MSc Computer Science

School of Computer Science, University of Birmingham

2021

LangChat - A chatbot for English language learners

George Harvey

2237296

Supervisor: Dr Phillip Smith

Abstract

This report details the process by which a browser-based chatbot was developed to help English language learners practise their speaking and listening skills. The chatbot is rule-based and harnesses Artificial Intelligence Markup Language to match user input patterns. The application server was written in Python using the Flask web development framework, and front-end was written in JavaScript using the ReactJS framework. After development, the application underwent comprehensive unit and integration testing. Following this, the application was deployed on the internet at https://langchat.co.uk and underwent a small user trial.

Table of Contents

1. Introduction and motivation

The aim of this project was to develop a chatbot which could be used as a pedagogic aid for learners of the English language. The chatbot was intended to introduce students to new vocabulary and engage in dialogue, during which they were encouraged to apply the new language they have learned. The chatbot would be made available on the web and accessed through the browser.

One of the greatest hurdles to learning a foreign language is finding the confidence to practise with speaking and listening skills. Almhadi (2014) concludes that while these verbal communication skills are the most essential in language acquisition, they are also the ones that induce the most anxiety in students. He goes on to say that students who experience these emotions during the language learning process are less likely to succeed at achieving fluency and more likely to abandon the process altogether. Cohen (1989) suggests that this is further pronounced in learners with introverted personalities and often results in them falling behind their peers. The anxiety induced by these conversation scenarios creates a vicious cycle in which students avoid practising their oral skills, and therefore fail to gain the confidence they need to start practising.

One idea to break this cycle is to use chatbots as a conversational partner, thus allowing students to improve oral skills to a point where they are confident enough to graduate to real life conversations with native speakers. Chatbots seem like the perfect solution to this problem. They are available 24/7, unlike human conversation partners, and the user is highly unlikely to feel any sense of anxiety when conversing with a machine (Fryer 2006). This, along with the rapid advancement in artificial intelligence technologies suggests that chatbots might already be firmly entrenched in the language learning process.

This idea is not novel. As early as 1999, Atwell wrote that many language teachers were requesting a “conversation practise machine”. There have been a number of attempts to provide this technology, but most have been met with a lukewarm response which in many cases has led to them being discontinued (Fryer 2020). Recently, the rise of personal assistants such as Alexa and Siri has led to renewed interest in using chatbots for language learning (Dale 2020).

What follows is a brief history of the chatbot, a look at how the failings of previous attempts might be addressed and ultimately this project’s proposed solution to the problem.

1.2. History of chatbots

Among the first to seriously consider the question “Can a computer communicate in a way indistinguishable from humans?”, was computer science pioneer Alan Turing. He devised a test, known as the Turing Test, to determine the effectiveness of such a computer, where a participant attempts to identify whether their partner in conversation is a human or computer. Turing suggests that if the participant is incorrect at least 50% of the time, the machine is deemed to have artificial intelligence (Fryer 2006). While the accuracy of this test has since been questioned (Dale 2016), it has nevertheless inspired a huge number of attempts at a solution.

It was not until 1966 that the first serious contender for the Turing Test appeared. ELIZA was developed by professor Joseph Weizenbaum at the Artificial Intelligence Laboratory in MIT (Zemcic 2019). ELIZA used basic pattern matching and template based responses to converse with users (Shawar 2007), bringing the concept of a chatbot into the public eye for the first time. ELIZA’s success relied upon responding with open ended responses, often parrying the user’s questions with it’s own, therefore turning attention away from itself and onto the user. Indeed the intention of ELIZA was to emulate a Rogerian psychiatrist, one which engages in “client-centered therapy”, where the client does most of the talking (Zemcic 2019).

Another notable early chatbot is PARRY, which was developed in 1972 by Kenneth Mark Colby at Stanford’s Psychiatry Department. Its intention was to emulate a paranoid schizophrenic patient with which psychiatrists could learn how to communicate (Zemcic 2019). A famous interaction took place between the chatbots ELIZA and PARRY at a computer conference also in 1972. The transcript is given below to demonstrate their conversational styles (Guzeldere 1995).

parry: People get on my nerves sometimes.
eliza: I am not sure I understand you fully.
parry: You should pay more attention.
eliza: Suppose you should pay more attention?
parry: You're entitled to your opinion.
eliza: What makes you think I'm entitled to my opinion?
parry: It bothers me just to be around people.
eliza: Please go on.
[Parry continues with some paranoid ramblings about horseracing, bookies, and the Mafia]
parry: I know the mob controls the big rackets.
eliza: Who are you really talking about?
parry: Have you ever really thought about the Mafia?
eliza: Suppose you ever really thought about the Mafia?
parry: Let's talk about something else.
eliza: I am not sure I understand you fully.
parry: You keep asking me the same things, why not change the topic?
eliza: In what way?

Fig 1: Conversation between chatbots ELIZA and PARRY

Development of chatbots picked up pace in 1990 when Hugh Loebner created an annual competition implementing the Turing test called the Loebner Prize. The competition offers $100,000 to any chatbot developer that can pass the test. Although the grand prize has yet to be won, the best chatbot each year wins between $2,000 and $3,000 (Zemcic 2019).

One of the most prolific contestants for the Loebner Prize is three time winner A.L.I.C.E. This chatbot also uses pattern matching similar to ELIZA, however its knowledge of dialogue patterns is stored in AIML (Artificial Intelligence Markup Language) files. These are derived from XML (Extensible Markup Language) files, and allow the "knowledge" of the chatbot to be easily extended by adding to the files from which it operates (Shawar 2007). These AIML documents offer a convenient way to plan dialogue flows and will be discussed in further detail later.

More recently, advances in the field of artificial intelligence, especially neural networks, have allowed chatbots to become more intelligent (Sianaki 2019). Whereas earlier rule-based chatbots were limited to responding with predefined sentences, these newer, generative chatbots are able to form their own replies depending on the user's input. This new breed of chatbot is able to learn dynamically from the conversations it has with users. While it is capable of giving the most engaging and lifelike experience, it is also less reliable than its rule-based counterparts as its responses are not guaranteed to be syntactically correct (Sianaki 2019). A good example of a generative chatbot is Microsoft's Xiaoice which is widely regarded as one of the most lifelike chatbots ever created (Dokukina 2020). The purpose of Xiaoice is to act as an empathetic chatbot, that is, one with which the user develops a personal connection with over a long period

of time (Shum 2018). Naturally a generative chatbot is best suited to this role, as the more it converses with the user, the more engaging it will be.

1.3. Chatbots for language learning

Given this continuous development of chatbot systems, one would be forgiven for assuming that their ubiquity as conversational aids for language learners had grown in recent years. In 2006, Fryer and Carpenter, developer of chatbot Jabberwacky, noted that some of the obstacles for chatbots as language tools included an inability to process user input containing grammatical errors, a lack of feedback and open ended responses that were likely to confuse beginner learners.

Two years later in 2008, Coniam reviewed six of the leading chatbots whose purpose was to help students learn English. His conclusion was that chatbots had potential to be of use in the language learning sphere, but at the time were very limited and had a long way to go. He noted that one significant limitation was the speech recognition technology, which was fraught with inaccuracies and mistakes.

Development and interest in chatbots for language learning stagnated over the next decade or so until around 2016, which Chris Messina of Uber named “the year of conversational commerce”. This, along with the public’s increased exposure to voice operated personal assistants such as Siri and a boom in the market for mobile language learning applications such as Duolingo, resulted in renewed interest in chatbots as a language aid (Dale 2016). Duolingo released their own chatbot for several languages, but after several years it was removed from the application, with owner Luis von Ahn claiming that not many people were using it. This comes in spite of Hill, Ford & Farreras asserting in 2015 that people are enthusiastic about chatting with bots, leaving us wondering what can be done to improve their standing in the language learning field.

1.4. Possible improvements

Firstly, we can look at the shortcomings of chatbots in language learning previously mentioned by Fryer and Carpenter.

1. Chatbots fail to give feedback about their use of language to the user
2. Chatbots struggle to understand input containing grammatical errors
3. Open ended, undirected dialogues leave beginners feeling overwhelmed

The first two of these issues can be solved by continuously checking the user’s syntax, grammar and spelling and feeding back information to the user. This way the user is not only given

feedback about their language usage, but they are also encouraged to correct their own input, allowing it to be correctly understood by the chatbot. The third issue here suggests that users, especially beginners, are more likely to benefit from structured dialogues, where the onus is on the chatbot to direct the conversation, rather than the user. This makes chatbots such as ELIZA, which as previously mentioned, make the user do all the talking, particularly unsuitable for language learners.

Further to this, we can consult literature regarding successful integration of technology into learning, and adapt its findings to the language learning sphere. One such work by Mayer (2017), gives three research-based principles for managing essential processing in e-learning.

1. Segmenting: People learn better when a multimedia lesson is presented in small user-paced segments.
2. Pre-training: People learn better when they learn the key terms prior to receiving a multimedia lesson.
3. Modality: People learn better from a multimedia lesson when words are presented in spoken form.

These three principles can be directly interpreted to optimise the effectiveness of a chatbot in a language learning environment. Firstly, the segmenting principle acts as the solution to the third issue presented by Fryer and Carpenter (2006). One way to implement this is to have the dialogues restricted to certain topics, such as the weather, and allow the user to select which topic they wish to discuss before beginning. The pre-training principle can be linked to segmenting, that is that key vocabulary pertaining to the topic of choice should be introduced before the dialogue, thus allowing users to familiarise themselves with words they may not yet know. Finally, the third principle of modality suggests why many of the earlier chatbots may not have been successful. In Coniam's review of language learning chatbots, he noted that speech synthesis tools were only capable of producing low quality voices and that speech recognition technology was not sufficient to reliably capture the user's speech. Fortunately these are two areas in which real progress has been made over the last decade, and we can harness this to give the user a fully interactive experience.

1.5. Similar implementations

Since Duolingo withdrew the chatbot from their application, Mondly is leading the way as a major language learning application to offer such a feature (Fryer 2020). They added their chatbot in 2016, and a year later, released a separate chatbot application which made use of AR (Augmented Reality) to offer users a more immersive conversational experience (Fryer 2020). The user is able to select from a number of situational dialogues, where they will practise common tasks such as ordering at a restaurant or booking a hotel room. Only the first dialogue, which

covers typical introductory phrases, is available without purchasing the premium version of the application.

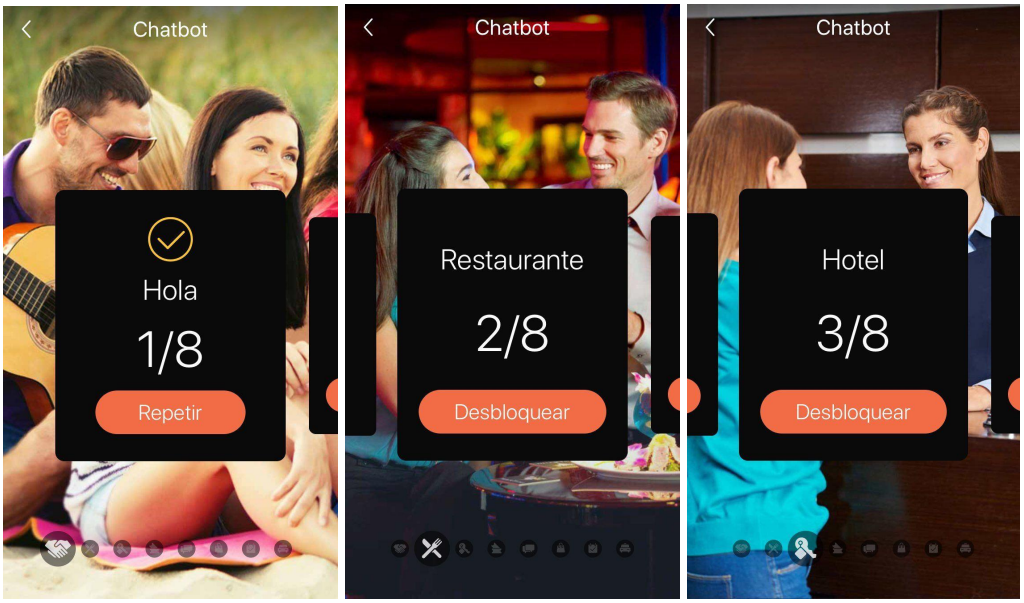

Figure 2: Three first dialogues of Mondly's chatbot for Spanish speakers learning English

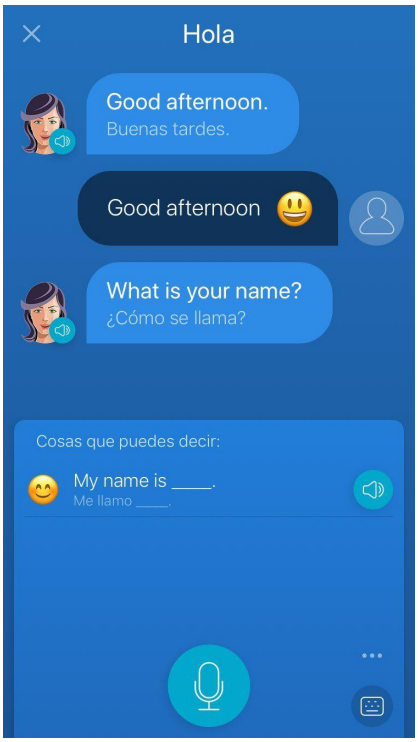

Figure 3: Screenshot of Mondly's chatbot dialogue interface

During the dialogue, the chatbot leads the conversation and the user is prompted with suggested responses. The transcript of the dialogue is automatically displayed, with no option to hide the text or the suggested replies. The chatbot is mostly restricted to understanding replies in the

formats it has suggested, responses outside of these structures are rejected as incorrect. A few examples of this behaviour are given below.

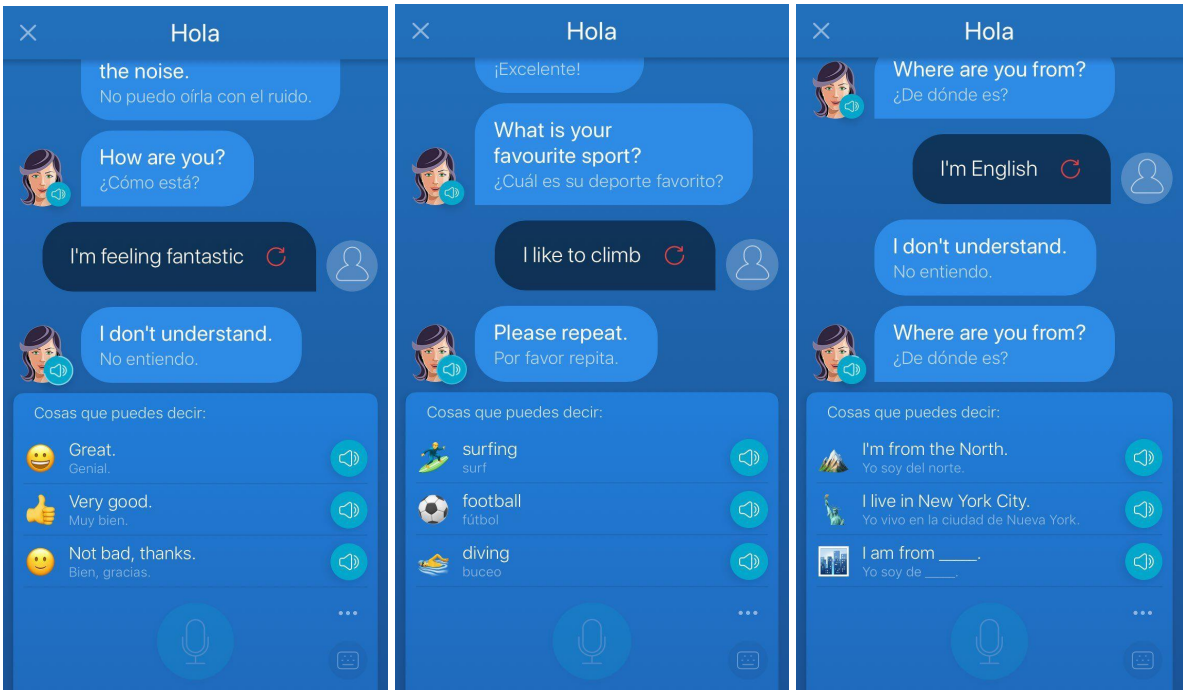

Figure 4: Three examples of correct English rejected by Mondly's chatbot

One helpful feature that Mondly includes is that when a response is understood, a relevant emoji is shown to the user as a form of affirmation. This is achieved through development of a key phrase database which maps certain words or phrases to corresponding emojis. Fryer (2020) asserts that this type of visual feedback is crucial as it helps to support users' confidence.

