Team 36
WordTree

Team Members:

CHIAM YUN QING	A0204978R
JUSTIN GNOH KIT PEOW	A0202054Y
TAN BOON JI	A0200362Y
UDAYAGIRI NIKHILA SAI	A0207967N

Table of Contents

Contributions

Name	Technical Contributions	Non-Technical Contributions
CHIAM YUN QING	Implement essay and word microservices, Implement k8s integration	Requirements Documentation, Project Report
JUSTIN GNOH KIT PEOW	Set up frontend, Implement frontend integration with the microservices	Requirements Documentation, Webapp Design, Project Report
TAN BOON JI	Implement community, nut, user and notification microservices	Requirements Documentation, Project Report
UDAYAGIRI NIKHILA SAI	Implement auth and challenge microservices	Requirements Documentation, Project Report

Tech Stack

Frontend	React
Backend	Express.js/Node.js
Database	PostgreSQL
Orchestration Service	Docker/ Kubernetes
Project Management Tools	GitHub Issues

1. Project Description

The current Covid situation has caused many kids to be deprived of social interaction in schools due to reduced activities in school. As such, they may find it harder to make friends. Moreover, the Singapore education leaves little room for kids to apply their creative writing skills and we believe that there is a solution for kids to have an interactive and engaging outlet online. Hence, we created WordTree, a multiplayer turn-based descriptive/creative writing platform for primary school students.

WordTree focuses on the creative writing aspect of the English language. Users take turns to collaboratively write an essay to create a story that both users can call their own. Once a user has created a WordTree account, they can specify their topics of interest and be matched with users of similar interests. A challenge is created for each pair of users. Word prompts are shown in the challenge and users who use the word prompts in their essay will be rewarded points, which we call Nuts. For each turn they complete and for each word they use in their essay, they will receive a nut. Once a pair completes an essay after a preset number of turns, their completed story will go up onto the community page for other users to read and upvote.

2. Functional Requirements

We have grouped our functional requirements according to the services that WordTree is expected to provide. The services are broadly categorised into User Authentication, Profiling, Matching of Partners, Turn-Based Solving, Vocabulary Bank and Community Page. We also prioritised the requirements accordingly based on the microservices which is elaborated in 7. Microservices.

2.1 User Authentication

No.	Functional Requirement	Priority
F1.1	Users can create a new account.	High as it forms the minimal working system.
F1.2	Users can login with valid credentials.	
F1.3	Users can logout from their account.	

No.	Functional Requirement	Priority
F1.4	Users can only access resources (e.g. profile page, acceptance of challenge) authorized for their accounts.	

2.2 Profiling

No.	Functional Requirement	Priority
F2.1	Users can create one profile per account.	Low, as it can be deemed as optional extensions to the system.
F2.2	Users can update their profile.	
F2.3	Users can input their basic personal details (e.g. age, school, gender).	
F2.4	Users can indicate their interests for matching.	

2.3 Choosing Partners

No.	Functional Requirement	Priority
F3.1	Users can initiate a match.	High as it forms the minimal working system.
F3.2	Users can choose the numbers of turns they intend to have and the word limit per turn in the match.	
F3.3	Users can accept a match request posted by other users.	
F3.4	Users can see the match details when accepting the match (e.g. other user's name, the topic and number of turns).	

2.4 Turn Based Solving

No.	Functional Requirement	Priority
No.	Functional Requirement	Priority

No.	Functional Requirement	Priority
F4.1	Users can type in their story in a textbox within the pre-decided word character limit.	High as it forms the minimal working system.
F4.2	Users can earn nuts (i.e. scores) when they finish a challenge based on the number of turns they completed in the challenge.	
F4.3	*Users shall get a notification in the application when their turn is up next. (**Unfortunately, we have yet to be integrate this to frontend due to time constraints but the backend is already implemented.)*	

2.5 Vocabulary Bank

No.	Functional Requirement	Priority
F5.1	Users should be able to see the 3 words from the vocabulary bank given at each turn that they should use.	Medium, it is an important extension to WordTree but the lack of it will not affect the functionality of the system.
F5.2	Users should be able to see real-time status of the 3 words if they have used the word in their essay.	
F5.3	Users would be prompted on the number of nuts (i.e. scores) that they have collected for the turn, according to the number of words that they have used currently.	

2.6 Community Page

No.	Functional Requirement	Priority
F6.1	All public visitors can view the completed essays of other users.	Low, as it can be deemed as optional extensions to the

F6.2	All WordTree users can upvote the completed essays of other users.	system.

3. Non-Functional Requirements

3.1 NFR

We have identified three main non-functional requirements, in order of their priorities, as follows:
NFR 1. [Browser Compatibility] WordTree should be supported by common browsers, including IE, Chrome.
NFR 2. [Security] WordTree should only be used by authenticated users with a valid account.
NFR 3. [Scalability] WordTree should support more than 50 concurrent users.

NFR 1 is of the highest priority because it is the determining factor as to who can use WordTree. It specifies an optimal platform for WordTree. Otherwise, it is hard to attract users to our platform. Many young kids typically have access to these common browsers.
This is followed by NFR 2 which protects the state of the game for each user, allowing users to have a smooth user experience on our platform. NFR 2 also only permits users to view challenges tagged to their account, making it a private connection between two users. We provide a more detailed explanation on how we achieve NFR 2 in 3.2 Security Requirement. NFR 3 is important in microservices architecture but has the lowest priority out of the 3 identified NFRs as our main goal is to attract people to use our application first before scaling the application to accommodate even more users. We provide a more detailed explanation on how NFR 3 is achieved in 3.3 Scalability Requirement.

3.2 Security Requirement

WordTree should only be used by authenticated users with a valid account.
As WordTree is a platform where users can create content (i.e. the essays they write), WordTree needs to be secure and only allows authenticated users to write essays through their accounts. We used Firebase Authentication to authenticate users on the frontend and retrieve an access token which is passed to the backend for verification before any response is returned.

The turn-based challenge encapsulates the complicated logic as to which user can input their essay paragraphs during which point in time of the challenge. Moreover, players not part of the challenge should not be able to add to the challenge as this may ruin the essay that those involved in the challenge is building. Hence it is important that all these checks are done before returning the relevant responses. Checks as to whether the user is authenticated is done by the Auth Server, explained in 7.1 Auth Service. Checks as to whether the user sending the request to contribute to the challenge is done in Challenge service as this is highly related to the data of the challenge. It is explained in more detail in 7.6 Challenge Service.

3.3 Scalability Requirement

WordTree should support more than 50 concurrent users.
To achieve scalability for our backend services, we made use of Kubernetes horizontal pod auto-scaler to scale up the number of pods for the backend services when high load is encountered. We provide screenshots from a demonstration of the scaling of more specifically, the User microservice, in action as below.

Note: The k8s cluster is run locally so our configuration standards are much lower to prevent overheating of our laptops. For the purpose of this demonstration, we only have a maximum of 3 replicas and the pods scale with a utilisation of only 5%. In actual production, the figures should be set appropriately, for example, there should be up to 5 replicas with a utilisation of 70% before scaling. There should also be at least 2 replicas per deployment to ensure availability of the microservices.

An example of part of the HPA file for k8s is shown in Fig 1. *(Note: Actual production should set averageUtilisation to 70% instead of just 5% in this demo.)*

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  namespace: wordtree
  name: user-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: user-deployment
  minReplicas: 1
  maxReplicas: 3
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 5
```

Fig 1. HPA file for User Microservice Deployment

Before Load Test:

We have only 1 pod for each microservice initially. (*Note: Actual production should have at least 2 pods per microservice to achieve availability.*)

