Diagram for the game*

#### Design Decisions

Criteria	Microservice Architecture for backend	Monolithic Architecture for backend
Ease of implementation	Implementing multiple microservices and integrating them with the frontend takes more organization and planning increasing the complexity of implementation	Monolithic architecture allows for easier implementation as there is no explicit need to structure the backend services.
Maintainability/Error Resiliency	<p>Using microservices improves maintainability as each isolated service can be improved/updated/fixed without impacting the other microservices.</p> <p>This improves the crash resiliency of the backend.</p> <p>Each microservice can be independently scaled to handle the capacity for each of</p>	<p>Since, all the services are in one backend, they have higher coupling as compared with microservice architecture.</p> <p>Any fatal crashes in any of the services can directly impact the others even though they shouldn't.</p> <p>The backend service has to be scaled as a whole together where some of its resources are</p>

	its own microservice.	not efficiently utilised.
Extendability	<p>Each microservice is an individual component which makes adding on functionality to each microservice relatively straightforward.</p> <p>Adding new microservices as well are also easy given that the overarching structure is already set up.</p>	<p>In monolithic architecture due to a higher degree of coupling, extendability is also reduced.</p> <p>Adding any new functionality to a service may directly affect other services thus, requiring greater care when adding functionality to any service or even adding new services.</p>

Microservices architecture was chosen by our team as we felt that even with the increased complexity during implementation that it brings, it provides us with the freedom to be more flexible in choosing extensions to our project as well as in handling erroneous situations.

### Game Implementation

Criteria	Using socket namespaces	Using socket rooms
Ease of implementation	Dynamic socket namespaces are harder to implement, especially since none of us have worked with socket namespaces in the past.	Socket rooms are an intuitive concept and extremely easy to implement as all they need is just mentioning a room name.
Ease of synchronization	Namespaces increase the ease with which the two players can communicate with each other since it is possible to listen to all events inside a particular namespace.	Socket rooms have a flaw where the socket server cannot listen to events from a socket room hence, requiring a more complex system to ensure communication between players.

Due to time constraints and familiarity, our team chose to use socket rooms to implement the game. The added complexity of synchronization was offset by our increased familiarity with using socket rooms. Given more time, socket namespaces would have been adopted for the game.

## Testing

All of the tests created are run using GitHub Actions to ensure that each commit passes. We do this by creating a **node.js.yml** file which github uses to run our tests.

### Testing Strategy:

1. Each feature must have at least one positive and one negative test case testing its accuracy and resilience.

2. Each developer creates tests based on the features implemented by them in the week. For example: the person implementing add flashcard functionality is also in charge of testing this feature out.
3. All existing test cases are run in addition to the test cases added in to ensure that adding in the new feature does not cause existing features to behave unexpectedly.
4. System tests for example the load testing of kubernetes clusters are done as a group - since it's unfeasible to expect one person to write all these tests.

### **Unit Test**

Backend testing makes use of Mocha and supertest for testing the individual microservices. Each of the microservice routes has been tested for correctness and resiliency to erroneous requests. The unit tests for each microservice request consist of at least 1 positive and 1 negative test case. Testing for sockets in the game microservice was performed using the k6.io library.

### **Frontend Test**

Frontend testing is done using cypress where cypress allows for fast and reliable frontend simulation testing. Numerous tests were done to make sure that the outcome for each of the user's actions was as intended. We focused mainly on the components that utilise React's states and made sure that it was working as intended. Error handling in the frontend was also tested.

### **Socket Test**

Socket testing was done with Jest and involved testing different socket events to ensure that the client and server sockets communicate with each other as expected. The added complexity of synchronizing both the frontend and backend meant that socket testing did not cover all socket events but rather only tested them in isolation to ensure that each event worked as intended on its own. This is one area of testing where improvements can be made on integration testing.

### **Regression test**

Our testing strategy in essence revolves around regression testing. Any new code added in must pass the previous test cases before it can be committed into the master branch. Using Github actions, we ensured that this was the case.

### **Manual testing**

Manually testing out features for bugs proved to be extremely effective for our team especially when we asked our friends to help us try out the website. It helped us uncover bugs we never noticed before such as the game not working when the user requested to play again. User/manual testing of our website was a key part of our testing strategy.

### **User testing**

In order to gain a better understanding of what we needed to work on, we undertook user testing. Over the course of the project, we interviewed people and asked for their opinions on our website. The survey sample was small given our time constraints but the feedback we received was crucial in our development process. The survey results are summarized below:

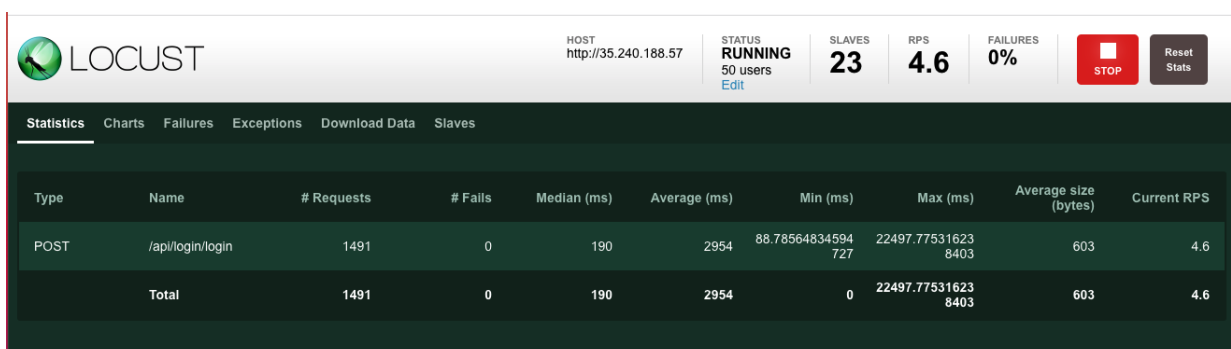
1. 87% of users responded positively to our UI
2. Similar proportion of users responded positively to the flashcards feature
3. 76% of users said they liked the game



This is a word cloud of the feedback we received from users. The main improvements suggested were adding more functionality to the game making it more like kahoot and adding more languages.

## Load test

To test our kubernetes horizontal auto pod scaler load capacity we did load testing using Locust. Locust simulates test users to send HTTP requests to the backend microservice. For our test, we tested the login POST API call. We simulated 50 users using Locust, and the median request time for Login is fast at only 190ms. This proves SC1.0, that our system can handle 50 concurrent users.



## Deployment

The backend microservices were all first Dockerized and then orchestrated with Kubernetes. After setting up Kubernetes locally, the next step was for us to use Github Actions to deploy passing builds onto Google Kubernetes Engine. This helped us with CI and CD. Each microservice is deployed using a different Actions job, which allows them to be independently deployed from

one another. A docker image is created and uploaded to Google's Container Registry for each job, which ensures CD. Each microservice uses GKE's load balancer to retrieve an IP address

## **5. Suggestions for improvements and enhancements**

While we believe that the UI of our website is highly intuitive as mentioned in our project requirements and validated by the user testing, there is scope for improvement especially when it comes to the game. One improvement that could be made for the game UI especially would be to make it more like kahoot so that users remain interested in it for long periods of time.

Our project, while already boasting a fully functional game and flashcard creation system, would be greatly improved with the below mentioned enhancements. The main improvements we see as necessary in the future based on user testing and personal experience are as follows:

1. Support for full sentences as well as for a larger range of languages. In the future, we do envision LanguageLearners as a more dynamic website where users can choose any language and interact with like minded users.
2. Ability to share flashcards with other users
3. Forum system where people can interact with other users and try to gain insights about the language they are trying to learn (stackoverflow).
4. Automatic translation for inputted words/sentences and suggestions on spelling or grammar.



## 6. Reflections and learning points

### Reflection

Our main takeaway from the project was that good planning is essential for development of any viable software product. Our initial proposal was too ambitious given the time constraints and our other commitments meaning that we had to prioritize which requirements to fulfill. This was where our initial planning came into use as we had a clear idea as to which requirements were more important than others and hence, were able to efficiently choose or prune functional requirements from our final product.

Another takeaway was the allocation of work amongst group members and our resolution of issues which propped up especially while deploying. We chose to split the requirements into smaller tasks and assigned these tasks in an equitable manner to all the group members. However, the reality is that some tasks do tend to be a lot harder than others for example setting up the access token system for login. Hence, a more dynamic issue allocation system was adopted where if a group member was stuck on a task for too long, another task was allocated to him and the whole group took turns trying to finish the task. This system worked wonders for us as it reduced downtime due to harder tasks.

### Learning Points

1. Continuous integration/deployment using github actions and gke:
  - The backend was deployed on Google Kubernetes Engine's servers, where each microservice is deployed as a service. This was actually one of the hardest parts of the project due to the numerous failed attempts at setting up the CD. Even the CI on Github actions made us understand the difference between testing something locally and remotely. All in all the CI/CD proved to be some of the most important places where we learnt.
2. Backend and frontend architecture
  - Code architecture is an extremely important part of planning. Our code was based upon the Model View Controller design pattern. This made it relatively easy to separate the files and code such that anyone implementing any component would know where to find the respective components or which folder to implement a new component at. This structure was extremely important in helping maintain the code base neat and working in parallel.
3. Communication using sockets
  - Socket architecture is different from the normal client-server architecture used in most of our previous projects. Using sockets for the game microservice required thinking in terms of how sockets would communicate with each other (rooms/namespaces) rather than just with the frontend which was the requirement for the other microservices. Sockets while flexible come with their own constraints such as the lack of synchronization as well as the inefficient run time due to the fact that they are constantly listening for events.
4. Using kubernetes as an orchestration service
  - Our collective knowledge of using kubernetes came from Task A of the OTOT tasks. We extended this knowledge to place our backend in a kubernetes cluster

with a horizontal pod scaler. This of course came with its own challenges and was a great learning experience in the end.

5. Agile software development

- The project deadlines and requirements alongside our other commitments meant that the requirements priorities changed fluidly. To ensure that the entire team was on the same page regarding all these changes, we held almost daily discussions regarding the project progress and weekly meetings to consolidate all the information. Our software development workflow hence, most resembled that of Agile something which most of the team used in their internships. This allowed for a smooth transition for the project development.