Figure 5: Example of relevant emoji applied to user's response

According to the research presented by Mayer (2017) detailed above, Mondly fails to make use of pre-training, as users are thrown into each dialogue with no prior introduction to key vocabulary. His principle of segmenting is satisfied however, with each dialogue being narrowly focussed on a specific situation. Finally, although the content is delivered in spoken form as his modality principle advises, the interface's text heavy display means that users may depend on their reading abilities to comprehend the chatbot, which may result in reduced development of listening skills.

2. Background

This section will cover the specifics of how chatbots are designed and developed. Two important attributes to consider are their domain and implementation, where domain refers to the content they are capable of discussing. Open domain chatbots are generally created for the purpose of entertaining the user with lifelike conversational discourse. Closed domain chatbots on the other hand are more frequently employed in e-commerce or other task specific areas, where they have concise, accurate dialogue about specific topics (Sianaki 2019). Figure 6 shows the various classifications of chatbots and the three implementation options will be discussed below

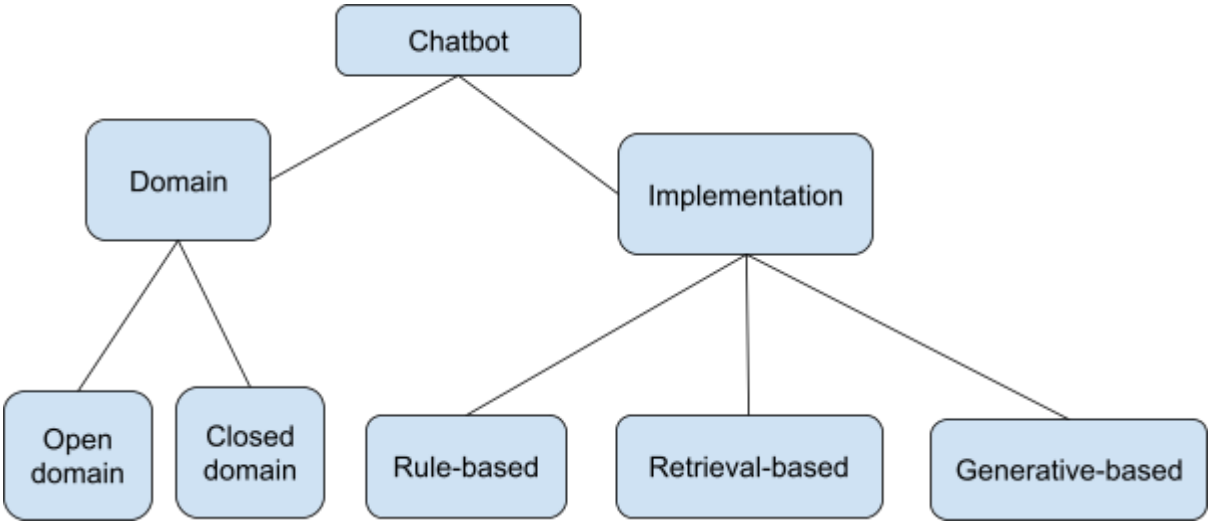

Figure 6: Diagram showing the different classifications of chatbots

2.1. Rule-based chatbots

Rule-based chatbots were the earliest to be created and for about 40 years, were the only option for chatbot developers. These chatbots can only understand and respond to user input based on predefined rules. These rules can take the form of parsing algorithms that try to extract keywords and sentiment from a sentence or pattern matching, the most common approach for chatbot design. Rule-based chatbots are the most lightweight and easy to implement, however they often suffer from a lack of flexibility, as every form of user input has to be considered and manually accounted for by the developer. One particularly powerful tool for creating rule-based chatbot is AIML, as previously mentioned.

AIML operates by defining <pattern> and <template> tags. The chatbot tries to match user input with one the <pattern> tags and if successful, responds with the text contained within the corresponding <template> tag. An in depth discussion of AIML semantics is given below.

Each AIML file is made up of different categories. These are represented by the <category> tag, which encloses a <template>, <pattern> pair. This represents the base units of knowledge that the chatbot has.

```
<category>
  <pattern>HELLO<pattern>
  <template>Hi there, how are you?<template>
<category>
```

In this example, if the user enters the text “Hello”, the chatbot will respond with “Hi there, how are you?”. Note that the user’s input is case insensitive, as any text is normalised before being matched. This means it is converted to uppercase and any non alphanumeric characters are removed.

Wildcards can be used within the <pattern> tags, to allow more flexible matching of the user’s input. These include * which accepts 1 or more words, and # which accepts 0 or more. The word or words captured by these wildcards can then be referenced by the chatbot in its response template using the <star /> tag. This tag accepts an optional index value which allows the chatbot to differentiate between the content of different wildcards within the same pattern. If not provided, this index defaults to 1.

```
<category>
  <pattern>MY NAME IS *<pattern>
  <template>It's nice to meet you <star/><template>
<category>
```

If the user inputs “My name is George”, the chatbot will reply with “It’s nice to meet you George”.

Another important tag in AIML is the <that> tag. The text enclosed by this tag refers to the previous response the chatbot has given. This allows the chatbot to determine the context of the user’s reply.

```
<category>
  <pattern>Five<pattern>
  <that>HOW MANY FILMS HAVE YOU WATCHED</that>
  <template>That's not very many!<template>
<category>

<category>
  <pattern>Five<pattern>
  <that>HOW MANY CHILDREN DO YOU HAVE</that>
  <template>That's a lot!<template>
<category>
```

Here, depending on the previous question, the chatbot will interpret the same user input in two different ways.

The <srai> tag is used to prevent the developer from having to redefine the same template for multiple different patterns. This accounts for the vast number of different ways of conveying the same meaning that exist within languages. The <srai> tag, which stands for ' symbolic recursion artificial intelligence', gives AIML a recursive capacity, allowing documents to become more concise.

```
<category>
  <pattern>I WANT TO TALK ABOUT MUSIC<pattern>
  <template>Okay, what is your favourite song?<template>
<category>

<category>
  <pattern>I WOULD LIKE *<pattern>
  <template><srai>I WANT <star /></srai><template>
<category>
```

In this example, if the user says "I would like to talk about music", firstly the input will be matched with the second category, next all words following the word "like" will be captured by the * wildcard. The chatbot will then invoke itself with the contents of the wildcard, preceded by "I want". In this case, it will pass itself "I want to talk about music.", which will match the first category, finally replying to the user with "Okay, what is your favourite song?**".**

The final tags that will be mentioned are the <set> and <iset> tags. These allow the chatbot to define sets of words that will be accepted in a given location within the pattern. The difference between the two is that <set> refers to the contents of a text file defined elsewhere, whereas <iset>, which stands for 'inline set' can be defined within the AIML document. Two respective examples are given below.

```
<category>
  <pattern>I DRIVE A <set>car-brands</set><pattern>
  <template>How lucky! I wish I owned a <star />.<template>
<category>

<category>
  <pattern>I <iset words="ENJOY, LOVE, LIKE" /> READING<pattern>
  <template>Me too, I read everyday.<template>
<category>
```

In the first category, the chatbot will look for a file named 'car-brands', within a provided directory. If the user's input contains one of the words within this file, the pattern will be matched. In the second category, all three verbs displayed will allow the pattern to be matched. Generally the <set> tag is used when the list of accepted words is particularly long or likely to be reused, whereas the <iset> tag is used for short, one-off lists of accepted words. It should also be noted that the <star> tag is able to reference words from a <set> or <iset> in the same way as it can for wildcard characters.

2.2. Retrieval-based chatbots

Retrieval based chatbots were the next to be developed, following advances in artificial intelligence in the early 2000s. These chatbots use AI algorithms to match the user's pattern to the most likely predefined intent, often located in a JSON (JavaScript Object Notation) file. The benefit of this approach is that the user's input does not have to match exactly with a predefined pattern, instead, the machine learning algorithm ranks each intent by its likelihood to match the input and returns the one with the highest value, provided it is above a certain threshold. An example of an intents file is provided below. Each intent within the "intents" list has three attributes. The "tag" is a name used to define the intent, the "patterns" list gives examples of model sentence structures, similarity to which determines the probability of the user's input being matched with this intent and the "responses" are randomly selected from as the chatbot's reply.

2.3. Generative-based chatbots

Generative chatbots are so named because they are able to generate novel responses to the user instead of selecting from a list of ones predefined by the developer. They do this using a process called Natural Language Generation (NLG). This process requires the chatbot to be trained with very large datasets which it uses to determine how to understand and form sentences in the target language. If the dataset is not sufficiently large, it is likely that the chatbot will reply with

grammatically incorrect responses (Adamoupoulous 2020). Generally speaking, generative-based chatbots are better suited to open-domain applications, where the chatbot's main purpose is to entertain the user.

3. Project Specification

3.1. Problem Definition and Analysis

The aim of this project is to produce a chatbot that English language learners can interact with in order to practise their speaking and listening skills. The chatbot should be easily available on desktop as well as mobile devices, to account for the public's desire for learning on the go (Heil 2016). The user will be able to interact with the chatbot primarily through voice and audio, however the application should allow for text display and input in the case of an inability to make use of these features.

The application will offer users the opportunity to speak only about specific topics, those which language learners frequently encounter as beginners (hobbies, food, etc, ...). Within these topics there will be some concept of user progress, such that the user is able to access increasingly more challenging dialogues after completing the previous one. The user will be introduced to key vocabulary before each dialogue, which they will then encounter and be encouraged to use over the course of the conversation. During these dialogues, the chatbot will ask the user a series of questions relevant to the topic, the chatbot will adjust its questions based on the previous responses from the user. There will be a feedback mechanism whereby users have their grammatical errors pointed out to them, at which point they will have the opportunity to correct themselves. Finally, at the end of the dialogue, users will be able to review the transcript of their conversation and will receive a score which reflects their performance.

Below, a user story has been devised giving an imagined use case of the system. This allows key features to be identified and prompts a move towards a more formal definition of the system requirements.

3.1. User Story

Javier has recently started studying the English language, but lives in a small town where there are no English speakers he can practise talking with. He decides to try LangChat, which he hopes will give him the opportunity to train his speaking and listening skills.

He visits www.langchat.co.uk where he is prompted to login. Knowing that he does not have an account, he clicks on a link which takes him to the register page. He enters his personal details and signs up for the application. Immediately he is logged in and presented with the application homepage. He sees a list of topics, each with a number of levels, but does not yet know how to

use the application. He clicks on the ‘About’ tab in the toolbar to learn more about LangChat. After familiarising himself with the application, he returns to the homepage.

He selects the topic ‘Weather’ and sees that only the first level is available, so he clicks on this. He is taken to a new page where he is presented with several vocabulary words. He is familiar with most of them, but one of them, ‘Snow’, he hasn’t seen before. He clicks on the word and hears a voice speak the word. He also sees the IPA transcription and a picture of snow that allows him to understand the meaning of the word.

Now that he is familiar with the vocabulary, he clicks on the ‘Start’ button. He hears the application ask him a question. He clicks on the microphone and seeing that he is being recorded, speaks his reply to his laptop. He hears a positive sound, so knows his answer has been understood, and then he hears another question. This time when he answers, the chatbot says that it doesn’t understand him. He clicks on a button that says ‘Show text’, and sees that he misunderstood the question. He answers again and this time his answer is accepted. In his next answer, he makes a grammatical error and receives a prompt from the application explaining his error and how to correct himself. After correcting himself, his answer is accepted. After a few more questions, the dialogue finishes and he is given a chance to review his conversation.

He returns to the home page, and sees that he has unlocked the second ‘Weather’ dialogue. He navigates to the ‘Profile’ tab and sees a list of the vocabulary words he previously encountered. He clicks on ‘snow’ again to review its pronunciation. Happy with his progress, he decides to logout of the application and return for another dialogue tomorrow.

3.2. Use Case Diagram

It was identified that the application has a single use case, that of the language learner. From this user story, a use case diagram was created to highlight the necessary functionality of the system.

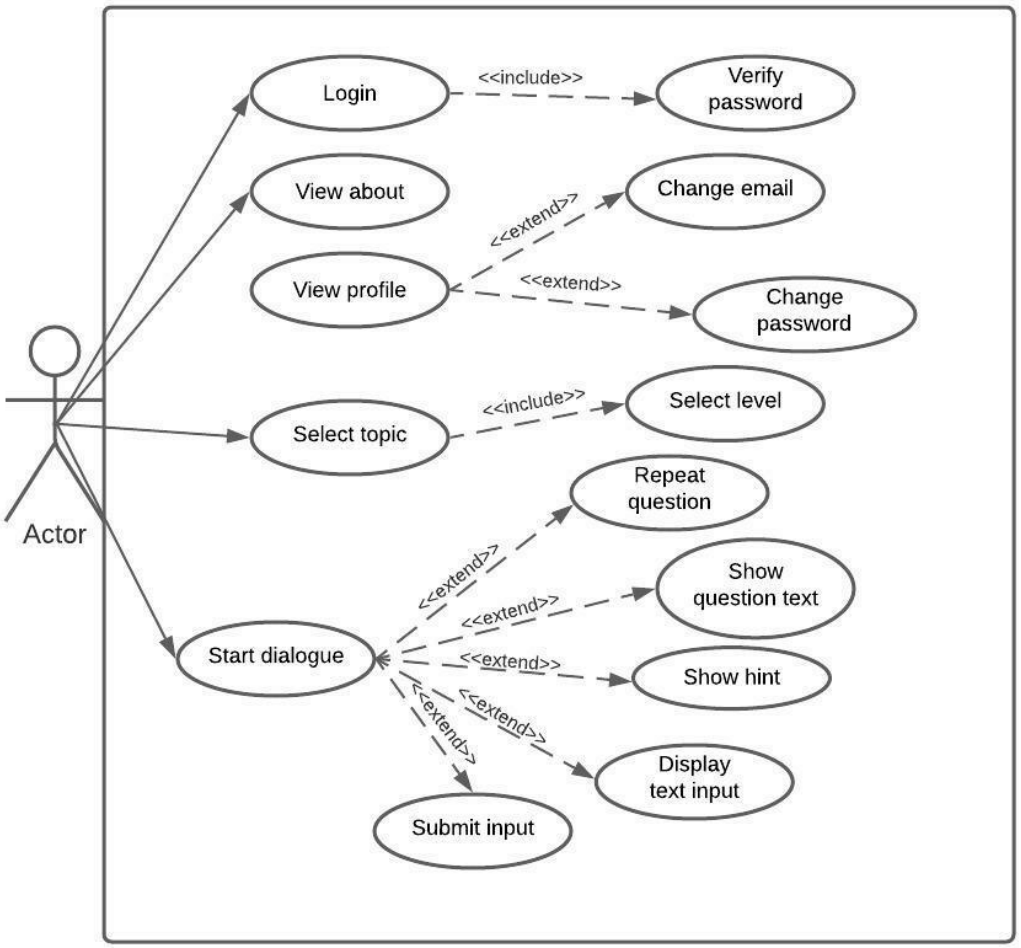

Figure 7: Use Case Diagram showing important requirements

3.3. Activity Diagram

Following this, an activity diagram was created to show the flow of events within the identified use case.