```
Yuns-MacBook-Air:kubernetes yunqing$ kubectl get pods -n wordtree
NAME                                      READY    STATUS    RESTARTS    AGE
auth-deployment-74fc94496c-krtjn          1/1      Running   0           8m51s
challenge-deployment-fbbfcf976-c7pn8      1/1      Running   0           8m52s
community-deployment-5f74b5c5dc-lrcxp     1/1      Running   0           8m51s
essay-deployment-f4cb66959-5s82d          1/1      Running   0           8m52s
notification-deployment-9bf6d7679-n9h82   1/1      Running   0           8m51s
nut-deployment-6675885b8c-7zcd5           1/1      Running   0           8m52s
user-deployment-d4fd68c9b-rkprx           1/1      Running   0           8m52s
word-deployment-58bd5f587b-kxff6          1/1      Running   0           8m52s
```
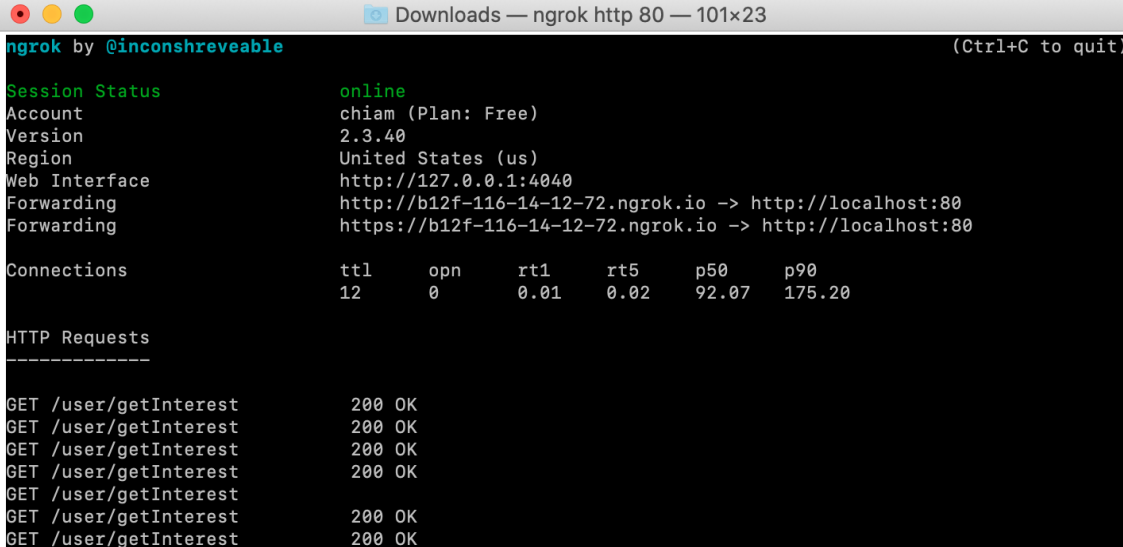
Fig 2. Only 1 Pod Per Microservice Before Auto-Scaling

Load Test:

For this load test, we focus on demonstrating the scaling of the User microservice pod. We test with the endpoint /user/getInterest which routes to the User microservice on our backend routes.

We first set up the IP Address with Ngrok to route to our API gateway at http://localhost:80 so that we can load test with another pod using busybox.

Fig 3. Using Ngrok to create an IP address to Ping In Another Pod

We then create a pod to load test using the busybox image which pings the end point http://<ngrok-generated-link>/user/getInterest infinitely to load test User microservice.

Fig 4. Load Test By Calling The Endpoint Infinitely

After Load Test:

After a few minutes, the load exceeds the utilisation for which the pods will scale and the autoscaler will resize accordingly as shown in Fig 5.

```
Name:                                              user-hpa
Namespace:                                         wordtree
Labels:                                            <none>
Annotations:                                       <none>
CreationTimestamp:                                 Tue, 09 Nov 2021 09:29:01 +0800
Reference:                                         Deployment/user-deployment
Metrics:                                           ( current / target )
  resource cpu on pods  (as a percentage of request):  6% (12m) / 5%
Min replicas:                                      1
Max replicas:                                      3
Deployment pods:                                   1 current / 2 desired
Conditions:
  Type            Status   Reason              Message
  ----            ------   ------              -------
  AbleToScale     True     SucceededRescale    the HPA controller was able to update the target sca
le to 2
  ScalingActive   True     ValidMetricFound    the HPA was able to successfully calculate a replica
 count from cpu resource utilization (percentage of request)
  ScalingLimited  False    DesiredWithinRange  the desired count is within the acceptable range
Events:
  Type     Reason            Age    From                         Message
  ----     ------            ----   ----                         -------
  Normal   SuccessfulRescale  15s   horizontal-pod-autoscaler    New size: 2; reason: cpu resource ut
ilization (percentage of request) above target
```

Fig 5. Successful Rescale as the Utilisation of CPU is Exceeded

5. Architecture Design

5.1 Architecture Diagram

We followed the microservices architecture for this project using kubernetes (k8s) on Docker Desktop Application as the orchestration service. The backend services have been split into 8 services that help facilitate the main functionality of WordTree. The API gateway and backend services reside in the Kubernetes cluster while the data storage is deployed on ElephantSQL.
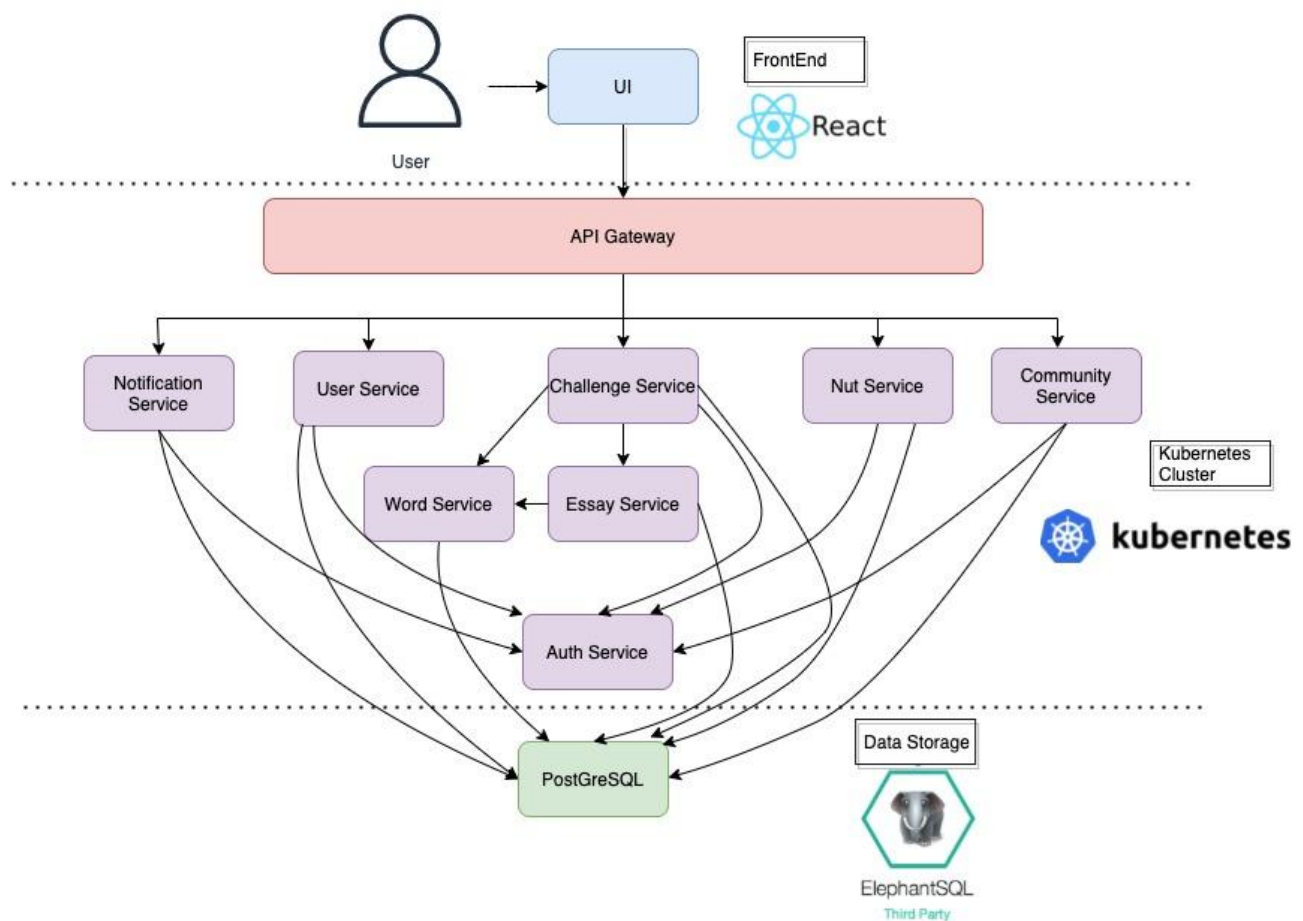

Fig 6. High Level Architecture Diagram for WordTree

Each backend service has a k8s Service resource configured so that the API gateway can redirect the requests from the frontend to the relevant microservices. On k8s, the API gateway is implemented using an NGINX ingress controller. The usage of Service resources also allows us

14

to make use of the in-built coreDNS service for service discovery. Furthermore, communication between microservices simply relies on the configuration of the Service resource (e.g. To communicate with Word Service via HTTP, one just needs to call http://word-service:8080 without having to know about the actual IP address). K8s also allows us to achieve scalability in microservices architecture using the horizontal pod auto-scaler as was discussed in 3.3 Scalability Requirement.

5.2 Architecture Decisions

We present below some of the decisions that were made with regards to the architecture for this project. A design consideration that we made but not is mentioned in this section is the use of Shared Database which will be discussed in 6.2 Shared Database Pattern.

Decision 1: Overall Architecture		
Criteria	**Design 1: Microservices Architecture**	**Design 2: Monolith Architecture**
Ease of Implementation	Low initially given the new technologies that need to be picked up by the team to implement the architecture properly but high at the end due to low coupling of microservices	High, no new technologies (i.e. k8s) need to be adapted
Ease of splitting workload	High, new functionalities can be introduced easily by including more microservices, which can be implemented by different members	Low, it may be difficult for different members to work on the project due to tightly coupled modules
Final Decision: Design 1 is preferred due to its benefits from low coupling and scalability which allows different members to work on different components easily. The long term benefits of being able to extend the project more easily outweighs the steep learning curve that may be encountered initially.		

Decision 2: Database of Application		
Criteria	**Design 1: Deploy Stateful Sets on k8s for database**	**Design 2: Use an external database that resides outside of the k8s cluster**
Local Storage Space	High, as we are running the k8s cluster locally	Low, as the database is hosted on the cloud
Ease of Management	Low, since we are not deploying our application on the cloud, any updates to the schemas of our database need to be manually configured by each member who are testing the endpoints	High, as it is a shared database, any updates to the schema just need to be run once on the cloud
Final Decision: Design 2 is preferred because of the benefits of a fully-managed database by external providers. This makes it easier for us to test and update the schemas directly while working on different microservices in parallel.		

Decision 3: Communication between Microservices		
Criteria	**Design 1: Pub Sub Channel**	**Design 2: HTTP requests**
Relevance	For the scope of our project, most of our messaging is one to one, 1 service needs to send the message to only another service. Hence we cannot take advantage of the pub sub model where multiple publishers can send messages to the same topic or	HTTP requests are more suitable to our scope of the project since it sends from 1 service to another specific service.

	multiple subscribers can receive messages from the same topic.	
Asynchronous	For some of our use cases we needed synchronous communication. However, pub-sub messaging pattern is generally asynchronous.	HTTP requests will send a response upon receiving the request and it provides a synchronous behaviour which is needed as services might need to act on the response received back.
Extensibility	Pub-sub model is easy to extend to new services If another service is added which requires the same message, they can subscribe to the topic and receive the same messages.	For HTTP requests, may not be as extensible. If another service is added which requires the same message, another HTTP request needs to be sent to the new service.
Final Decision: Design 2 is preferred because it is more suitable to our scope of the project.		

6. Design Patterns

6.1 Data Access Object Pattern

In each of the microservices, we employed the use of Data Access Object (DAO) pattern so as to abstract out the interactions with the database on ElephantSQL. Fig 7 shows an example of the implementation of WordDAO in the Word microservice.