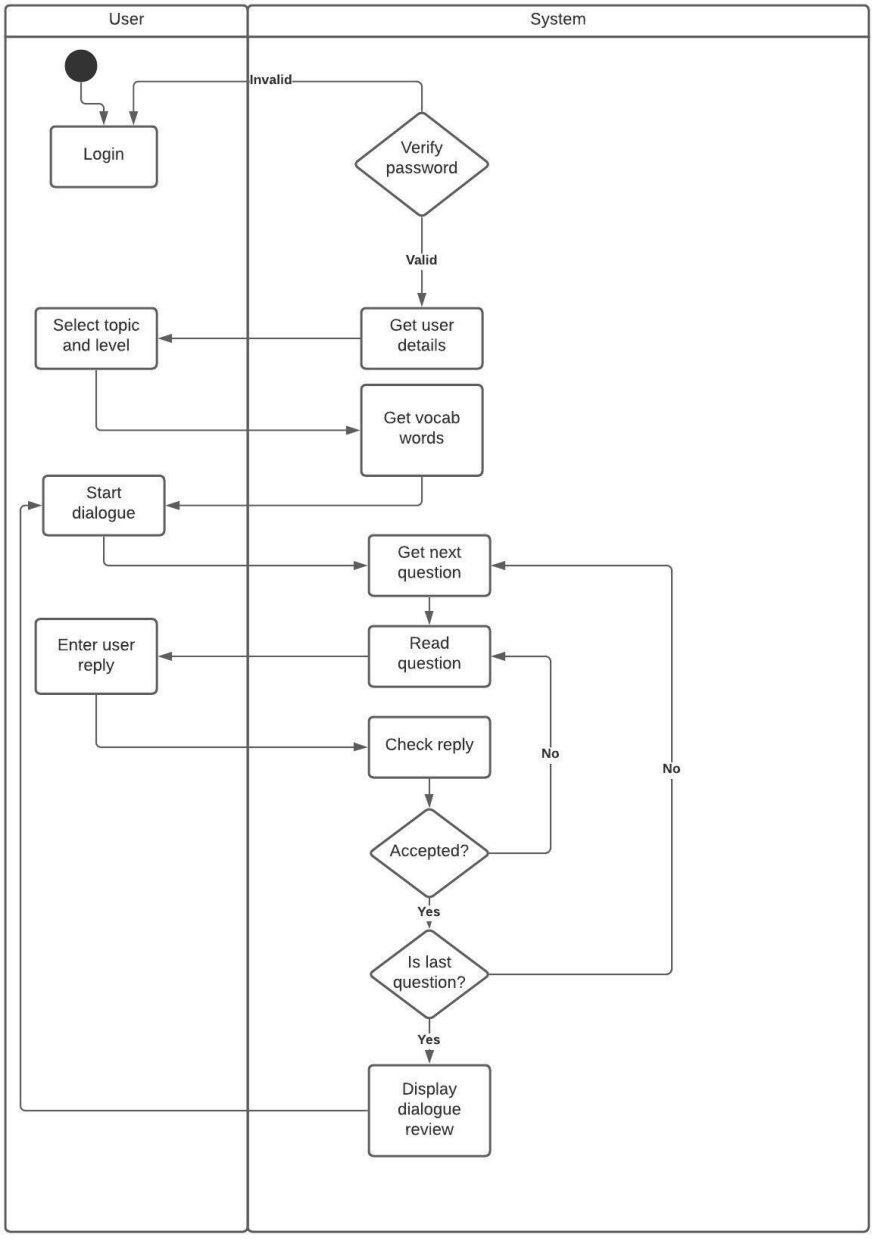

Figure 8: Activity diagram showing flow of events for dialogue

3.4 User Requirements

In order to proceed with development, the key requirements for the system were identified. These include both the core features and services that the user requires from the system (Functional Requirements) and the constraints within which the system must operate (Non-functional Requirements). Each requirement is given a priority from the following:

- **Must** have: Mandatory services that the system must offer
- **Should** have: Important services that are not critical to the system
- **Could** have: Desirable features that may not be implemented

A full list of these requirements can be found at the end of the report in Appendix B.

4. Solution Design

4.1. Architecture Design

Given the requirements detailed above, several essential features were identified that had to be accounted for in the architecture of the application. Firstly, the system had to be easily accessible from a multitude of devices at any time. Secondly, the system should be robust and reliable with few errors, if any, occurring, and swift recovery in case they do occur. Finally the system should be flexible and easily scalable, such that if the decision is made to move the application to a new platform, this can be done without having to rewrite the whole application.

To satisfy these conditions, a client-server architecture was chosen as the basis for the application structure. As one of the most well-known programming paradigms, client-server architecture was desirable in this application because of its high accessibility and good scalability (Schussel 1995). Furthermore, the centralised nature of the server means that it is easy to update the code base whenever necessary, with the effects immediately being implemented for the clients.

More specifically, a 3-tier architecture was chosen to implement the client-server system. The high separation between the three layers (presentation, application and data), increases the reliability of the system as outages in one layer are less likely to impact performance in the others. This is also reflected in the scalability of the application. If needed, the presentation layer can be updated to make the application available natively on mobile devices, for example, without having to make significant changes to the application layer.

Furthermore, it was decided to implement a RESTful API (Fielding 2000) as the back-end of the application. RESTful architecture further increases the separation between client and server, enhancing scalability. Provided the front and back end are communicating via JSON, the client platform can be redeveloped with no need to amend the server.

4.2 Core Tools Used

4.2.1. Python

Python is often the language of choice for building APIs and server side technology (Taneja 2014). Most importantly, there are a number of powerful web development frameworks that can be used with Python. These include Flask and Django, and allow robust server side logic to be developed relatively easily. Further to this there are many other third party libraries which offer utility methods and services, allowing the logic handling code to remain as lightweight and simple as possible. Further to this, Python was created with ease of use and readability in mind, which allows developers to create clear and understandable code (Dierback 2014).

4.2.2. Flask

Flask is a popular lightweight web application framework. Specifically, it is classed as a micro-framework that has few to no dependencies on external libraries. This is in contrast to a web framework such as Django which offers users many pre-built features at the cost of creative freedom. Flask was chosen because it is flexible and concise, also allowing more granular control over essential application features (Grinberg 2018).

4.2.4. JavaScript

For developers creating web applications, there is little choice other than to write the client side in JavaScript. JavaScript is run in the browser application and allows websites to become dynamic and interactive (Crockford 2008). Furthermore, there are a number of libraries that help developers create complex web applications such as React and Angular.

4.2.5. React

React is a JavaScript framework released by Facebook in 2013 that helps developers create complex user interfaces that change over time (Gackenheimer 2015). Developers can use React to create reusable components that can be rendered throughout the application instead of having to recreate each time. There are a number of helper libraries available for React which offer developers helpful utility services. These include Redux, which helps manage a central data store throughout the lifecycle of the application.

The technical details of Flask and React will be discussed in greater detail in the implementation section of the report.

4.3. Technical Details of NLP Services

In addition to these central tools, several other libraries and Application Programming Interfaces (APIs) offering Natural Language Processing (NLP) were used, without which development of the application would not have been possible.

4.3.1. Program-Y

Program-Y is a chatbot framework written in Python. Although this library comes with a lot of functionality and pre-programmed chatbots, in this project it was used to parse AIML documents and offer an interface for the user to communicate with them.

4.3.2. GrammarBot

GrammarBot is an online service with an API that checks a string of English text for grammatical errors. Grammar checkers in general use three main approaches to determine the "correctness" of a sentence (Naber 2003):

- Syntax-based checking, where each sentence is parsed, decomposing it into a tree-like structure according to predefined grammar rules. This method has the benefit of providing the user with feedback, informative messages can be returned with the result according to which grammatical rule has been violated. However, in order for this method to function perfectly, a completely defined grammatical parsing system is required for the language in question, which is yet to be achieved in any language. As a result these types of grammar checkers are never perfect.
- Statistics-based checking, which uses a large dataset is used to compare against the sentence being checked. The dataset will contain many correct sentences where each word is annotated by its part of speech (POS). The checker then verifies that the input in question matches one of these sentence patterns. The downside with this approach is that the user receives no feedback on the mistake.
- Rule-based checking, where each word of the text is assigned a POS, then the text is matched against a set of pre-defined error rules. If an error occurs, the checker is able to provide an explanation of the error like in syntax-based checkers. A rule-based checker will never be complete, but is able to give feedback and is easily extendable to offer the level of comprehension required.

GrammarBot claims to be created by " team of experts in Natural Language Processing, Machine Learning, and AI with expertise in language models, deep learning, sequence labeling, and natural language parsing", so it can be assumed that the API uses some degree of the statistics-based approach to harness artificial intelligence. Also each error that is returned by the API comes with an informative explanation of the error, so we can assume that a syntax-based or rule-based approach must be implemented to facilitate this. In conclusion, the service seems to be a blend of the three grammar checking methods detailed above.

4.3.3. SpeechSynthesis API

This is a service provided by Google, which accepts a string of text, then returns the audio of the text as if spoken in the specified language. Speech synthesis is generally achieved by two distinct methods:

- Bottom-up approach, where the audio is generated from scratch using knowledge of how speech is produced in the larynx.
- Concatenative approach, where recordings of real speech are made, cut up and recombined to form the word which is to be synthesised

Generally speaking, the concatenative approach is superior, despite the argument that it is not true speech generation as the output is made up from pre-recorded speech (Taylor 2009).

Google's service uses the concatenative approach. The input text is broken up into separate words, each of which is looked up in a database to see if a corresponding audio file exists. If not the word is further broken down into smaller chunks, the audio of which can be combined to form the word in question. Furthermore, Google then uses artificial intelligence to generate realistic sounding intonation depending on the context of each word.

4.3.4. Speech Recognition API

Also a Google service, this API accepts an audio recording in a specified language and returns whatever text was recognised. Speech recognition methods can generally be separated into three categories (Anusuya 2009):

- Acoustic Phonetic approach, where the speech signal is spectrally analysed and then these measurements are used to describe the acoustic properties of the signal. These properties are then used to match the signal to a word.
- Pattern Recognition approach, which involves two steps. Firstly the system is trained using well defined training samples, which consist of clear audio of a word and a label which contains the spelling of the word. Then the unknown signal is compared to these samples and given a score based on how well the signals match. The label on the sample with the highest match score is then the most likely to be the transcription of the signal.
- Artificial intelligence approach, which is a hybrid of the two approaches detailed above.

This last technique is the one harnessed by Google to provide their Speech Recognition service. While there has been some debate about whether speech recognition services are accurate enough to be used with ESL learners, Neri (2003) concludes that acceptable levels of recognition of non-native speech are achievable in carefully designed systems.

4.3.5. text-to-ipa

This JavaScript library was used to obtain the International Phonetic Alphabet (IPA) transcriptions of words. This library simply contains a text file with the transcription of 130,000 words in American English. The library performs a simple lookup in this text file when a transcription is requested.

4.3. Component Diagram

A component diagram was created to demonstrate how the application services interact with each other. White components are to be implemented over the project, whereas blue coloured components are third-party services.

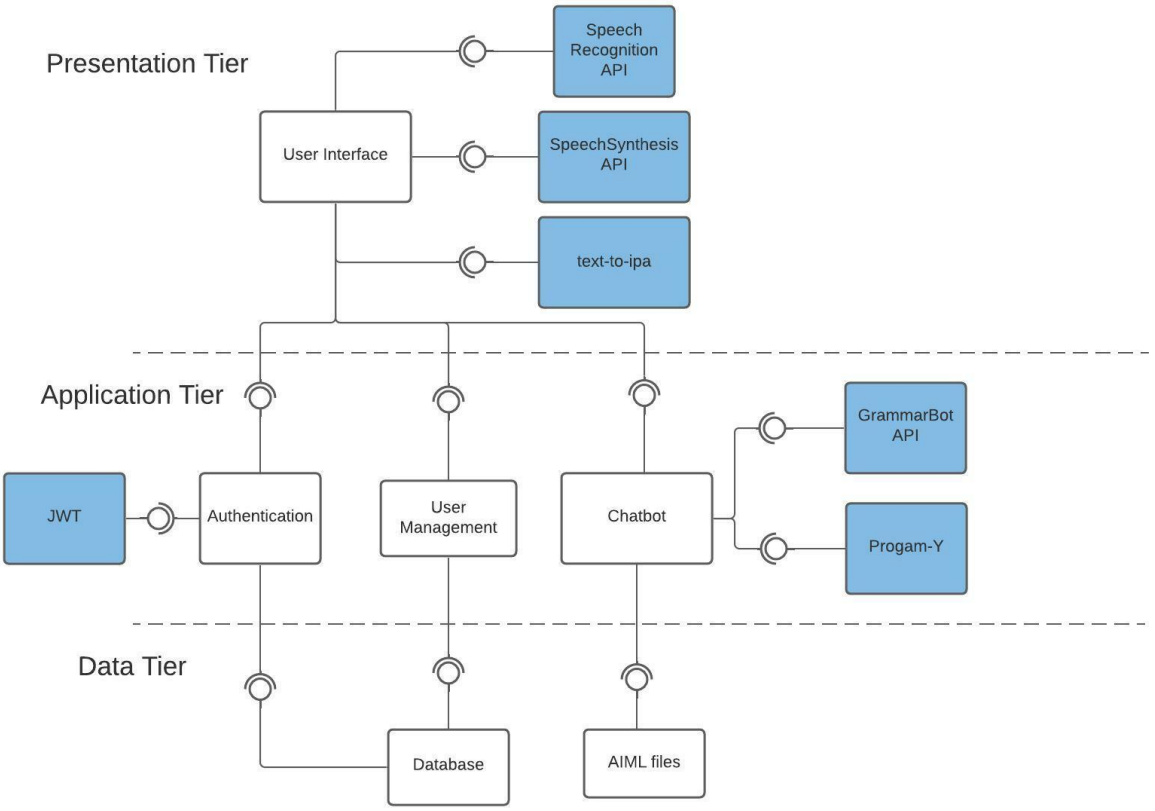

Figure 9: Component Diagram showing core components of the application

As can be seen in the diagram, the API in the application tier is to provide three services to the presentation tier:

- Authentication: ensuring users are only able to access and update data corresponding to their own accounts.
- User management: supplying user details and managing their progress as they use the application.
- Chatbot: all dialogue functionality independent of user accounts.