```javascript
import pg from 'pg';
import { POSTGRES_URL } from '../config/index.js';
const Pool = pg.Pool;
const pool = new Pool({
  connectionString: POSTGRES_URL
});

//function returns a single array of 3 words
export async function getWordsForTurn(challengeid, seqnum) {
  //Note: psql array is 1-indexing
  try {
    const wordarr = await pool.query("SELECT word_list FROM WordsPerChallenge WHERE challenge_id = $1 and seq_num = $2;", [challengeid, seqnum]);
    return wordarr.rows[0]["word_list"];
  } catch (err) {
    throw err;
  }
}

export async function insertWordList(challengeid, seqnum, wordarr) {
  try {
    await pool.query("INSERT INTO WordsPerChallenge(challenge_id, seq_num, word_list) VALUES($1, $2, $3);", [challengeid, seqnum, wordarr]);
  } catch (err) {
    console.log(err);
    throw err;
  }
}
```

Fig 7. WordDAO Implementation

The WordDAO is in charge of directly interacting with the database and performs create (insertWordList) and read (getWordsForTurn) operations.

6.2 Shared Database Pattern

WordTree uses the shared database pattern, whereby all the microservices share the same database. We considered the alternative of database per service pattern but we feel that given the scale of our project, there is no need for such complications.

Furthermore, our schemas are relational. For example, challenge table reference users table and essay table references challenge table. Hence, it makes more sense to use the same database for the schemas. However, in our implementation, due effort was made to ensure that each microservices does not directly access the tables that do not belong in their control.

6.3 Publish-Subscribe Pattern

WordTree website real-time notification system uses Socket.IO, which is an implementation of the pub-sub pattern in order to enable real-time notifications to be sent to the users who are online on the website. More details can be found in 7.8 Notification Service.

7. Microservices

We split the backend services into 8 main components, consisting of 7.1 Auth Service, 7.2 User Service, 7.3 Word Service, 7.4 Nut Service, 7.5 Essay Service, 7.6 Challenge Service, 7.7 Community Service and 7.8 Notification Service.

Within each microservice, we followed the Single Responsibility Principle, whereby each file is only responsible for a specific part of the functionality of the microservice.