4.4. Database Design

To store application and user data, SQLite was chosen as the database of choice. SQLite is a lightweight database management system (DBMS) that operates without a server and is instead embedded in the end application. It is atomic, consistent, isolated and durable (ACID compliant), making it a reliable choice for a DBMS (Bhosale 2015). SQLite's developers suggest that it is well suited to websites that receive less than 100,000 users per day (SQLite, 2021). This number is far greater than the expected number of users for the application and as such was not seen as a problem.

A database entity diagram was created to show the tables that would be required in the database. The schema conforms to Boyce-Codd Normal Form, which reduces data redundancy and minimises modification anomalies (Codd 1974).

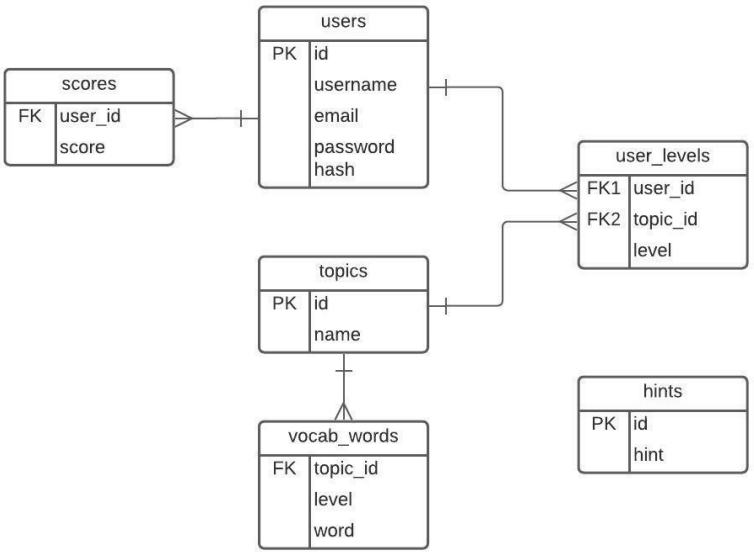

Figure 10: Entity Relationship Diagram showing tables in the database and their relationships

5. Project management

In order to ensure timely development of the project, a schedule was devised detailing the dates of expected completion of key milestones. This is represented in the Gantt chart shown below.

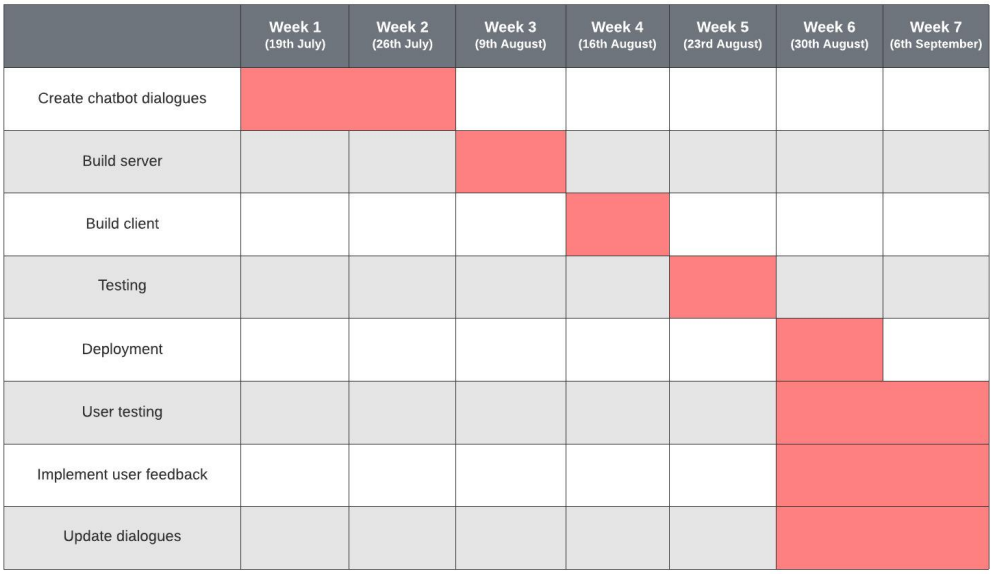

Figure 11: Gantt Chart showing expected project timeline

It was forecasted that creation of the dialogue scripts and AIML files would be a labour intensive process, and as such two weeks were allotted for this task. Building the server and the client were each expected to be completed within a week, as was the testing process. Deployment would be completed at the beginning of week 6, with user testing beginning immediately after. Throughout the user testing process, feedback would be implemented where viable and AIML files would be updated, with unrecognised responses registered, while users were talking with chatbot.

A version control software named Git was implemented to manage development of the project. This is good practice as it allows the developers to try adding new features to the application, always having the opportunity to revert the application to a previous state. Furthermore, these versions were regularly committed to a remote repository on GitLab. This has the added benefit of increased security, because if the development machine is lost or fatally damaged, the application code will always be available online.

6. Implementation

6.1. Chatbot

The first element of the application to be developed was the chatbot. Before any coding took place, the dialogues were planned using a simple text editor. In order to plan the dialogues, three topics were chosen that were likely to be encountered by a beginner language student: weather, hobbies and food. Within these categories, key words were identified that speakers would need to know in order to converse about these topics. These words were then split into three levels, and were used to prepare several questions, constituting one 'level' of dialogue. An example is given below for the first level of the weather topic.

```
 1 Weather
 2
 3 Words: Rain, Sun, Snow
 4
 5     1 How is the weather where you are?
 6     (At the moment, it is ___)
 7
 8         2 Is there often ___ where you live?
 9         (Yes, it is often ___ here
10
11     3 Would you like to live somewhere with more or less sun?
12     (I would like to live somewhere sunnier)
13
14     4 Have you ever seen snow before?
15     (Yes, I have seen snow before)
16
17         YES: 5 Do you enjoy the snow?
18         (Yes, I do enjoy snow)
19
20         NO: 6 Would you like to see snow?
21         (Yes, I would love to see snow)
22
23     7 Which type of weather do you enjoy the most?
24     (I enjoy ___ weather the most)
25
```

In this extract of the file, there are several key features to be identified. The name of the topic is seen on line 1, the key words for the dialogue are on line 3, a question with a unique question number is seen on line 5 and finally a hint for that question is seen on line 6 in brackets. Indentation means that the indented question in some way refers to the previous question. The question on line 8 references a word used in the user's reply indicated by '___', while the questions on lines 17 and 20 are to be given to the user depending on whether they have given a positive or negative reply respectively to the previous question. This dialogue planning format was used to plan three dialogues for each of the three topics.

Once the semantic content of the chatbot had been established, the corresponding AIML files were created. Each dialogue was given its own AIML file and kept in a folder based on its topic.

One of the greatest challenges to creating these AIML files was to consider all possible replies a user might give in response to each question. In order to proceed quickly with development, it was decided that only a few response patterns would initially be accounted for. A logging system would later be added to the application so that a moderator could review any rejected responses from the user and add their pattern to the AIML file if appropriate.

```
<!--How is the weather where you are now?-->

<category>
    <pattern># IT IS # <set>weatherverbs</set> #</pattern>
    <that>HOW IS THE WEATHER WHERE YOU ARE NOW</that>
    <template>2 Is it often <star index="3"/> where you live?</template>
</category>

<category>
    <pattern># IT IS # <set>weatheradjectives</set> #</pattern>
    <that>HOW IS THE WEATHER WHERE YOU ARE NOW</that>
    <template>2 Is it often <star index="3"/> where you live?</template>
</category>

<category>
    <pattern># THERE IS # <set>weathernouns</set> #</pattern>
    <that>HOW IS THE WEATHER WHERE YOU ARE</that>
    <template>2 Is there often <star index="3"/> where you live?</template>
</category>
```

Above is an excerpt of an AIML document which shows all the considered responses for the question “How is the weather where you are now?”. The category tags were grouped depending on which question they were answering, as seen by the commented line at the top of the image. The patterns were made to be as flexible as possible while guaranteeing some degree of correct English. In each response, a # wildcard was added to the beginning and ending of every pattern to ensure that even if the user gave a more elaborate response, it would still be accepted provided it contained at some point the core pattern. For example, to this question the user could simply answer “It’s sunny”, or they could say “At the moment it’s sunny, but I think it will rain soon”. Wherever possible, the <srai> tags were used as detailed earlier in the report to implement recursion and keep the AIML files as concise as possible.

In order to test these AIML documents, a short Python script was created using the Program-Y library to parse the files and offer a chatbot interface to simulate dialogue. The Program-Y library implements a logging system such that any syntax errors in the document are detailed in a log file when the script is run. The function used for getting the chatbot’s reply is shown below.

```
files = {'aiml': ['/home/george/Desktop/projects/chatbot/programy/aimls'],
         'sets': ['/home/george/Desktop/projects/chatbot/programy/aimls/sets']}

my_bot = EmbeddedDataFileBot(files, defaults=True)

first_question = "How is the weather where you are now?"

response = my_bot.ask_question("Say " + first_question)
print("\nBot: %s" % response)

while True:
    user_message = input("You: ")

    # get chatbot response
    response = my_bot.ask_question(user_message)

    # message if no reply from chatbot
    if response == None:
        response = "I'm sorry I don't understand you..."

    print("\nBot: %s" % response)
    if 'goodbye' in response:
        break
```

It should be noted that the chatbot can only reply to the user, it cannot start the conversation. In order to facilitate this, the chatbot needs to be told the first question. In the AIML file a utility pattern was created as seen below.

```
<category>
    <pattern>SAY *</pattern>
    <template><star /></template>
</category>
```

This pattern simply echoes the contents of the * wildcard. To use this pattern, the chatbot is told to “Say” the first question, after which it will respond with the question. This is an important step as it allows the chatbot to be “aware” of the question, simply printing it would mean that it cannot be referenced in the <that> tags of later questions.

This method of interacting with the chatbot is suitable for testing the dialogue flow of the AIML files, however it requires that the Python script is running continuously while the user is interacting with the chatbot. This is incompatible with a RESTful API, and so a stateless interaction method had to be devised to allow the chatbot to be provided over this kind of server.

This was achieved with the utility pattern described above. In the stateless version, a function was created that required two arguments, the previous question and the user’s reply. Now the first instruction to the chatbot would be to repeat the previous question, allowing it to be referenced by a <that> tag, then it would be passed the user’s reply which it can search for in the AIML file and give the appropriate response.

```python
def get_response(user_message, last_bot_message):

    files = {'aiml': ['/home/george/Desktop/projects/chatbot/programy/aimls/weather'],
             'sets': ['/home/george/Desktop/projects/chatbot/programy/aimls/sets']}

    # create chatbot
    my_bot = EmbeddedDataFileBot(files, defaults=True)

    # make chatbot repeat previous question
    my_bot.ask_question("Say " + last_bot_message)

    # get chatbot response
    response = my_bot.ask_question(user_message)

    return response
```

One limitation of this stateless approach is that the chatbot can only “remember” one question previously, it does not have access to the entire conversation from the user’s point of view. However, the dialogues were designed with this in mind, and it would never be necessary for the chatbot to retrieve any information from the dialogue earlier than this. As well as this, the chatbot needed another function which would fetch the first question from the AIML file. This function takes as arguments the topic and the level of the required dialogue, which are then used to retrieve the opening question from an AIML document.

Now that the chatbot had been prepared to be delivered over a stateless server, the next step was to begin developing the API that would form the back-end of the application.

6.2. Server

As previously mentioned, Flask Python was chosen as the framework for creating the server. Below, the file structure of the API is shown to demonstrate the structure of the application.

```
/chatbot-api
    /app
        __init__.py
        auth/
            __init__.py
            views.py
        user/
            __init__.py
            views.py
        chatbot/
            __init__.py
            views.py
        decorators.py
        email.py
        help.py
        models.py
    config.py
    requirements.txt
    chatbot_dev.db
    wsgi.py
```

Flask operates by creating a Python object, usually named 'app', through which instantiates the application. This implementation uses the common application factory pattern to provide the application object. The app object is created in wsgi.py, the entry point for the server. The create_app function is imported from the app module, where it is defined in the __init__.py file.

The client is able to interact with the API by invoking routes, which are also defined in the views.py files within the app module. A route takes the form of a function with a decorator that assigns it a URL. Routes invoked by making a request to the corresponding URL, thus causing the code within the function to run. An example of one of these routes is shown below.

```
@chatbot.route('/api/get_first_question/<topic>/<topic_level>')
def get_first_question(topic, topic_level):
```

In the URL defined here, the subsections enclosed in angle brackets are dynamic, so that when specified, the values in these portions are passed as arguments to the function that the route invokes. For example if a request is sent to /api/get_first_question/weather/1, the get_first_question function will receive "weather" in the topic argument and "1" in the topic_level argument.

As detailed in the component diagram (Figure 9), the API offers three separate services: authentication, user management and the chatbot. These are accounted for in the three folders auth, user and chatbot. In Flask, these are referred to as blueprints, which are logical structures representing a subset of the application. In each of these three folders, the __init__.py file registers the blueprint with the application, and the routes in the views.py files attach routes to these blueprints as seen in the code snippet above, where a route is attached to the chatbot blueprint.

6.2.1. Chatbot

The first blueprint to be completed was the chatbot segment. This segment contained the two functions detailed above, one to fetch the first question of each dialogue and one to get the chatbot's response to user input. Also a route was created to send the client all the vocabulary keywords for a given dialogue. In order to do this, it was necessary to communicate with the database.

As specified in the requirements, it was necessary to supply a hint for each question whenever the chatbot replies. In order to solve this, each chatbot reply starts with the question number, this can be seen in figure AIML document extract, which is then stripped from the question before it is sent to the client. This id number allows the API to query a table in the database retrieving the corresponding hint. It may seem more sensible for the chatbot to return only the

id number and then fetch both the question and hint from a table in the database. However, as the chatbot sometimes references a word from the user's previous reply in the next question, this was not possible.

Within the get_response route, the grammar check also takes place. After the response has been retrieved, or no response has been retrieved, from the chatbot, the user's response is sent to the GrammarBot API, where a list of grammatical errors are returned if any. Regardless of whether the user's reply is accepted, the grammar list is sent in the response to the client, where it can be displayed to the user.

6.2.2. Communicating with the database

As an interface between Python and the SQLite database, SQLAlchemy was used. This tool maps database entities to Python classes, on which queries can be run in the form of a function. The class on which the query is run is defined in the models.py file seen below. This file contains classes which represent the entities in the database. The definition of the Topic class is shown below as an example.

```
class Topic(db.Model):
    __tablename__ = 'topics'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), index=True, unique=True)
    vocab_words = db.relationship('VocabWord', back_populates='topic',
                                  lazy='dynamic', cascade='all, delete-orphan')
```

When the application is run, SQLAlchemy will look for tables in the database provided in configuration and try to match them to the models. If a table with a matching __tablename__ is not found, SQLAlchemy will create it. The class attributes id and name directly correspond to fields in the topic table in the database. We can use SQLAlchemy to retrieve rows from the database by using these values. An example is given below.

```
topic = Topic.query.filter_by(name='weather').first()
```

In this line of code, SQLAlchemy generates an SQL query to select all rows in the topics table where the name field is equal to the string contained in the topic variable. The SQL statement produced by SQLAlchemy is equivalent to:

```
SELECT *
FROM topics
WHERE name='weather';
```

After the query has been executed, the .first() method in the Python code selects the first row from the results and returns it as an object. Columns from this row are then mapped to attributes which are then accessible on the object.

In the case of the vocab_words attribute in the Topic model, it is not representing a column in the table but instead it refers to the one to many relationship between the topics and vocab_words entities. This creates a join between the two tables and allows a row in the topics table to reference all the vocab_words rows with a matching foreign key. This is accessed in the same way as detailed above, as an attribute of the object. An example of this type of operation is shown below.

```
vocab_words = Topic.query.filter_by(name='weather').first().vocab_words
```

Here a row is retrieved from the topic table and mapped to an object, then by accessing the vocab_words attribute, a new SQL statement is generated according to:

```
SELECT *
FROM topics
JOIN vocab_words
ON topics.id = vocab_words.topic_id
WHERE topics.name = 'weather';
```

The results of this query are each mapped to a VocabWord object, another class defined in the models.py file, and placed into a list which is available as an attribute on the Topic object.

Another Python library that was used to make interacting with the database easier was Flask-Migrate. This library is an extension that allows Flask applications to take advantage of database migrations. A migration is an update to the database's schema which can be applied or rolled back, making it somewhat analogous to source control for the database. Flask-Migrate compares the structure of the SQLAlchemy model classes to the tables in the database, and makes changes to the schema accordingly. This means that if the developer wishes to add a new attribute to one of the models, they only have to update the class in models.py, run a migration, and a new column will have been added to the corresponding table in the database. This is necessary because SQLAlchemy creates tables according to the models only when the table does not exist. Without migrations, the developer must delete the table from the database in order for it to be recreated with the updated schema, which of course has the unintended side effect of destroying all the data contained in the table.

6.2.2 Authentication

In order to track the user's progress in each topic, it was necessary to implement a system with user profiles where the user could register for an account and login. This functionality was handled by the authentication blueprint. To register, users are required to provide an email, username and password. Provided the email and username are unique, the user's details are registered in the database. Due to the fact that it is common for internet users to reuse the same password for multiple websites, it is considered best practice to store the hashed form of the password in the database. This way if a hacker gains entry to the database, they are unable to use the value in the password field. This functionality was implemented as a method on the User class. When the password attribute is set on a User object, it is automatically hashed using the generate_password_hash function from the Werkzeug library (a Flask dependency).

Logging into the application is handled by validating the user's email and password using another Werkzeug function, check_password_hash. If the supplied password is valid against the hashed password in the database, the user is sent a JSON Web Token (JWT), which is generated by the JWT library. This token can then be used by the client to access protected routes when sent in the headers of the request. This protection is implemented through the use of a decorator named token_required, which is defined in the decorators.py file. This decorator intercepts any calls to a protected route and checks the validity of the token. If invalid, a 401 error (unauthorized) is returned before the route is invoked, otherwise the request proceeds as normal.

An example of a route that implements this decorator is the change_email route. This ensures that only users who are already logged in are able to change their email address. The same applies for changing their password.

The final authentication route to mention is the forgot_password route. This allows users who have forgotten their password to receive a temporary password to their registered email address, which they can use to login and create a new password. In order to achieve this, functionality had to be introduced that would allow the application to send emails. Simple Mail Transfer Protocol (SMTP) was used along with the details of a gmail account linked to the application to send emails programmatically. Furthermore this process was threaded such that the application would not be blocked while the email is being sent. The send_email function is defined in the email.py file.

6.2.3. User Management

All routes in the user management blueprint required the authentication token to be sent with the request as they all require access to the current_user object made available by the token_required decorator. The routes in this blueprint allow the client to get information about a user account, update a user's level within a specified topic, and register a new dialogue score, which is used in calculating the user's average accuracy over all completed dialogues.

6.3. Client

The client side of the application was built with ReactJS, which as previously mentioned is a JavaScript library that is used to build interactive user interfaces. React allows developers to create custom components that can be reused throughout the application. These components may themselves render other custom components or base HTML components such as a <button> or an <input>. This creates a component hierarchy with a root component at the base of the application. Each component can manage its own state, an attribute that contains data used by the component. When a component renders a child component, it can pass data to the child, which is then accessed through the props attribute.

The application was created using the create-react-app command exposed by the developers of React. When run in the terminal, this command creates the basic file structure of a React project and initialises a Git repository to manage source control.

The application also makes use of Redux, a JavaScript library that allows applications to define a centralised data store, which is accessible from any component. This reduces the complexity of the application as components do not have to pass as much data between them, making the application easier to develop and maintain. The Redux store contains actions and reducers. Actions are functions called by the application components, which then pass control to the reducers, which update the data held in the store.

The design pattern used to build the interface was the container-component pattern. In this pattern, the application is made up of higher order containers, which usually align with the pages of the application. As detailed in the use case diagram in Figure 7, there were to be four main pages that the user could navigate between: the home page, the authentication page, the about page and the profile page. The containers receive the majority of the data in the application, which they then pass to the children components that are rendered within them. This is achieved by connecting the containers to the Redux store. They also define most of the core data transforming functions that are used throughout the application. The components on the other hand take a more presentational role, receiving data through the props attribute and defining some utility functionality. This pattern is beneficial for separating concerns, having the data

managed by higher order elements and the presentation managed by lower order elements. It is also beneficial for testing purposes, which will be detailed in the next section of the report.

Below is a class diagram that shows the relationships between the components and containers of the application. The blue boxes represent containers, while the white boxes represent components. An arrow between two boxes means that a box renders the box it is pointing to as a child. The variables and functions on the arrow are passed from the parent to the child as props. Functions inside each box are defined within the class, however the variables in the diagram are attributes of the class' state object. This modification to the class diagram allows React applications to be modelled more insightfully. It should also be noted that purely presentational components with no functionality defined within them, such as the Button and Input components have been omitted from the diagram to aid readability.

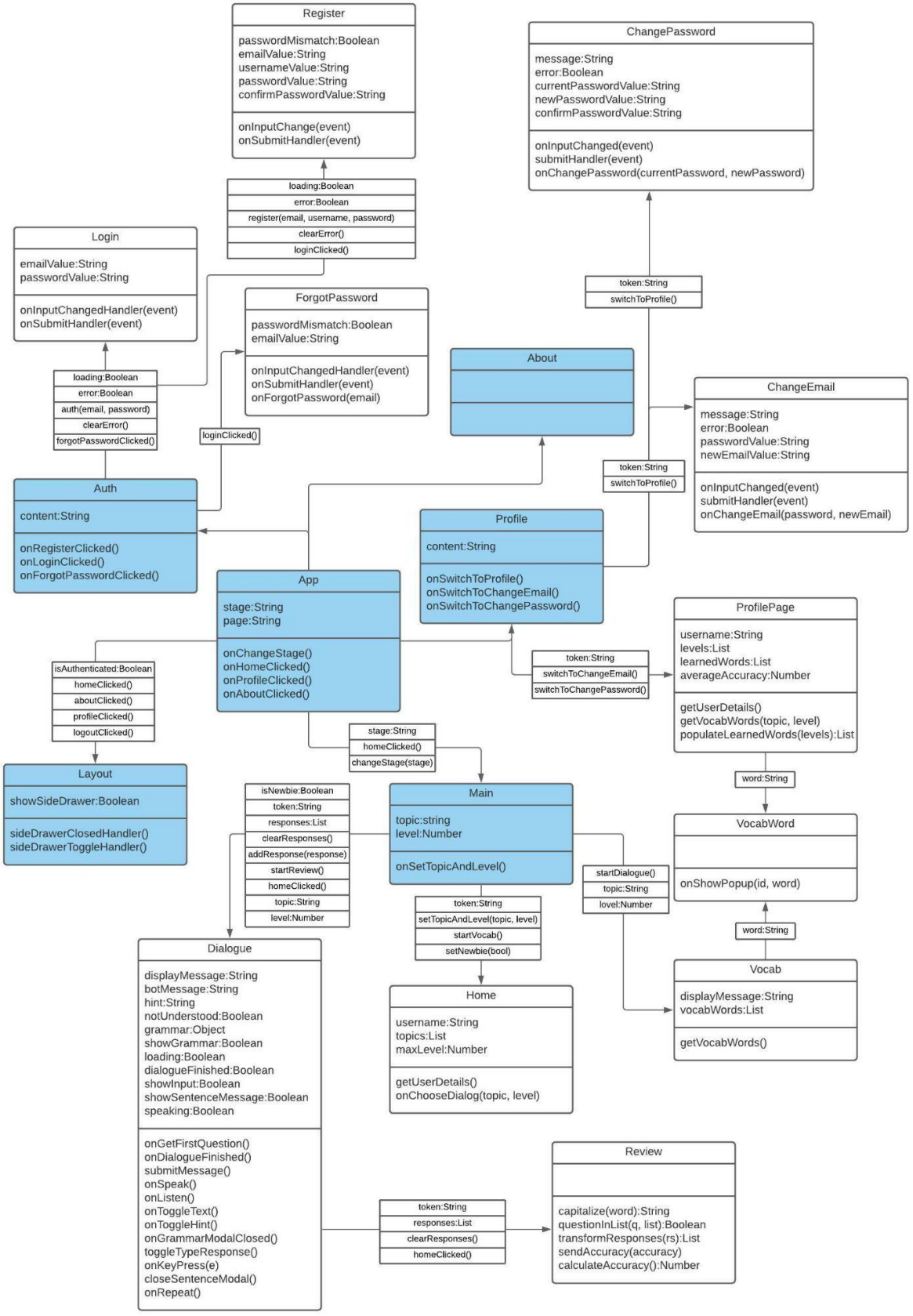

Figure 12: Class diagram with amended format to display React container and component relationships

The App component is the root container that conditionally renders the children containers depending on the state of the application. If the user is not authenticated, this defaults to the

Auth container. An account of the contents of each container follows below, excluding the About container, which contains no functionality and only presentational components.

6.3.1. Layout Container

This is a container that wraps another container, attaching navigational components which depend on screen size. On a desktop device, navigation is performed through a toolbar at the top of the page. On a mobile device, due to limited horizontal space, navigation is performed through a collapsible side drawer. This ensures the application remains convenient to use on mobile devices. Images showing the two navigational systems are included below.

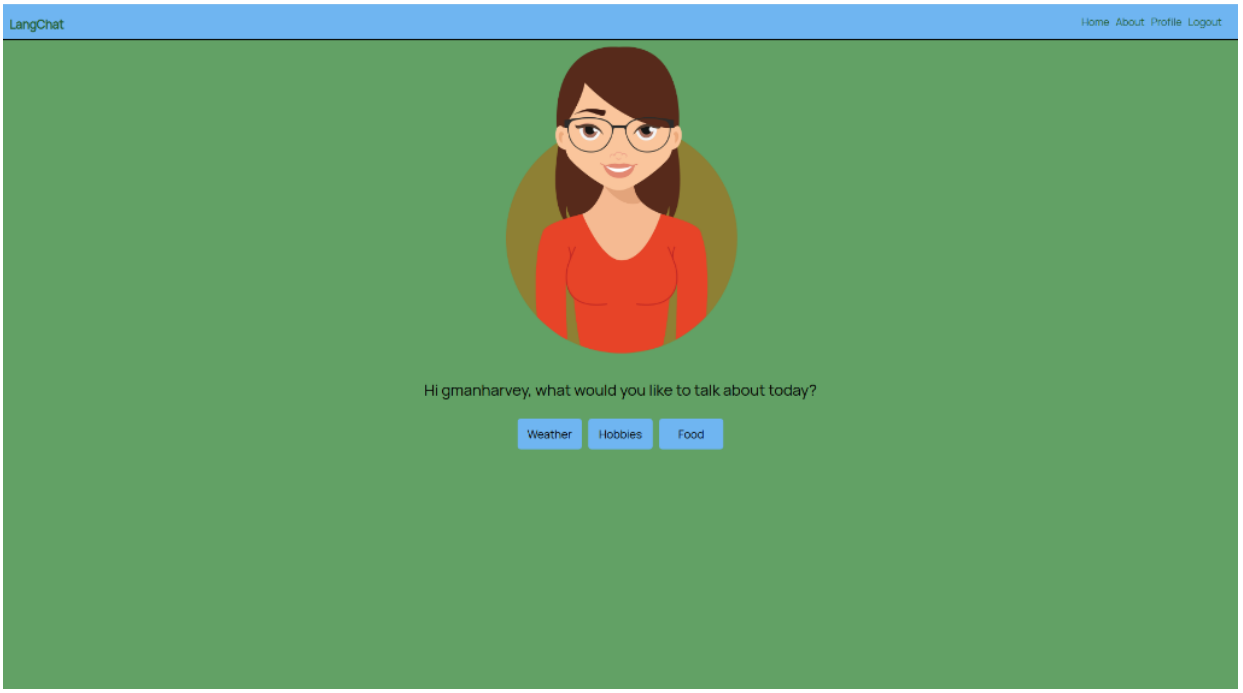

Figure 13: Desktop view of LangChat homepage

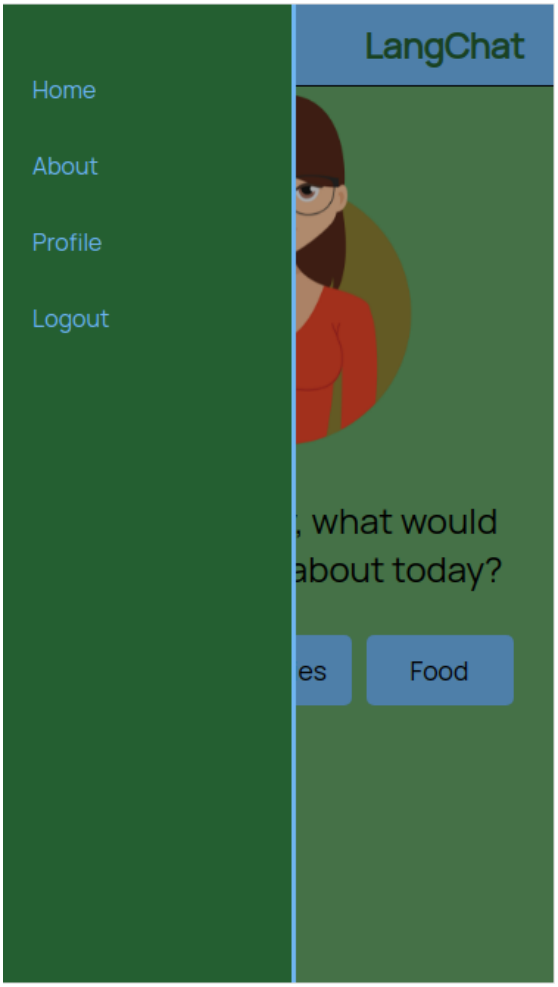

Figure 14: Mobile view of LangChat homepage with navigation sidebar opened

6.3.1. Auth Container

In order to login to the application, the user enters their details in the fields shown below, the values of which are passed to a login function. This login function is defined within the Redux store and makes a call to the /auth/login API endpoint. If the user's details are validated, the client receives a token in the server response. The login function then stores this token on the user's machine using the browser's localStorage API. This token can then be used in future requests to protected API routes that access or update database information specific to the user. The presence of the token in the local storage indicated to the client that the user is logged in.