```
∨ nut
  > node_modules
  ∨ src
    ∨ config
      JS index.js
    ∨ controllers
      JS NutController.js
    ∨ database
      JS db.js
      JS NutDao.js
    ∨ routes
      JS NutRouter.js
    JS index.js
```
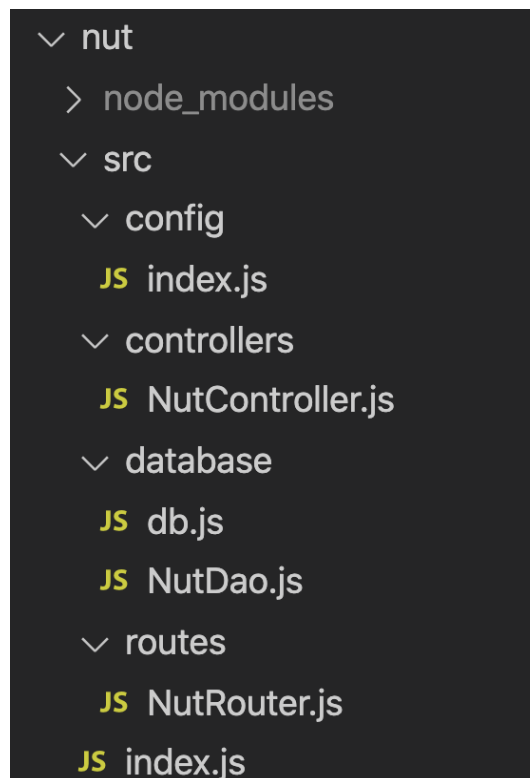
Fig 8. Example of File Structure Within Nut Service

With reference to Fig 8, the functionalities of Nut Service are being split into 3 main components - (1) Routes (in charge of routing within Nut Service), (2) Database (in charge of database

operations within Nut Service and (3) Controllers (in charge of handling the requests and generating the responses accordingly). The Config folder consists of configuration data, such as the link to our ElephantSQL database.

7.1 Auth Service

As mentioned earlier in [3.2 Security Requirement](#), authentication is important in our application so that others do not misuse a player's credentials to input essays as if they were the player. We used Firebase Authentication to authenticate users. The frontend will authenticate users by using Firebase API and retrieve an access token which is sent along any requests to the backend services. Backend services with paths that are routed to by the API gateway will send a request to the Auth service to verify the token. The Auth Service will use the API provided by Firebase to validate the token and get the user ID which is returned in the response to the backend service.

Some backend services with routes that are not accessed by any client are not exposed on the Public IP. These services are used by other services internally and do not need an access token to be accessed.

7.2 User Service

The User Service is responsible for managing all information related to the users of our application, which can be categorized into three components:

- User accounts - Stores information that are only accessible to the user themselves. Currently, only their email account is stored.
- User profiles - Stores information that other users of the application can see. This includes their name, date of birth and joined datetime.
- User interests - Stores the genre of essays that each user is interested in. Each user can have zero or more interests.

However, the User Service does not handle any authentication related actions such as logging in, logging out or changing of password as these are handled by Firebase Authentication. User passwords will be stored and managed by Firebase and not in our database.

When users first create their account, their account and profile information are created in our database through the User service. After they have successfully created their account, they can go on to retrieve and manage their profile and interests through the User service.

Users' nuts are managed separately by the Nut Service, and User Service will retrieve all user's nuts information from the Nut Service. The User Service does not have endpoints to add or remove nuts from users as that is the responsibility of the Nut Service.

We list some API endpoints of User Service as below, more can be found at 12.1 More API endpoints of User Service.

Create user account and profile

HTTP Method	POST
Route	/user/createUser
Service Calling	Frontend
Description of API	Creates the user account and profile information in the database.
Request Header	x-access-token : access token
Request Body	userId : firebase uid email : varchar name : varchar dateOfBirth : date format 'MM/DD/YYYY' or 'YYYY/MM/DD'
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 401 Unauthorized : Invalid access token 403 Forbidden : Access token does not match firebase uid in request body 500 Internal Server Error : An error occurred in User Service

Update the interests of a user

HTTP Method	PUT
Route	/user/updateUserInterest

Service Calling	Frontend
Description of API	Updates the interests of a user by access token.
Request Header	x-access-token : access token
Request Body	interest : [ ] (empty array = no interest) **OR** ['interest'] (only one interest) **OR** ['interest 1', 'interest 2', …] (many interests)
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 401 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in User Service

7.3 Word Service

The Word Service is in charge of generating and maintaining the words to be used by the player for each turn. The generation of words is done with the help of a third-party API and is only done once at the start of each challenge. Hence, when a challenge is initiated, all the words for the challenge turns are already pre-computed. The responsibilities of the Word Service are as follows:

- Interact with Merriam-Webster's Thesaurus API to get the words which are synonyms of the player's interest.
- Populate all the words for that challenge, taking into consideration the number of turns. For each turn, 3 words are generated.
- Shuffle the order of the words so that players with same interests will not always get the same 3 words (due to limitations of using third party API for word generation)

A rough sketch of the sequence diagram for Word Service when a call to populate the word list for a challenge is made is shown in Fig 9.
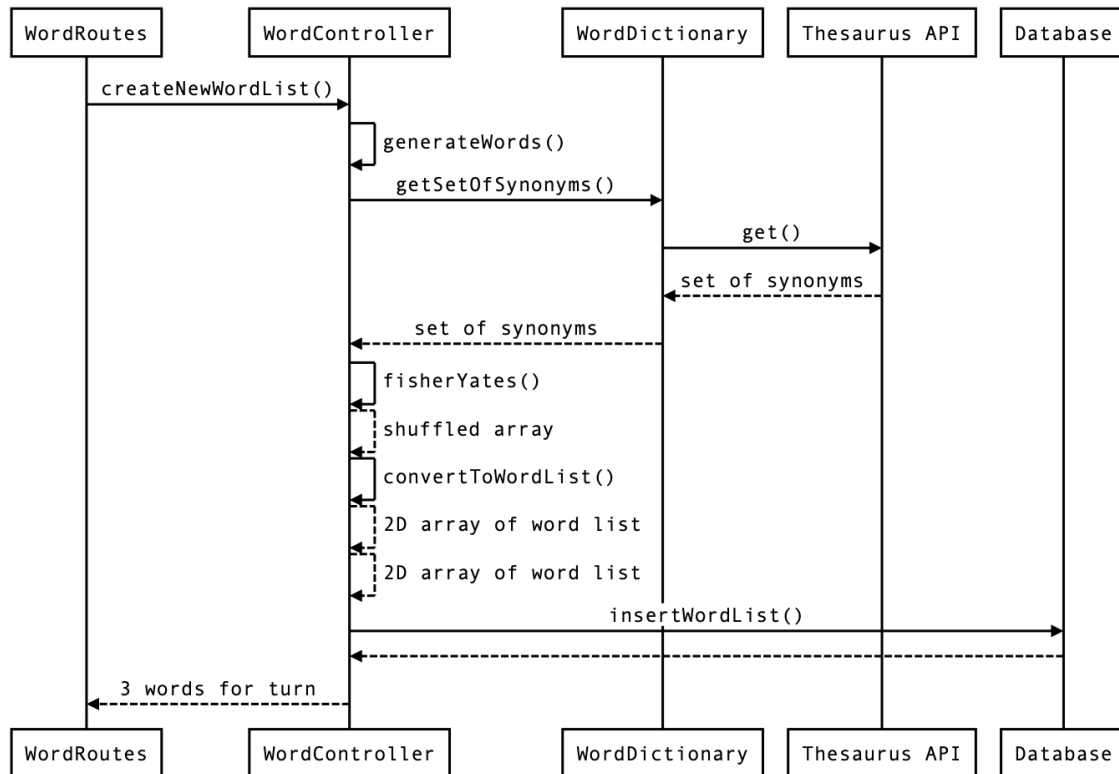
Fig 9. Sequence Diagram To Populate Word List For A Challenge

The API endpoints to Word Service are illustrated below:

Initialise word list

HTTP Method	POST
Route	/wordlist
Service Calling	Challenge Service
Description of API	Generate the words that will be used by players in a challenge.
Request Body	challenge_id : an integer interest : a string num_of_total_turns: an integer
Response	200 Success: An array of 3 words for the first turn. 400 Bad Request

<u>Get words for turn</u>

HTTP Method	GET
Route	/words/<challengeid>/<sequencenum>
Service Calling	Challenge Service, Essay Service
Description of API	Get the 3 words meant for that challenge and sequence.
Request Body	-
Response	200 Success: Response body - An array of 3 words for that sequence and challenge. 400 Bad Request

7.4 Nut Service

The Nut Service handles all the nuts of the users, which is the point system of the application. Nuts are categorised into three different types:

- Essay Nut - These nuts are awarded to users for using the words given when writing their essays. For each word used, one nut is given.
- Community Challenge Nut - These nuts are awarded to users for upvotes on their complete challenge. Users can upvote completed challenges in the community page, and one nut is given for each upvote.
- Community Essay Nut - These nuts are awarded to users for upvotes on their essay paragraphs that they wrote for the challenges. Users can upvote essay paragraphs for challenges in the community page, and one nut is given for each upvote. Note that every challenge will have multiple essay paragraphs written by the squirrel and the raccoon users, and upvotes can be given to individual essay paragraphs separately.

The API endpoints of Nut Service are listed below:

<u>Add essay nut to a user</u>

HTTP Method	POST

Route	/nut/addEssayNut
Service Calling	Essay Service
Description of API	Add essay nuts to the user after each submission of essay paragraphs by the user.
Request Header	-
Request Body	userId : firebase uid nut : integer challengeId : integer seqNum : integer
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 500 Internal Server Error : An error occurred in Nut Service

Delete essay nut of a user

HTTP Method	DELETE
Route	/nut/deleteEssayNut
Service Calling	Essay Service
Description of API	Delete essay nuts of a user. This can be for various reasons, such as improper essays submitted or improper usage of the vocabulary words.
Request Header	-
Request Body	userId : firebase uid challengeId : integer seqNum : integer
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters

	500 Internal Server Error : An error occurred in Nut Service

Get number of nuts of a user

HTTP Method	GET
Route	/nut/getUserNut/<userId>
Service Calling	User Service
Description of API	Gets all the number of nuts (essay nut, community challenge nut and community essay nut) of a user by access token.
Request Header	-
Request Body	-

Response	200 Success : OK
	{
	   totalNut : …,
	   essayNut: …,
	   communityChallengeNut: …,
	   communityEssayNut: …
	}
	500 Internal Server Error : An error occurred in Nut Service

The API endpoints for adding and deleting community challenge nut and community essay nut are similar to the essay nut and can be found in 12.1 More API endpoints of Nut Service.

7.5 Essay Service

The Essay Service is in charge of managing the essays that the players have written for each turn, with the functions of:

- Checking the number of words that the player has used from the provided words
- Keeping track of the ongoing essays for each challenge
- Keeping track of all the completed essays

The architecture diagram for Essay Service is as shown in Fig 10.

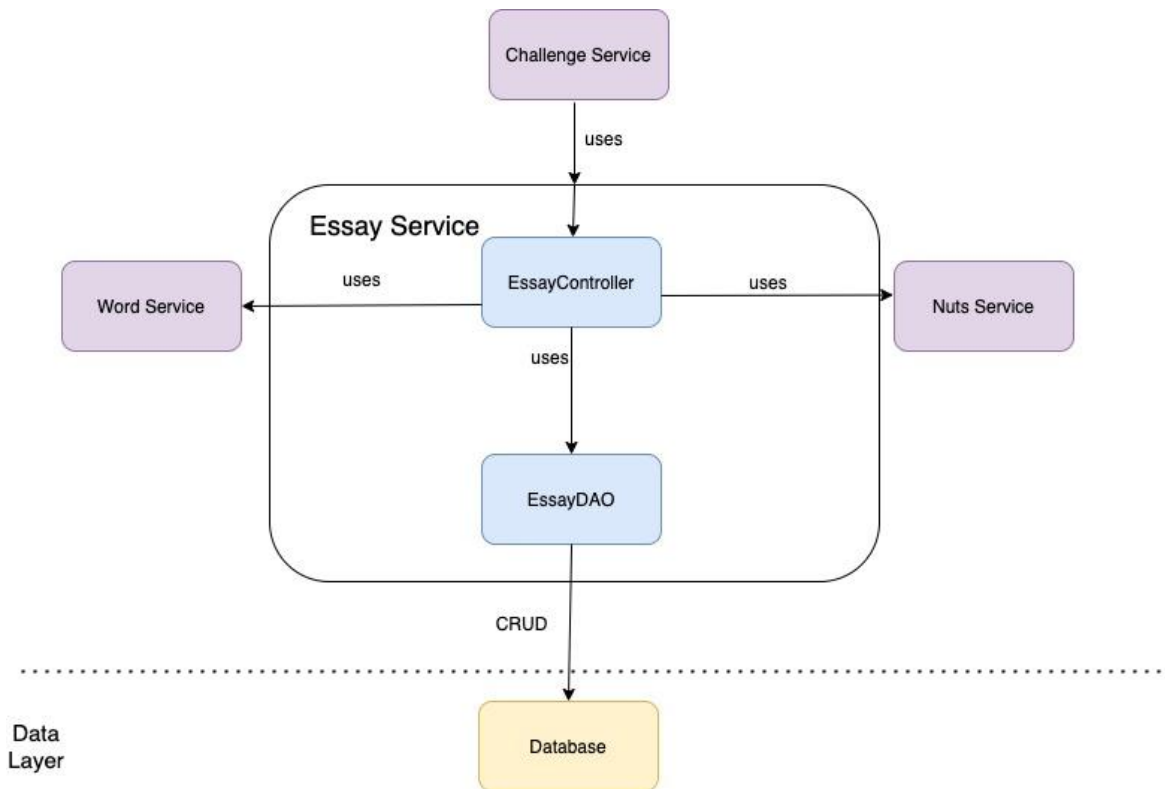*Fig 10. Architecture Diagram For Essay Service*

The API endpoints to Essay Service are illustrated below:

Add new essay paragraph

HTTP Method	POST
Route	/newEssayPara/<challenge id>
Service Calling	Challenge Service

Description of API	Add the new essay paragraph for that challenge
Request Body	author_id : varchar seq_num : an integer essaypara: text
Response	200 Success 400 Bad Request

Get all essay paragraphs for a challenge

HTTP Method	GET
Route	/allEssayPara/<challenge id>
Service Calling	Challenge Service, Community Service
Description of API	Get all the essay paragraphs for a challenge sorted by sequence number
Request Body	-

Response	200 Success: {
	      {seq_num: …, author_id: …, essay_para: …, words_used:...},
	      {seq_num: …, author_id: …, essay_para: …, words_used:...} …
	      }
	
	400 Bad Request

7.6 Challenge Service

The Challenge Service is responsible for maintaining and retrieving data of challenges as well as executing the compute logic of whose turn it is to write the essay paragraph or whether they are even allowed to do so.

Creating a challenge

A challenge allows for 3 customised settings.

1. Number of turns in a challenge
2. Character limit per essay paragraph
3. Interest - the essay topic which the essay should revolve around

This information is sent in the create challenge endpoint as shown below. This will initialise the entry in the challenge table.

HTTP Method	POST
Route	/challenge/
Service calling	Frontend
Description of API	Creates a new challenge
Request Header	x-access-token : token given by firebase from frontend
Request Body	squirrel_id : id of player initiating challenge num_of_total_turns : 4 or 6, word_limit_per_turn : 300 or 500 interest : a valid genre in string (i.e. "crime")
Response Status	400 Bad Request if missing or invalid fields 200 Success
Response Body	First three words for squirrel to start with { "words" : ["one", "two", "three"] }

Adding an Essay paragraph in a challenge

To add an essay paragraph during a challenge, first we need to check that the user is part of the challenge. Then we need to check the number of sequences that have been completed in the challenge. With this information we can check if it is the user's turn. Based on the sequence

number after this turn has been completed, we then change the status of the challenge accordingly. This logic is represented by the activity diagram in Fig 11.
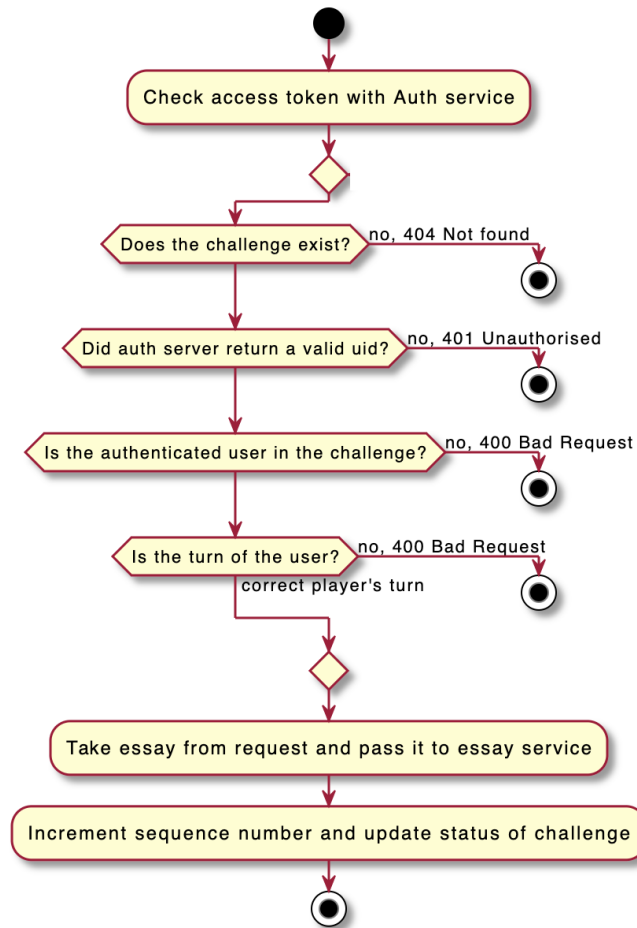
Fig 11. Activity Diagram for Adding Essay Paragraph to a Challenge

The Challenge Service passes the essay paragraph to the Essay Service. The logic of handling essay paragraphs is separated into another service as there are added responsibilities such as adding a nut (score point) to the user and checking if the given words are used. Hence to ensure better decoupling, Essay Service is added, which is explained in 7.5 Essay Service.

Status of challenge

The status of a challenge highly related to the current sequence number. Sequence number refers to the sequence a challenge is in, i.e. how many turns have been completed.

For ease of referring to the players in the challenge, we have come up with this terminology.

- Squirrel: The player who initiates a challenge.
- Raccoon: The player who accepts a challenge that was initiated by a squirrel.

Sequence Number	Status of Challenge	Description
0	DRAFT	The challenge has just been created and squirrel has to submit an essay paragraph before others can accept this challenge.
1	WAITING_MATCH	Other players can see this challenge and accept it if they wish to. Upon accepting, they become the raccoon of the challenge. One cannot accept a challenge they have created, and a challenge cannot be accepted more than one time.
2- 3/5	ONGOING	A challenge can have 4 or 6 turns determined by the squirrel upon creation of the challenge. Any challenge with a sequence number between 2 and the number of turns of the challenge is considered ongoing.
4/6	COMPLETED	A challenge with all its sequences completed would have reached a sequence number of 4/6 and the status of the challenge will be set to completed.

Accepting a challenge

For a challenge to be accepted, it must be first created by a player (known as the squirrel in the challenge) and the squirrel needs to add the first essay paragraph. This is so that the idea of the essay can be determined by the squirrel. After doing so, the challenge would have the status WAITING_MATCH and another player can accept the challenge. The business rules involved in accepting a challenge is represented in Fig 12.
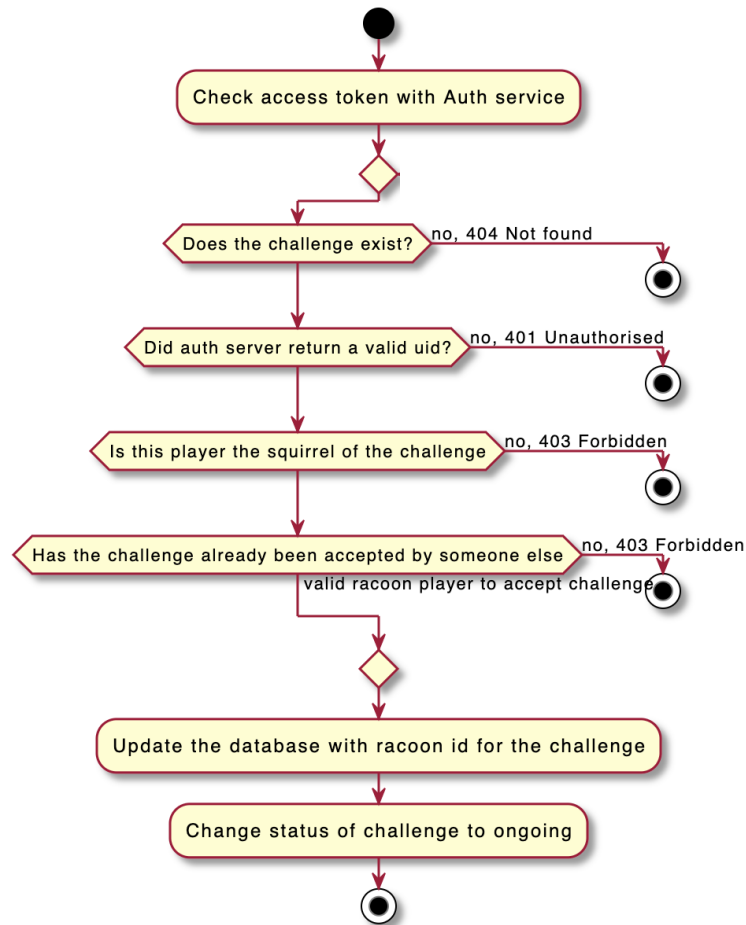
Fig 12. Activity Diagram for Accepting a Challenge

Getting the challenge data

Additionally, users need to see the existing challenge and more details of it, hence the Challenge Service offers 3 GET requests to provide data of challenges based on user id, data of a specific challenge and its related essay paragraphs as well as data of all challenges that are waiting for a match. An example of a GET request (getting challenges based on user id) is shown below.

HTTP Method	GET
Route	/challenge
Service calling	Frontend
Description of API	Get all challenges of a user

Request Header	x-access-token : access token

Response Body	List of challenges
	
	[{ challenge_id : id of challenge
	
	    title : title of challenge in string (i.e. "Title")
	
	    squirrel_id : id of player initiating challenge
	
	    racoon_id : id of player accepting challenge
	
	    num_of_total_turns : 4 or 6
	
	    char_limit_per_turn : 1000 or 1500
	
	    interest: a valid genre in string (i.e. "crime")
	
	    status_of_challenge : status of challenge
	
	    last_modified_time: last modified time of the challenge
	
	    squirrel_name
	
	    racoon_name
	
	    num_of_sequences_completed
	
	    Time_of_last_completed_sequence:
	
	    awaiting_turn_uid: the uid of user to send the essay para next
	
	}, { … }]

More API endpoints of Challenge Service can be found in 12.3 More API endpoints of Challenge Service.

7.7 Community Service

The primary responsibility of the Community Service is to retrieve relevant information for the community page to display to its users. This community aspect aims to connect players with one other and allow them to read other essays, further enriching their knowledge. The community page also allows users to upvote others' challenges and essay paragraphs which are noteworthy and well-written. The Community Service has two endpoints:

- For users to retrieve a list of all completed challenges
- For users to retrieve all information of a specific completed challenge by challenge id

These data are passed to the frontend and displayed to the users.

The Community Service is not responsible for upvoting of challenges or essay paragraphs as that is the responsibility of the Nut Service as elaborated in 7.4 Nut Service.

Get a list of completed challenges

HTTP Method	GET
Route	/community/listChallenges/<offset?>/<limit?> <offset?> and <limit?> parameters in the URL are *optional*. <offset?> is a number that allows you to indicate from which completed challenge to retrieve in the list. If offset is 50, the 51st completed challenge and onwards are retrieved. Completed challenges are sorted in descending order by challenge id. <limit?> is a number that allows you to indicate the maximum number of completed challenges you want to retrieve from the API. If the limit is 100, up to 100 completed challenges will be returned from the API.
Service Calling	Frontend
Description of API	Get a list of completed challenges to be displayed for the community page.
Request Header	x-access-token : access token
Request Body	-

Response	200 Success : OK
	 [
	   { challenge_id : …, title : …, squirrel_id : …, racoon_id : …,
	     squirrel_name: …, racoon_name: …,
	     num_of_total_turns : …, word_limit_per_turn : …,
	     interest : …, upvotes : …, upvoted : … },
	   { challenge_id : …, title : …, squirrel_id : …, racoon_id : …,
	     squirrel_name: …, racoon_name: …,

	num_of_total_turns : …, word_limit_per_turn : …, interest : …, upvotes : …, upvoted : … }, ...] 401 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in Community Service The upvoted value will let you know if the user of the access token has upvoted the challenge or not. If the user has upvoted the challenge, the upvoted value will be his firebase uid, if he did not upvote the challenge, the upvoted value will be null.

Get information of a completed challenge

HTTP Method	GET
Route	/community/getChallenge/<challengeId>
Service Calling	Frontend
Description of API	Get all the information related to a completed challenge by challenge id.
Request Header	x-access-token : access token
Request Body	-

Response	200 Success : OK
	
	[
	
	    challenge : { challenge_id : …, title : …, squirrel_id : …,
	
	       racoon_id : …, num_of_total_turns : …,
	
	       word_limit_per_turn : …, interest : …, upvotes : …,
	
	       upvoted : … },
	
	    essayPara : [
	
	       { challenge_id : …, seq_num : …, author_id : …,

<table>
<tr><td></td><td>

essay_para : …, words_used : …, upvotes : …,

    upvoted : … },

  { challenge_id : …, seq_num : …, author_id : …,