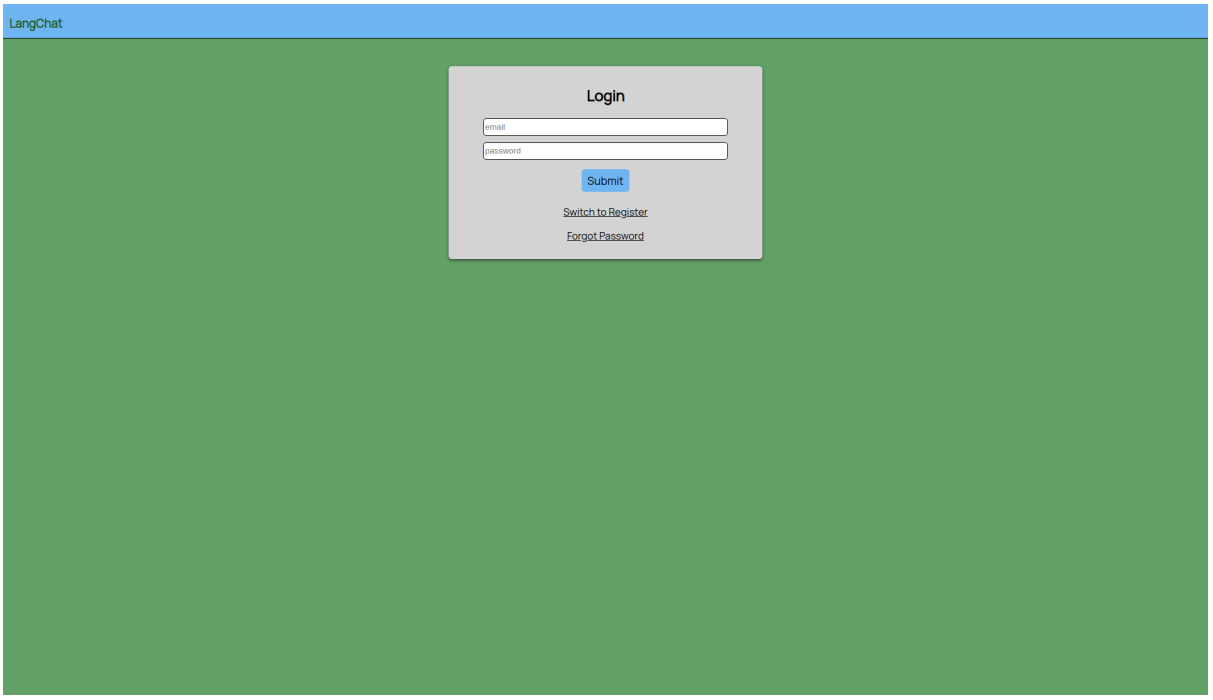

Figure 15: LangChat authentication screen

When the application is opened, an automatic login function is called by the root App container. This function checks local storage for the presence of a token. If a valid token is found, the user is automatically logged in, meaning they are not forced to log in every time they use the application. The token is sent to the client along with an expiration time. When this time is reached, the token will no longer be valid on the server and the client application clears the token from the user's machine. This is a safety measure that prevents users from remaining logged in on machines they are no longer using. Of course the user also has the option of logging out manually, a function that deletes the token in local storage.

6.3.2. Main Container

The Main container as the name suggests holds most of the functionality of the application. It renders three components conditionally, the Home, Vocab and Dialogue components.

The Home component is the first thing the user sees after logging into the application. It displays the dialogue topics, each of which hold a dropdown menu containing buttons that lead to the three dialogue levels.

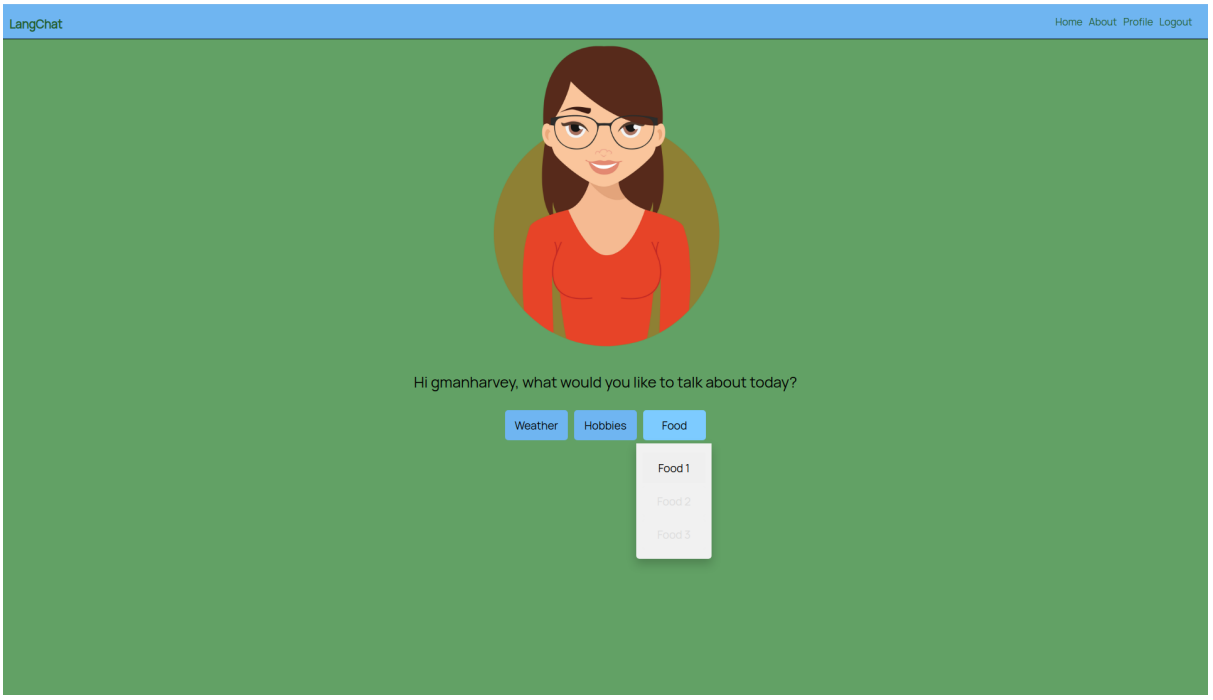

Figure 16: LangChat homepage showing topic dropdown menu

The buttons that lead to the second and third dialogues of each topic are conditionally disabled depending on the user's level. This information, along with the user's username shown in the display message is retrieved from the database by making a request to the get_user_details API endpoint when the Home component is mounted on the DOM.

The Vocab component prepares the user for the dialogue by showing the key vocabulary words defined in the database. They are retrieved upon mounting the component with a call to the get_topic_words/<topic>/<level>. Each of the words is rendered as a button which reads the word through the SpeechSynthesis API and displays a popup when clicked. The popup contains an image of the word's definition, which is stored in the assets/ folder, and the IPA transcription of the word. This is obtained by using the text-to-ipa library's lookup function.

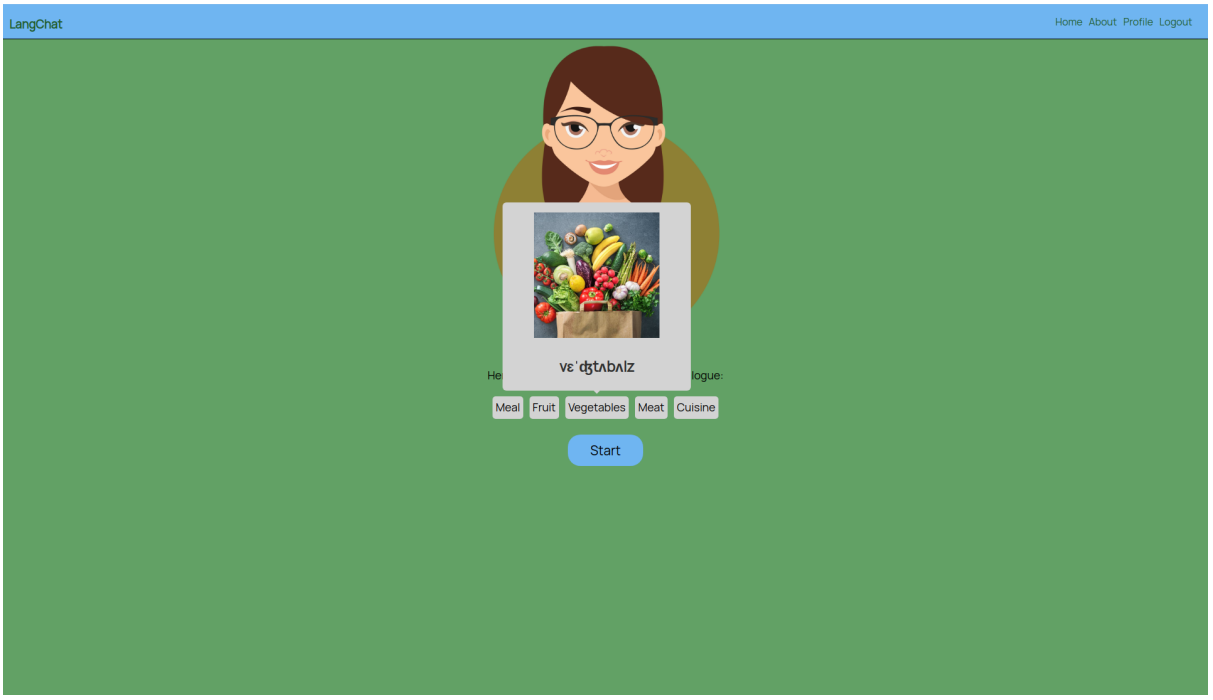

Figure 17: LangChat vocab page showing key word information

The dialogue component is the most complex component in the application. It is responsible for collecting the user's input to each question and presenting the chatbot's responses to the user. The pipeline through which data moves between the dialogue component and the API is shown below.

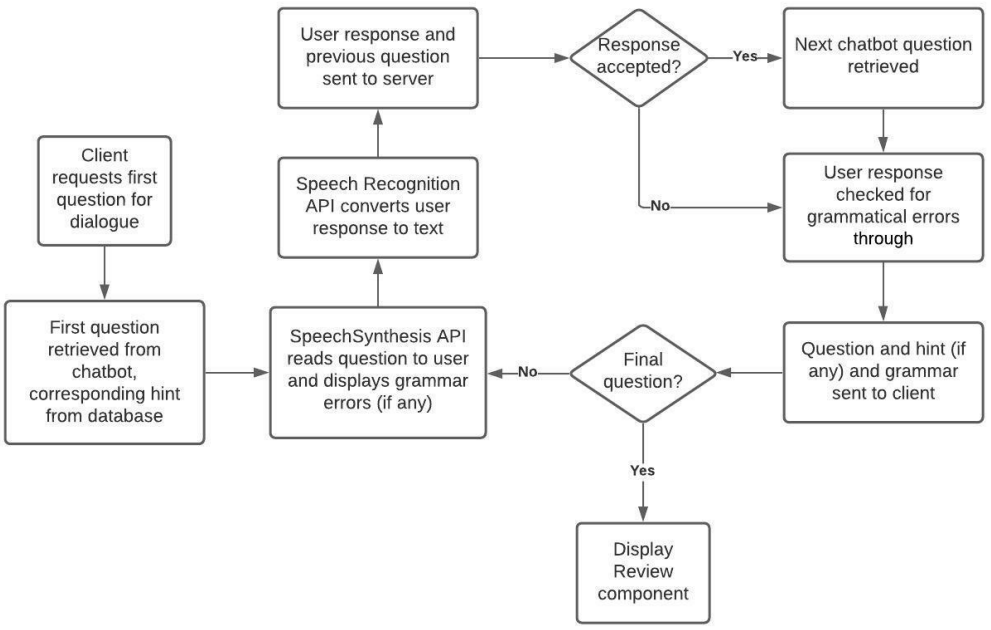

Figure 18: Diagram showing data flow during dialogue

The client retrieves the first question of the dialogue by making a request to the get_first_quetion/<topic>/<level> API endpoint. User messages are sent to the 'get_response/<topic>' endpoint. The topic parameter in the URL tells the route which AIML folder it should look for a response in. In the body of the get_response request are the user's message and the last question that the client received as detailed earlier in the report.

Throughout the duration of the dialogue, the user has the options indicated by the buttons in Figure 19. These are displaying the text of the question, in case the SpeechSynthesis API is not available, repeating the question audio, displaying a hint or inputting text via the keyboard in case the Speech Recognition API is not available. If the user's reply is not understood, the text returned by the Speech Recognition API is displayed, so the user is aware if their reply has been correctly transcribed. This is demonstrated in the image below.

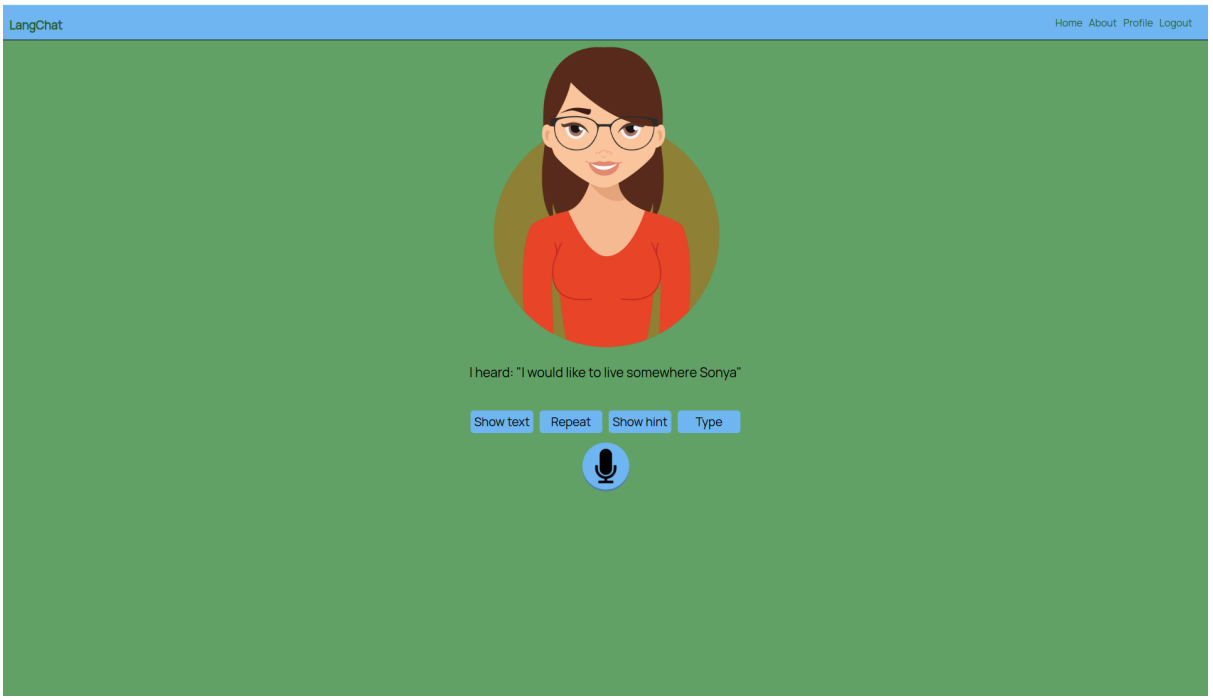

Figure 19: LangChat dialogue screen

In this example, the user had spoken "I would like to live somewhere sunnier" into the microphone, however the Speech Recognition API transcribed this incorrectly as "I would like to live somewhere Sonya. Repeated tries, all with a native English accent, yielded the same result. Should this situation arise, the user has the option to bypass the speech recognition and type their reply.