    essay_para : …, words_used : …, upvotes : …,

    upvoted : … },

  ...],

challengeUser : [

  { user_id : …, user_name : …, date_of_birth : …,

    joined_datetime : … },

  { user_id : …, user_name : …, date_of_birth : …,

    joined_datetime : … }]

]

401 Unauthorized : Invalid access token

404 Not Found : Challenge does not exist or is not completed

500 Internal Server Error : An error occurred in Community Service

The upvoted value will let you know if the user of the access token has upvoted the challenge and essay paragraph or not. If the user has upvoted the challenge or essay paragraph, the upvoted value will be his firebase uid, if he did not upvote, the upvoted value will be null.

The challengerUser will return two user profile information, which are the squirrel and racoon user profile.

</td></tr>
</table>

7.8 Notification Service

Note: The Notification Service has been fully implemented in the backend but has yet to be integrated with the frontend due to time constraints. However, we are able to demonstrate a rough working notification service.

The Notification Service's responsibility is to allow backend microservices to create notifications through the service and for the frontend to retrieve user notifications and update if the user has read the notification. The real-time notification system is handled separately on a HTTP server created in the Notification Service. More information can be found in 7.8.1 Socket.IO in Notification Service.

The API endpoints of Notification Service are listed below:

Add notification to a user

HTTP Method	POST
Route	/notification/addNotification
Service Calling	Backend Microservices
Description of API	Add a new notification to a user by firebase uid.
Request Header	-
Request Body	userId : firebase uid notification : varchar *(optional)* notificationLink : varchar
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 500 Internal Server Error : An error occurred in Notification Service

Get recent notifications of a user

HTTP Method	GET
Route	/notification/getNotification

Service Calling	Frontend
Description of API	Get the latest 15 notifications of the user sorted by descending order of the time the notification is created.
Request Header	x-access-token : access token
Request Body	-

Response	200 Success : OK
	
	[
	   { notification_id : …, user_id : …, creation_date_time : …,
	      notification : …, is_viewed : …, viewed_date_time : …,
	      notification_link : … },
	   { notification_id : …, user_id : …, creation_date_time : …,
	      notification : …, is_viewed : …, viewed_date_time : …,
	      notification_link : … },
	   …
	]
	400 Unauthorized : Invalid access token
	500 Internal Server Error : An error occurred in Notification Service

Get all notifications of a user

HTTP Method	GET
Route	/notification/getAllNotification
Service Calling	Frontend
Description of API	Get all the notifications of the user sorted by descending order of the time the notification is created.
Request Header	x-access-token : access token
Request Body	-

Response	200 Success : OK
	`[`
	`{ notification_id : …, user_id : …, creation_date_time : …,`
	`notification : …, is_viewed : …, viewed_date_time : …,`
	`notification_link : … },`
	`{ notification_id : …, user_id : …, creation_date_time : …,`
	`notification : …, is_viewed : …, viewed_date_time : …,`
	`notification_link : … },`
	`…`
	`]`
	400 Unauthorized : Invalid access token
	500 Internal Server Error : An error occurred in Notification Service

User viewed a notification

HTTP Method	PUT
Route	/notification/viewedNotification
Service Calling	Frontend
Description of API	Update the database that the user has viewed a notification by notification id.
Request Header	x-access-token : access token
Request Body	notificationId : integer
Response	200 Success : OK
	400 Unauthorized : Invalid access token
	500 Internal Server Error : An error occurred in Notification Service

User viewed all notifications

HTTP Method	PUT
Route	/notification/viewedAllNotification

Service Calling	Frontend
Description of API	Update the database that the user has viewed all his/her notifications.
Request Header	x-access-token : access token
Request Body	-
Response	200 Success : OK 400 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in Notification Service

7.8.1 Socket.IO in Notification Service

The real-time notification system uses Socket.IO, a library that enables real-time, bidirectional and event-based communication between the client and the server. In our case, the client will have each tab on a web browser that opens up our frontend web page, and the server will be the HTTP server that is created in our Notification Service's Express server.

Each client will try to establish a WebSocket connection to our server. WebSocket is a computer communications protocol that provides a full-duplex and low-latency communication channel over a single TCP connection. Each socket connection has a unique 'socket_id' and it is used in our server to identify the socket that each particular user is on. Therefore, a mapping of 'user_id' to 'socket_id' is created on our server to enable live notifications to be pushed to the user via the socket connection.

It is important to note that on every web page reload (i.e. going to another web page and reloading the current web page), the TCP connection would be disconnected and a new socket connection is established with the server. Therefore, the mapping is updated every time the socket connection is re-established.

As notifications are designed to be only available for authenticated users, once our frontend identifies that the user is authenticated after calling Firebase authentication, the client will emit a 'new_connection' event to the server with his/her 'user_id' in the event. The server listens to the

'new_connection' event, and it creates the mapping between the 'user_id' and 'socket_id' whenever the event is triggered. When the socket is disconnected, the mapping is removed from the server. Therefore, the mapping maintained in the server maps users who are currently online on our application to the socket connection that they have established.

When a new notification is added in the backend, the server emits a 'new_notification' event to the client via the socket connection. The server first checks if the user is online to receive live notifications by checking if his/her 'user_id' exists in the mapping. If the user is online, the new notification is then published to the user via the socket connection through the 'new_notification' event. The client subscribes to the 'new_notification' event and will receive the notification which the frontend then updates the notification into the UI accordingly.

The design of the publish-subscribe pattern between the client and server is shown in the diagram below.
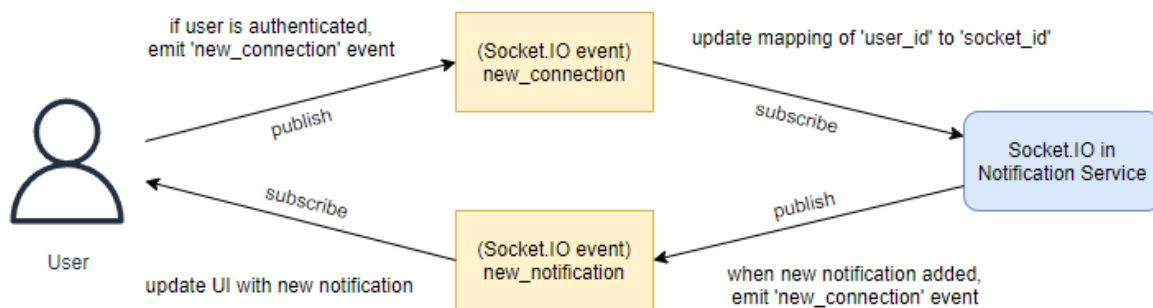

Fig 13. Design of Socket.IO Publish-Subscribe Pattern in our application

A user can open many tabs of our application in their web browser, and as mentioned, each tab is a client itself. This means that the mapping maintained in our server needs to be able to handle many 'socket_id' being mapped to a single 'user_id'. The server creates two maps, one to map each 'user_id' to a set of 'socket_id' and the other to map each 'socket_id' to one 'user_id'. An example of the mapping is shown below.

'user_id' to 'socket_id' map
['user1' : { 'socket1', 'socket2', 'socket3' },
  'user2' : { 'socket4' }]

<u>'socket_id' to 'user_id' map</u>

['socket1' : 'user1',

 'socket2' : 'user1',

 'socket3' : 'user1',

 'socket4' : 'user2']

8. Frontend

8.1 Tech Stack

Our frontend uses the React JavaScript library for building user interfaces and UI components, and uses React-Bootstrap as our frontend stylesheet library.

8.1.1 React

React is a javascript library that supports front-end development. It is component-based and allows users to build encapsulated components that manage their own state. React will efficiently update and render the appropriate components as the data changes, making React a dynamic solution for front-end development.

Notable Concepts:

State Management
States were used in various components to allow for dynamic rendering. By attaching event handlers like onChange() allows for these states to be updated and rendered immediately so that users can see the changes dynamically.
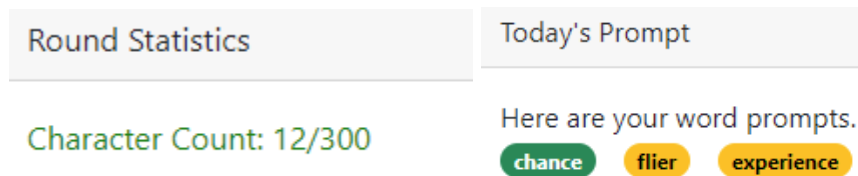

Fig 14. Dynamic State Changes in Character Count

Conditional Rendering
When designing React components, conditional rendering is often required to show a specific component depending on a given state. For example, in our Challenges page that pulls all challenge data for the current user, a function is called to derive the status of each challenge.
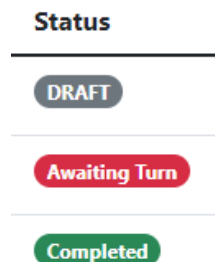
Another example of conditional rendering would be in the Navigation Bar component. Within the component, the tabs available for authorized users will only be shown if said user exists.

```
{user && (
  <Nav className="me-auto">
    <Nav.Link href="/profile">Profile</Nav.Link>
    <Nav.Link href="/challenge">Challenges</Nav.Link>
    <Nav.Link href="/community">TreeHouse</Nav.Link>
    <Button variant="light" className="ml-auto" onClic
  </Nav>
)}
```

Fig 16. Code On Conditional Rendering In Navigation Bar Component

8.1.2 React-Bootstrap

React-Bootstrap is built on top of the ever-so popular CSS framework - Bootstrap. Each UI component provided by React-Bootstrap was built ground up as a React component making it easy to integrate with React. This minimises unnecessary dependencies and keeps the source code uniformed. Furthermore, these UI components support Bootstrap class names, allowing us to write custom styling in finer granularity.

In order to create components that were styled to our theme, we utilised a custom CSS file that acts to augment the existing styling of components. Some key

- Custom CSS

Centralised API file
As there were several endpoints for the front-end to make API calls to, we encapsulated all API calls into functions and stored them in a centralised API file. "Axios", a promise-based HTTP Client, was used to make HTTP requests to the back-end services. The benefits of doing this, reduces repeated code to make API calls in different components. It also adheres to separation of concerns design principle, where the responsibility of making API calls is abstracted away from the main frontend logic.

Player experience
To ensure good player experience, we need to ensure that the business logic in terms of ensuring the correct turn is determined, ensuring the player can only add to challenges that they are part of, and other business rules, is adhered to. If invalid requests to the backend is sent, the relevant error messages are returned but these checks also need to be done in the frontend

and provide visual cues to the player. For example, if it is not the player's turn, the inputs are disabled.

Fig 17. Input is Disabled as Not Player's Turn

Furthermore, the status of the challenge is updated accordingly so that players know which state of the challenge they are at.

Authentication

As mentioned earlier in Security Requirement, security is an important concern in our platform. Hence we use Firebase authentication to authenticate users. With Firebase API, we are able to add a user, log them in and retrieve an access token. This token is sent along with requests to the backend, so that backend can verify the token and its user by calling the Auth Service.

9. Application Screenshots

We provide some screenshots of WordTree below.

Fig 18. Registration Page of WordTree

Fig 19. Selection of Interests

Fig 20. TreeHouse Community Page

WordTree   Profile   Challenges   TreeHouse   Logout   🔔

Completed Challenge

Go back / Challenge ID: 4

ID	Upvotes	Title	Interest	Authors	Turns
4	2	Journey to the Dragon Island	Adventure	Yun Qing Justin	4

Completed Essay

We took the risk, and hoped to get a reward worthy of it. Although it was dangerous, we decided to gamble on the voyage to the mysterious dragon island. The journey could jeopardize our live, but the fable we heard said the reward was as high as the stake. Upon reaching the dragon island at twilight, we discovered the tale was true. We were standing in front of a huge dragon, shaking in fear.

Squirrel: Yun Qing ⬆

We took the risk, and hoped to get a reward worthy of it.

Racoon: Justin ⬆

Although it was dangerous, we decided to gamble on the voyage to the mysterious dragon island.

Squirrel: Yun Qing ⬆

The journey could jeopardize our live, but the fable we heard said the reward was as high as the stake.

Fig 21. Completed Essay In TreeHouse

WordTree   Profile   Challenges   TreeHouse   Logout   🔔

View Challenge Requests

Challenges / View challenge requests

ID	Squirrel	Title	Interests	Turns	Word Limit	Actions
5	Boon Ji	Untitled	Sci-Fi	6 rounds	1500 char limit	Accept
3	Nikhila	Untitled	Fantasy	4 rounds	1500 char limit	Accept

Fig 22. View All Challenges Requests

Fig 23. Ongoing Challenges Type

10. Project Management

We held weekly 1-hour sprints to update one another about our progress and to plan and distribute the workload for the coming week. We had a working google document which we referred to and updated every week for sprint planning. We listed the tasks and allocated manpower for the tasks. Some screenshots from the document are provided below with regards to our development plan. As it is a working document, some strikethroughs represent completion of tasks when we are reviewing the tasks during the sprint itself.

27-09-2021

- Set up
    - Replace some existing Files
- Add into SQL file (complete design of database) - split by Service:
    - Create User Table - Justin
    - Create Profile Table - BJ
    - Create Challenge Table - Nikhila
    - Create EssayPara Table - YQ
    - Create Word Table - YQ
    - Create Nut Tables - BJ
    - + any other tables we may need
- ~~Create new challenge endpoint (**Challenge Service**) - Nikhila~~
- Create Essay Service + Word Service endpoints (Essay + **Word Service**) - YQ
- Connect to Auth0 management API - Justin
- Nut Service - BJ
- Model Profile Page (Maybe next sprint) - Justin
- Kafka vs RabbitMQ (Slight Readup) -BJ

Git Branch Naming:
\<issue id\>-\<description\>

Issue Naming:
- Indicate services, (which) Service, UI, Others
- Requirements - FR / NFR

Commit Naming:
Hotfix: Hotfix abc bug
Feat: Add ui button
Bug: Fix abc bug
Doc: Add documentation

Fig 24. Distribution of Work and Establishing Git Naming Conventions on 27/9/2021

06-10-2021 (Wednesday - Monday)
- ~~Start a conversation thread with Jeremy our mentor (Yun Qing)~~
- **Complete Authorization** with Firebase (Justin)
- Setup an **API Gateway** (Yun Qing Nikhila) Imma explore nginx
    - Create an issue for discussion & reference to PR
- ~~Send UserDB create table (Justin)~~
- Authorisation Service [future]
- **Notification Service (BoonJi)**
    - Pub-Sub messaging
    - UI
- **Basic Styling of pages (Justin)**
- ~~Combine the tables + put on cloud (Yun Qing -> will do by tdy) [DONE]~~
- ~~Finish up endpoints for Essay (Yun Qing) [DONE]~~
- ~~Finish up endpoints for profile / nut (by friday)~~
- **Finish up endpoints for challenge service (Nikhila)**

Soft targets: 6/7th Oct PR existing changes into code base

Fig 25. Distribution of Work and Reviewing of Completed Work on 6/20/2021

We made use of Github Issues to keep track of our tasks and link them to the respective pull requests when completed. Some sample issues are shown below.

☐ ⊙ **Integrate Notification Service with K8s** `priority: high`

#55 opened 4 days ago by tanboonji

☐ ⊙ **Build Community Page** `priority: high`

#33 opened 16 days ago by justgnoh   ☑ 3 tasks   ⬦ v2 – Core Featu...

☐ ⊙ **Integrate Challenges Page with API endpoints** `priority: high`

#32 opened 16 days ago by justgnoh   ☑ 4 tasks

Fig 26. Sample Github Issues Raised

To ensure code quality, we also established that PRs for important features must be reviewed by at least 1 reviewer before they can be merged to master branch.

11. Learning Experience

11.1 Suggestions for Improvement/ Enhancements

We present some areas where we could have improved on or implemented differently for WordTree:

- We used vanilla SQL queries to implement our data access object. We could explore using Object-Relational Mapper (ORM) for such implementations in future.
- We considered integrating Redis caching for some endpoints so that the GET queries can be returned faster. However, we were unable to implement this due to time constraints.
- We should consider integrating some form of testing (e.g. Jest) which we were unable to complete due to time constraints.

We also propose the following extensions that can be made to WordTree:

- We can consider implementing a player chat feature whereby if the two players are online, they can come together to work on the same essay together by communicating through the chat. This is an alternative to the turn-based solving approach that we currently support.
- We can consider setting time limits on each player's turn.

11.2 Challenges Faced/ Reflections

As most of us were rather new to web-application development, we were faced with a steep learning curve to learn everything from scratch such as JavaScript, Kubernetes, using JWT tokens. The time taken to pick up these skills resulted in us not being able to achieve some of the features that we planned to have originally. We also had to find up about good practices when using these new technologies as we wanted to apply them properly instead of just trying to get our application to work. For example, we initially implemented our code in classes in JavaScript. However, we later realise that in practice, classes are usually used when using typescript instead. Hence, we converted our code to export the functions instead.

We also faced issues with regards to the frontend and backend communication. In retrospect, what we should have done was to discuss how to implement the frontend and backend together instead of separating them so that it will make it easier to integrate the API calls.