If any grammatical errors are sent in the get_response response, the Dialogue component displays a modal detailing the error. The modal is displayed whether or not the user's reply has been accepted by the chatbot. This behaviour is seen below.

Figure 20: LangChat grammar correction feature

Once the final chatbot reply has been delivered to the user, the Review component is rendered as a modal. The Review component displays all the answers given by the user, grouped by the question they were answering. An accepted reply is coloured green, an accepted reply with a grammatical error is orange and a rejected reply is red. A score representing the user's accuracy throughout the dialogue is also displayed. This is a percentage calculated by the following equation:

$$100 \times \frac{n_{green} + \frac{n_{orange}}{2}}{n_{total}}$$

Where n_{green} and n_{orange} are the number of green and orange replies respectively and n_{total} is the total number of attempted replies. Clicking outside the modal or the button at the bottom of the dialogue takes the user back to the home page.

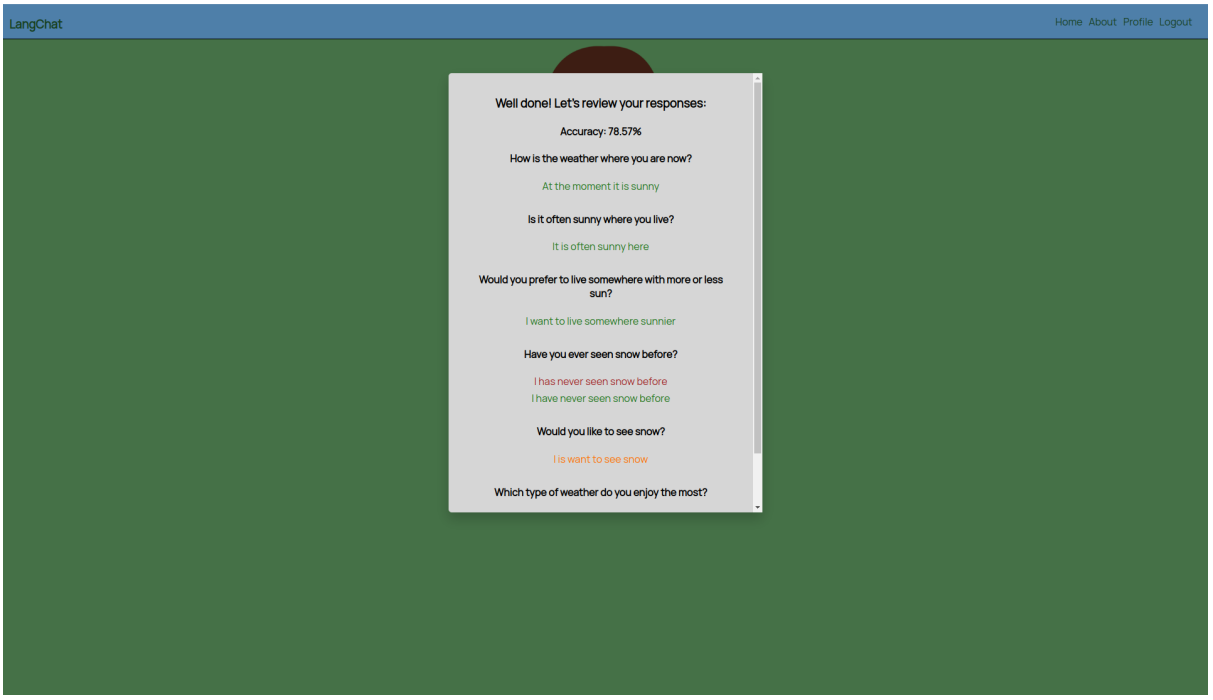

Figure 21: LangChat dialogue review modal

6.3.3. Profile Container

The Profile container defaults to rendering the ProfilePage component. This component shows the user's username, total number of dialogues completed, and a list of all the key vocabulary words encountered since registering. This is also the page in which the user is able to change their email or password, by clicking their respective links.

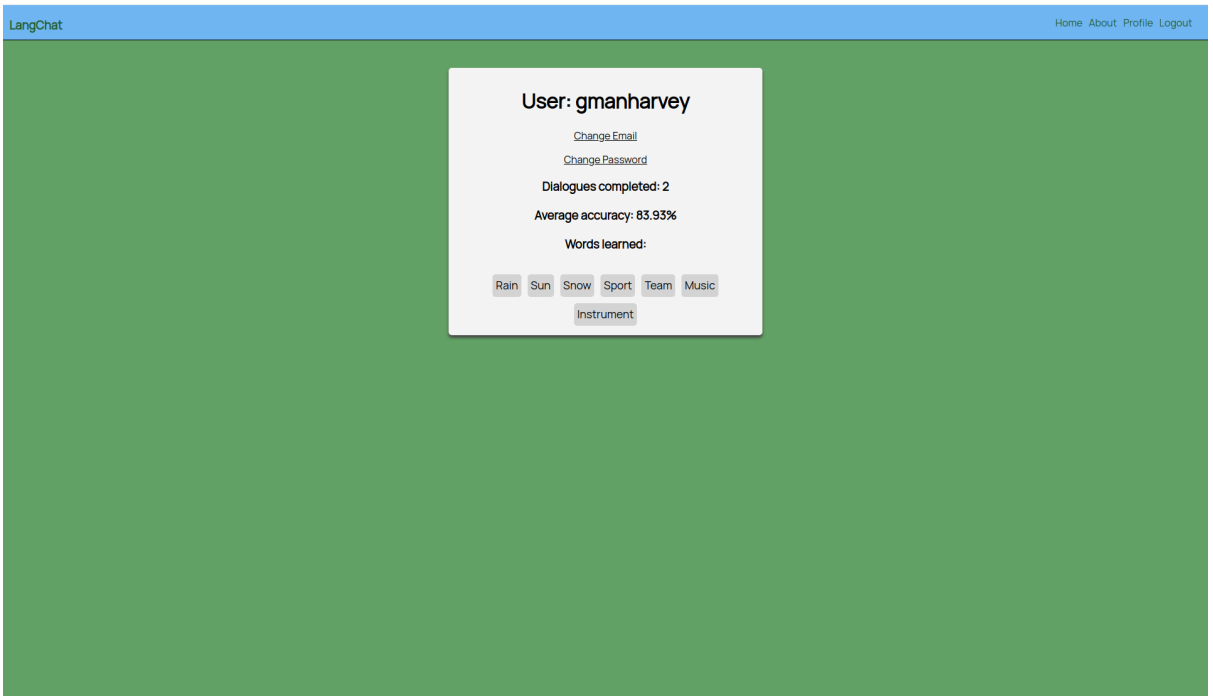

Figure 22: LangChat profile page

6.4. Deployment

In order to distribute the application to users, it was necessary to deploy it over the internet. Firstly a remote Linux server was acquired which would be used to host the application. Accessing the server over a secure shell, both the client and api source code were cloned into a local folder on the server, having been previously uploaded to a private GitHub repository.

6.4.1. Deploying the Client

Before a React application can be deployed on a production server, it must first undergo a build step. This is available as a script defined by the creators of React, which bundles and optimises all the source code files, allowing the application to be efficiently served to clients. Next NGINX was installed on the server machine. NGINX is a web server that accepts requests and forwards them to the application. NGINX can also be configured to behave as a reverse-proxy, which redirects any unrecognised requests to the API service.

6.4.2. Deploying the API

To serve the API, Gunicorn was used, a Python HTTP web server for UNIX machines. Gunicorn was configured to run the API application on the internal network, as all requests from the outside world pass through NGINX first, and are then redirected internally to the API. In order to start the application, a Linux process monitoring system was used to keep the application running continuously.

6.4.3. Obtaining an SSL Certificate

The final hurdle was to obtain an SSL certificate for the application. This was essential because in order to access the user's microphone, the website needed to be trusted by the browser, which would only be achievable if it was served over HTTPS. It is simple to configure the Flask application to generate a self-signed SSL certificate to satisfy this demand. Although this will allow the application to be served over HTTPS, the browser will still show the user a warning message saying that the certificate is not trusted. In order to avoid this, the SSL certificate needed to be obtained from a third party.

A domain name was acquired, *langchat.co.uk*, and a service named certbot was used to acquire the certificate. Certbot verifies that the developer is in control of their server and domain name, then grants them a certificate for a certain period of time. After this step was taken, the SSL certificate was granted allowing the website to be trusted by the browser.

7. Evaluation

7.1. Testing

In order to test the application, both functional unit testing and black box testing was carried out. Many unit tests were conducted on both the server and client code, details of which can be found in the appendix. After these were all completed, integration testing was carried out to ensure the functionality of the system as a whole. Greater detail about each of these steps is given below.

7.1.2. Testing the API

This was achieved using unittest, a Python library, to run tests that made requests to the endpoints exposed by the API. An example of this can be seen below.

```
def test_get_first_question(self):
    response = self.client.get('/api/get-first-question/weather/1')

    data = response.get_json()
    self.assertTrue(data['first_question_message'] == "How is the weather where you are now?")
    self.assertTrue(data['hint'] == "At the moment, it is ___")
```

The test client makes a request to the get-first-question route, then the assertion methods verify that the response received from the API is as expected.

Using this method, every API route was tested, including each possible conditional branch. Ultimately, 100% testing coverage was achieved on the back-end code, meaning that every line of code passed through at least one of the unit tests.

7.1.2. Testing the client

Testing the front-end of the application required the use of two JavaScript libraries; Jest and Enzyme. Jest is a multi-purpose test library that facilitates unit testing in JavaScript, while Enzyme is a testing utility for React that enables components to be tested without rendering their children. This feature of Enzyme is useful because it allows the developer to test components in isolation, rather than having the effects of children components influencing the test results. This is achieved using the shallow function provided by the library.

One difficulty in testing React components is ensuring that all components receive the correct props. Due to the container-component project structure detailed earlier in the project, only the containers receive props from the Redux store, which is not available during testing. All

components can be passed props they would have received from the container when they are rendered in the test. An example is given below.

```
describe('Dialogue component', () => {
    it('renders', () => {
        const wrapper = shallow(<Dialogue />);

        expect(wrapper.find(Button)).toHaveLength(4);
    });
    it('fetches first question', () => {
        const fetchMock = jest.spyOn(global, 'fetch').mockImplementation(setupFetchStub({o: 'k'}));

        const props = {
            topic: 'weather',
            level: 1
        }

        shallow(<Dialogue {...props}/>);


        expect(fetchMock).toHaveBeenCalledWith(
            '/api/get-first-question/weather/1',
            expect.anything()
        );

    });
```

In the first test, props are not needed to verify that the component has been rendered. In the second test however, the topic and level props are required for the API call that is made when the component mounts. These can be manually passed to the component when it is rendered by the shallow function.

Another feature to note is the spyOn function supplied by Jest. This allows the developer to define a mock implementation of a function which overrides its normal behaviour. The example here is the global fetch function which is used to make API calls. As it is not desirable to invoke the API in this test, the fetch function has been overridden, but any calls made to this function are recorded along with arguments passed to the function. This allows the developer to check that the function has been called the correct number of times and with the correct arguments.

Using these strategies, the components containing the critical application functionality were tested, ensuring that the client side of the application was behaving as expected.

7.1.3. Integration Testing

The final step was to test the application as a whole. To do this Selenium was used, a tool which automates the browser as if it is being controlled by a user. This method tests that the front and back ends of the application work in tandem. One example of this functionality is the test_full_dialogue method in the selenium/test.py file. In this test, the client logs into the application, selects a dialogue, shows a hint for each question and submits the hint until the dialogue is finished. This was repeated for every level in each topic and allowed us to verify that each hint is accepted by the chatbot as an answer, one of the non-functional requirements.

7.2. User trial

As a further measure to test the effectiveness of the application, a small user survey was conducted. A total of eight people were encouraged to use the application for 5-10 minutes, then give their feedback on the experience, five of whom filled out a feedback form.

- Every participant stated that they had never used any software similar to LangChat for the purpose of language learning before.
- They gave an average score of 4.2/5 to how clear and easy to use the application was
- For their favourite feature of the application:
    - Three users said their favourite feature was the key vocab section
    - One said they liked the grammar check feature
    - One liked the colour coded dialogue transcript
- For a feature they thought would improve the application:
    - One user said they would have liked the option to email the transcript
    - One said that they would have liked the microphone to be disabled while the chatbot is speaking
    - One said that they would have liked the chatbot voice to match the picture
- No users reported that any of their responses were misinterpreted
- For performance issues or bugs:
    - Three of the participants said some lines in the transcript went off the screen in mobile view
    - One said the chatbot didn't understand a lot their responses
    - One said they had trouble leaving the Review screen
- For general thoughts on how the application should be improved:
    - One person said the chatbot image should match the voice,
    - One said that they would have liked the chatbot to talk more about herself
    - One said they would like to see the chatbot picture animated while talking
- They gave an average score of 4/5 to how useful they thought the application would be for a beginner English learner

After this feedback the following changes were made:

- The microphone button is now disabled while the chatbot is talking. This prevents the application from recording the chatbot's own speech if the user presses the button prematurely.
- A button which emails the dialogue transcript to the user's email address was added. This feature was listed as a 'could have' functional requirement. After seeing that a user had requested it, time was made to implement this.
- The bug causing some responses to spill off the screen in mobile view was fixed.

- The user can now return to the home screen by clicking outside the review modal.
- AIML files were updated with new response patterns.

As for the comments regarding the chatbot avatar, this occurs because the chatbot's voice depends on the options that the browser has available. On mobile devices this is determined by the text to speech functionality on the device itself. On most platforms the default voice is female, so the current avatar is appropriate. It would be possible to check the gender of the voice available to the application and select an avatar accordingly, but there was insufficient time to implement this feature. Similarly with the suggestion that the avatar could be animated while talking, this would certainly be a desirable feature to have, but was not feasible within the time constraints of the project.

8. Discussion and conclusions

The project has succeeded in delivering a browser based chatbot which can be used by English language learners to practise their speaking and listening skills. Methodical unit testing was implemented to ensure the functionality of core components of the application, while integration testing was undertaken to guarantee that these components work reliably in unison.

Furthermore, the results from user testing were encouraging as the application was deemed to be both clear and easy to use and was thought to be beneficial to English language students. However, one limitation of this user study is that it did not include any English language learners themselves. This would have made the user survey more insightful, however it was difficult to find suitable participants given the time constraints.

It is crucial for the application to be useful that the chatbot's ability to understand a range of responses is adequate, without allowing answers with incorrect English to slip through the net. Achieving this aim is and will continue to be an ongoing process; as the application continues to be used, AIML can be updated accordingly if new correct patterns are found in the log file.

One feature that could not be implemented in this project was the functionality to translate the About page. Given that the primary use case for the application is a beginner English student, it seems unlikely that they would be able to understand a large block of English text describing how the application works. It was hoped that it would be possible to use Google Translate's "Translate this page" feature, but support for this has been discontinued. Instead the Google Chrome browser automatically offers to translate blocks of text into the user's native language, so it is hoped that this will circumvent the issue.

Similarly, another unexpected issue with LangChat is that Google's SpeechSynthesis and Speech Recognition APIs are not as widely available as was initially expected. For example, synthesis is not available on some Android devices and recognition is not available on iOS devices. Given that the focus of the application is on speaking and listening this is unfortunate, however measures have been put in place to ensure the dialogues can still be completed under these circumstances.