We also faced some issues with the implementation of Socket.IO for live notification also along the way, which took tedious steps to debug and solve. Establishing the socket connection between the frontend and a standalone HTTP server was easy to implement. However, the problem came when the connection needed to be established through kubernetes with a nginx ingress controller. After the ingress problem was solved, we realised the frontend was establishing multiple socket connections with the server, and this caused the server to be unable to send notifications to the correct socket as the mapping maintained between 'socket_id' and 'user_id' was not updating properly. We realised this was due to React reloading the web page multiple times due to React's 'useEffect()' hook, and each web page reload would create a new socket connection but the previously created socket connections were not disconnected. Although the frontend has yet to implement notifications, we found solutions to this by either initiating the connection outside of the returned function of each web page or moving notification and sockets to a separate component that will be imported by each web page.

Overall, while we did not manage to achieve all the features that we had hoped to initially, we are definitely proud to have completed a fully functional web application and have learnt a lot of new technologies along the way, given how most of us did not have much experience in web application development.

12. Appendix

12.1 More API endpoints of User Service

<u>Get user profile information</u>

HTTP Method	GET
Route	/user/getUserProfile
Service Calling	Frontend
Description of API	Retrieves user profile information and user nuts information by access token.
Request Header	x-access-token : access token
Request Body	-

Response	200 Success : OK
	{
	   profile : { user_id: …, name: …, date_of_birth: …,
	     joined_datetime: … },
	   interest : [{ interest: … }, { interest: ... }, …],
	   nut : { totalNut: ..., essayNut: …, communityChallengeNut: …,
	     communityEssayNut: … }
	}
	401 Unauthorized : Invalid access token
	500 Internal Server Error : An error occurred in User Service

<u>Update user profile information</u>

HTTP Method	PUT
Route	/user/updateUserProfile
Service Calling	Frontend
Description of API	Updates user profile information by access token.
Request Header	x-access-token : access token
Request Body	name : varchar dateOfBirth : date format 'MM/DD/YYYY' or 'YYYY/MM/DD' *(optional)* interest : [ ] (empty array = no interest) **OR** ['interest'] (only one interest) **OR** ['interest 1', 'interest 2', …] (many interests)
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters

	401 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in User Service

Get all interests available in our application

HTTP Method	GET
Route	/user/getInterest
Service Calling	Frontend
Description of API	Get all the interests available in our application for users to select and indicate their interests from.
Request Header	-
Request Body	-

Response	200 Success : OK
	[
	   { interest : … },
	   { interest : … },
	   …
	]
	500 Internal Server Error : An error occurred in User Service

Get all interests of a user

HTTP Method	GET
Route	/user/getUserInterest
Service Calling	Frontend
Description of API	Gets all the interests of a user by access token.
Request Header	x-access-token : access token
Request Body	-

Response	200 Success : OK
	[
	   { interest : … },
	   { interest : … },
	   …
	]
	401 Unauthorized : Invalid access token
	500 Internal Server Error : An error occurred in User Service

Other API endpoints of User Service can be found in 7.2 User Service.

12.2 More API endpoints of Nut Service

Add community challenge nut to a user

HTTP Method	POST
Route	/nut/addCommunityChallengeNut
Service Calling	Frontend
Description of API	Add community challenge nuts to the two users of the challenge upon another user upvoting their challenge.
Request Header	x-access-token : access token
Request Body	upvotedUserId1 : firebase uid upvotedUserId2 : firebase uid challengeId : integer
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 401 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in Nut Service

Delete community challenge nut of a user

HTTP Method	DELETE
Route	/nut/deleteCommunityChallengeNut
Service Calling	Frontend
Description of API	Delete community challenge nuts of the two users of the challenge upon another user removing their upvote from the challenge.
Request Header	x-access-token : access token
Request Body	upvotedUserId1 : firebase uid upvotedUserId2 : firebase uid challengeId : integer
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 401 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in Nut Service

Add community essay nut of a user

HTTP Method	POST
Route	/nut/addCommunityEssayNut
Service Calling	Frontend

Description of API	Add community essay nuts to the user who wrote the essay paragraph upon another user upvoting his/her essay paragraph.
Request Header	x-access-token : access token
Request Body	upvotedUserId : firebase uid challengeId : integer seqNum : integer
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 401 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in Nut Service

Delete community essay nut of a user

HTTP Method	DELETE
Route	/nut/deleteCommunityEssayNut
Service Calling	Frontend
Description of API	Delete community essay nuts of the user who wrote the essay paragraph upon another user removing his/her upvote from the essay paragraph.
Request Header	x-access-token : access token
Request Body	upvotedUserId : firebase uid challengeId : integer seqNum : integer
Response	200 Success : OK 400 Bad Request : Invalid request body / missing parameters 401 Unauthorized : Invalid access token 500 Internal Server Error : An error occurred in Nut Service

Other API endpoints of Nut Service can be found in 7.4 Nut Service.

12.3 More API endpoints of Challenge Service

Add title to challenge

HTTP Method	POST
Route	/challenge/title
Service calling	Frontend

Description of API	Add title to the challenge
Request Header	x-access-token : token given by firebase from frontend
Request Body	challenge_id: challenge id title: title
Response Status	400 Bad Request if missing or invalid fields 200 Success

Accept Challenge

HTTP Method	POST
Route	/challenge/accept
Service calling	Frontend
Description of API	Assign the opposite player
Request Header	x-access-token : token given by firebase from frontend
Request Body	challenge_id: challenge id
Response Status	400 Bad Request if missing or invalid fields 200 Success
Response Body	Next three words for racoon to start with { "words" : ["one", "two", "three"] }

Get all waiting matches challenges

HTTP Method	GET
Route	/challenge/waiting
Service calling	Frontend
Description of API	Get all matches awaiting a partner
Request Header	x-access-token : token given by firebase from frontend
Response Status	400 Bad Request if missing or invalid fields 200 Success
Response Body	challenge_id : id of challenge title : title of challenge in string (i.e. "Title") squirrel_id : id of player initiating challenge racoon_id : id of player accepting challenge num_of_total_turns : 4 or 6 word_limit_per_turn : 300 or 500 interest : a valid genre in string (i.e. "crime")

	status_of_challenge : status of challenge words: words_for_next_sequence_if_exists last_modified_time: last modified time of the challenge squirrel_name racoon_name num_of_sequences_completed time_of_last_completed_sequence

Get challenge by challenge ID

HTTP Method	GET
Route	/challenge/<id>
Service calling	UI
Description of API	Gets the challenge details of the given challenge id
Request Header	x-access-token : access
Response Status	400 Bad Request if missing or invalid fields 403 Forbidden if the challenge is not completed and player accessing the challenge is not part of it 200 Success
Response Body	challenge_id : id of challenge title : title of challenge in string (i.e. "Title") squirrel_id : id of player initiating challenge racoon_id : id of player accepting challenge num_of_total_turns : 4 or 6 word_limit_per_turn : 300 or 500 interest : a valid genre in string (i.e. "crime") status_of_challenge : status of challenge words: words_for_next_sequence_if_exists last_modified_time: last modified time of the challenge squirrel_name racoon_name num_of_sequences_completed time_of_last_completed_sequence essay_paras: list of essay para objects : [{ seq_num: sequence number, author_id: id of author, essay_para: text of the essay in string, words_used: [list of words successfully used]}, ...]

Add essay para to a challenge

HTTP Method	PUT
Route	/challenge/<:id>

Service calling	UI
Description of API	Add an essay para to the existing challenge
Request Header	x-access-token : access
Request Body	essay_para : essay para being added title : '' or a legit title
Response Status	400 Bad Request if missing or invalid fields 404 If it is no such challenge 403 If it is not this player turn 403 If no more turns for this player 200 Success

Get challenges of a user by user ID

HTTP Method	GET
Route	/challenge
Service calling	UI
Description of API	Get all challenges of a user
Request Header	x-access-token : access token

Response Body	List of challenges
	[{ challenge_id : id of challenge
	     title : title of challenge in string (i.e. "Title")
	     squirrel_id : id of player initiating challenge
	     racoon_id : id of player accepting challenge
	     num_of_total_turns : 4 or 6
	     word_limit_per_turn : 300 or 500
	     interest: a valid genre in string (i.e. "crime")
	     status_of_challenge : status of challenge
	     last_modified_time: last modified time of the challenge
	squirrel_name
	racoon_name
	num_of_sequences_completed
	time_of_last_completed_sequence
	awaiting_turn_uid: the turn that has yet to submit
	}, { … }]

Response Status	400 Bad Request if missing or invalid fields 200 Success