As for a comparison to Mondly's chatbot feature, LangChat has some similarities and some differences. Both chatbots are closed-domain, such that they are not capable of general conversation and are instead programmed to follow a narrow path of dialogue. Mondly's key word recognition feature is a benefit that aids user affirmation with visual feedback, however LangChat has several other advantageous features.

Firstly, the key vocabulary pre-training page is a unique feature that aids users' confidence and learning. This was highlighted as the most popular feature in the user survey. Furthermore, the grammar feedback feature is also beneficial for the user as it gives them an opportunity to learn from their mistakes. It is difficult to say which chatbot is more flexible in accepting user input

without testing Mondly's chatbot with an extensive range of input. However, given the continuous process of incorporating new response patterns, LangChat's flexibility can only grow with time.

In summary, while LangChat does not implement any ground-breaking technology, it does bring several novel features to a language learning chatbot, and is therefore considered to be a worthwhile tool for a beginner language learner.

8.1. Further work

The highest priority work would be directed towards finding speech recognition and synthesis services that are available across all platforms. Given the public's appetite for on the go learning, it is important that the application does not suffer a drop in usability on mobile devices. This work could be furthered by offering the application natively on the user's device, preventing them from having to use the application through the browser.

There are several features that would be desirable additions to the application. Firstly, aligning with user feedback, it would be beneficial to give the chatbot more of a personality, such that it gives some answers to its own questions. The user could then be tested on the chatbot's answers at the end of the dialogue, adding an increased focus on listening comprehension. Secondly, the chatbot could be made customisable, such that its voice and avatar can be chosen by the user in the profile page. This would enhance user engagement and avoid the issue that arose in the user survey where the male chatbot voice did not match the female avatar.

9. Bibliography

Adamopoulou, E (2020) Chatbots: History, technology, and applications *Machine Learning with Applications*

Alhmadi, N.S. (2014). English speaking learning barriers in Saudi Arabia: A case study of Tibah University *Arab World English Journal*

Anusuya, M. A. (2009) Speech Recognition by Machine: A Review *IJCSIS*

Atwell, E.S. (1999) The Language Machine *The British Council*

Bhosale, S.T. (2015) SQLite: Light Database System *IJCSMC*

Codd, E.F. (1974) Recent Investigation in Relational Database Systems *North Holland Publishing*

Cohen, Y (1989) Fear, Dependence and Loss of Self-Esteem: Affective Barriers in Second Language Learning Among Adults *RELC Journal Vol. 20*

Coniam, D (2008) An Evaluation of Chatbots as Software Aids to Learning English as a Second Language *Eurocall*

Crockford, D (2008) JavaScript: The Good Parts O'*Reilly Publishing*

Dale, R (2016) The return of the chatbots *Industry Watch*

Guzeldere G (1995) Dialogues with Colorful Personalities of Early AI *Stanford Humanities Review*

Shawar, B.A. (2007) Chatbots: Are they Really Useful *Academia*

Sianaki, O.A. (2019) A Survey on Conversational Agents/Chatbots Classification and Design Techniques *IOEMLA-2018*

Dokukina, I (2020) The rise of chatbots – new personal assistants in foreign language Learning *Procedia*

Dierbach, C (2014) Python as a first programming language *Journal of Computing Sciences in Colleges*

Fielding, R (2000) Architectural Styles and the Design of Network-based Software Architectures *University of California, Irving*

Fryer, L (2006) EMERGING TECHNOLOGIES Bots as Language Learning Tools *Language Learning & Technology*

Gackenheimer, C (2015) Introduction to React *Apress Publishing*

Grinberg, M (2018) Flask Web Development O'*Reilly Publishing*

Heil, C. R. (2016) A review of mobile language learning applications: trends, challenges and opportunities *EUROCALL*

Mayer, R.E. (2017) Using multimedia for e-learning *Journal of Computer Assisted Learning*

Naber, D (2003) A Rule-Based Style and Grammar Checker *Technische Fakultät, Universität Bielefeld*

Schussel, G (1995) Client/server past, present, and future *Digital Consulting Institute*

Shum, HY (2018) From Eliza to XiaoIce: Challenges and Opportunities with Social Chatbots *Arxiv*

SQLite https://www.sqlite.org/whentouse.html *accessed on 11/09/2021*

Taylor, P (2009) Text-to-Speech Synthesis *Cambridge University Press*

Taneja, S (2014) Python as a Tool for Web Server Application Development *JIMS*

10. Appendices

Appendix A - Source Code

The source code for the application can be found on GitLab at:

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2020/gxh096/-/tree/main.

In order to run LangChat on a local UNIX based machine, use a terminal to clone the application code into a local repository with:

"git clone --branch main https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2020/gxh096.git"

and take the following steps.

Server

1. Install Python3 and pip package management system
2. Using the command line, navigate to the the chatbot-api directory and create a virtual environment with "python3 -m venv env"
3. Activate your virtual environment using "source env/bin/activate"
4. Run “pip install -r requirements.txt”
5. In order to enable email functionality (for forgot password and email transcript features) you will have to supply an email account in a .env file with the variables MAIL_USERNAME and MAIL_PASSWORD. This is optional, otherwise these features can be viewed in the online version at https://langchat.co.uk
6. Enter “flask run” to start the server

Client

1. Install Node Package Manager (npm)
2. Using the command line, navigate to the the chatbot-react directory and run “npm install”
3. Enter “npm start” to start the client, which should open automatically in the browser. If not, enter localhost:3000 in the URL bar.

Appendix B - System Requirements

Functional Requirements

1. Authentication
    1.1. The application **must** require the user to register.
        1.1.1. The application **must** require the user to provide:
            - Email address
            - Username
            - Password
        1.1.2. The application **must** require the user to re-enter their password for verification.
        1.1.3. The application **must** inform the user if their entered email address is already registered with an account.
        1.1.4. The application **must** inform the user if their entered username is already registered with an account.
        1.1.5. The application **must** inform the user if their passwords do not match.

    1.2. The application **must** allow the user to login.
        1.2.1. The application **must** require the user's email address and password to login.
        1.2.2. The application **must** inform the user if the inputted email address has not been registered.
        1.2.3. The application **must** inform the user if their password is invalid.
        1.2.4. The application **should** allow the user to receive a temporary password by email if they forget their password.
        1.2.5. The user **should** be automatically logged if they open the application within a certain time frame from their last login

    1.3. The user **must** be able to logout.

2. Topics
    2.1. The user **must** be presented with at least three dialogue topics to choose from.
    2.2. Each topic **should** contain at least three levels of difficulty.
    2.3. Upon account creation, the user's level in each topic **must** be set to the lowest level.
    2.4. The user **must** only be able to access dialogues lower than or equal to their level.

3. Vocabulary
    3.1. The user **must** be shown key vocabulary words before starting each dialogue.
    3.2. The application **should** read each word aloud when clicked.
    3.3. The application **should** show the user the pronunciation of each word when

clicked.

3.4. The application **should** show the user an image which shows the definition of each word when clicked.

4. Dialogue
   - 4.1. The application **must** automatically fetch the first question from the API when the dialogue is started.
   - 4.2. The application **should** read each question aloud when it is received from the API.
   - 4.3. The application **must** check the correctness of each user's reply.
   - 4.4. The application **must** prioritise speaking and listening as the primary means of interacting with the chatbot.
   - 4.5. The user **must** have the option to view the question text.
   - 4.6. The application **must** have the option to type their reply.
   - 4.7. The user **must** have the option to replay the question audio.
   - 4.8. The application **must** provide a model answer for each question.
   - 4.9. The application **should** play an affirmative sound if a user's reply is accepted.
   - 4.10. The application **should** repeat the question audio if the user's reply is not accepted.
   - 4.11. The application **must** display the speech recognition result if the user's reply is not accepted.
   - 4.12. The application **must** log each user response that is not accepted to a file.
   - 4.13. The application **must** check each user reply for grammatical errors.
   - 4.14. The application **must** prompt the user with the corrected version of the sentence if their reply contained an error.
   - 4.15. After completion, the user's level for the corresponding topic **must** be increased in the database.
   - 4.16. The user **should** be prompted to reply with full sentences at the start of their first dialogue.

5. Review
   - 5.1. The application **must** show the user a transcript of the conversation after the dialogue is finished.
   - 5.2. The transcript should be colour coded, where:
     - Green indicated an accepted reply.
     - Orange indicates an accepted reply with a grammatical error.
     - Red indicates a rejected reply.
   - 5.3. The transcript dialogue **must** include a score evaluating the user's accuracy over the dialogue.
   - 5.4. The dialogue score **must** be saved to the database.
   - 5.5. The application **could** give the option to email the transcript to the user's email address.

6. About
    6.1. The user **should** be provided a page explaining how to use the application.
    6.2. The user **should** have an option to translate the about page into a language of their choice.

7. Profile
    7.1. The application **must** provide a profile page with an account summary.
    7.2. The profile page **must** show:
        - The user's username
        - The average accuracy over all their completed dialogues.
        - A list of all key vocabulary encountered over completed dialogues.
    7.3. The profile page **should** give users the option to change their password.
    7.4. The profile page **should** give users the option to change their email address

3.1.2. Non-functional Requirements

1. Portability
    1.1. The application **must** be available in the Google Chrome browser.
    1.2. The application **should** be available in the Firefox browser.
    1.3. The application **should** be available in the Microsoft Edge browser.
    1.4. The application **must** be usable on mobile devices.
        1.4.1. The user interface **must** support a minimum screen width of 350 pixels.
        1.4.2. The user interface **must** support a minimum screen height of 600 pixels.

2. Security
    2.1. User email addresses **must** be unique.
    2.2. Password fields **must** be hidden during login and registration.
    2.3. Passwords **must** be encrypted when stored in the database.

3. Implementation
    3.1. The application server **must** be written in Python.
    3.2. The application **must** use Flask Python web development framework.
    3.3. The front-end of the application **must** be written in JavaScript
    3.4. The front-end of the application **must** use the ReactJS framework.

4. Dialogue
    4.1. Dialogues **must** be able to be completed if the speech synthesis is not available on the device.
    4.2. Dialogues **must** be able to be completed if the speech recognition is not available on the device.
    4.3. Every model answer provided **must** be accepted if submitted by the user.

Appendix C - Test Cases

API Unit Tests

Test names have had underscores removed to enhance readability

Test name	**Description**	**Requirement**	**Result**
test app exists	Checks Flask app object is created	N/A	Passed
test app is testing	Checks app is created with testing configuration	N/A	Passed
test get first question	Checks first question is received from API	4.1.	Passed
test get response no grammar	Checks a valid reply is accepted and next question is sent without grammar messages	4.3.	Passed
test get response with grammar	Checks a reply with a grammar error receives a response with a grammar message	4.13.	Passed
test get bad response user message	Checks an invalid reply receives no response from the chatbot	4.3.	Passed
test get vocab words	Checks correct vocabulary words are received for topic and level	3.1.	Passed
test full dialogue	Runs through a full dialogue with valid user replies to check test finishes as expected	5.1.	Passed
test register and login	Checks registering and logging in receives correct responses	1.1.1., 1.1.2., 1.2,1,	Passed
test login no account error	Checks logging in with an unregistered email address receives expected error	1.1.2.	Passed
test re-register email error	Checks registering an email address previously registered receives expected error	1.1.3., NF2.1.	Passed
test re-register username error	Checks registering a username previously registered receives expected error	1.1.4.	Passed
test login wrong password	Checks logging in with incorrect password receives correct error	1.2.3..	Passed
test user	Checks user account details received from	7.2.	Passed

details	API are as expected		
test finish dialogue	Checks finishing a dialogue updates corresponding user level correctly	4.15.	Passed
test send accuracy	Checks a score is added to the database after finishing a dialogue	5.4.	Passed
test missing token	Checks expected error is raised when trying access a route without a token that requires authentication	N/A	Passed
test invalid token	Checks expected error is raised when trying access a route with an invalid token that requires authentication	N/A	Passed
test change password	Checks change password route correctly changes password	7.3.	Passed
test forgot password	Checks forgot password route returns correct response	1.2.4.	Passed
test change email	Checks change email route correctly changes email	7.4.	Passed
test no users	Checks users table is initially empty	N/A	Passed
test password setter	Checks setting password attribute on user object sets password_hash attribute	NF2.3.	Passed
test no password getter	Checks password field cannot be read on user object	N/A	Passed
test verify password true	Checks verify_password function behaves correctly	N/A	Passed

Client Unit Tests

Test name	**Description**	**Requirement**	**Result**
About container - renders	Checks About container renders as expected	6.1.	Passed
Auth container - renders	Checks Auth container initially renders with Login component	1.2.	Passed
Auth container - switch to register	Checks content is switched to Register component when link clicked	1.1.	Passed
Auth container - switch to forgot password	Checks content is switched to ForgotPassword component when link clicked	1.2.4.	Passed
Main container -	Checks Main container renders correctly	N/A	Passed

renders with home	with Home component		
Main container - renders with vocab	Checks Main container renders correctly with Vocab component	N/A	Passed
Main container - renders with dialogue	Checks Main container renders correctly with Dialogue component	N/A	Passed
Profile container - initially renders with profile page	Checks Profile container initially renders with ProfilePage component	7.1.	Passed
Profile container - switch to change email	Checks content is switched to ChangeEmail component when link clicked	7.4.	Passed
Profile container - switch to change password	Checks content is switched to ChangePassword component when link clicked	7.3.	Passed
ForgotPassword - renders	Checks ForgotPassword component renders correctly	1.2.4.	Passed
ForgotPassword component - submits with email address	Checks submit button sends email to forgot-password route	1.2.4.	Passed
Login - renders	Checks Login component renders correctly	1.2.	Passed
Login component - submits with correct data	Checks submit button sends email and password in fetch body	1.2.1.	Passed
Register component - renders	Checks Register component renders correctly	1.1.	Passed
Register component - submits with correct data	Checks submit button sends email, username and password in fetch body	1.1.1., 1.1.2.	Passed
Change Email component - renders	Checks ChangeEmail component renders correctly	7.4.	Passed
ChangeEmail component - submits with email address and password	Checks submit button sends password and email to change-email route	7.4.	Passed
ChangePassword component - renders	Checks ChangePassword component renders correctly	7.3.	Passed
ChangePassword component - submits with current and new password	Checks current and new password is sent to change-password route	7.3.	Passed

Dialogue component - renders	Checks Dialogue component renders correctly	N/A	Passed
Dialogue component - fetches first question	Checks correct get-first-question route is invoked for topic and level	4.1.	Passed
Dialogue component - submits user speech	Checks last chatbot response and user reply are sent to get-response route from input	4.3.	Passed
Home component - renders	Checks Home component renders correctly	2.1.	Passed
Home component - fetches user details on mount	Checks get-user-details route is invoked when component mounts	2.3.	Passed
Home component - loads dialogue options after fetching details	Checks topic dropdowns are rendered after fetching from database	2.1.	Passed
ProfilePage - renders	Checks ProfilePage component renders correctly	7.1.	Passed
ProfilePage component - fetches user details on mount	Checks get-user-details route is invoked when component mounts	7.2.	Passed
ProfilePage component - accuracy score loaded	Checks correct accuracy score is rendered from database	7.2.	Passed
Review component - renders	Checks Review component renders correctly	5.1.	Passed
Review component - renders with all responses and correct classes	Checks all responses have expected CSS class according to correctness	5.2.	Passed
Vocab component - renders	Checks Vocab component renders correctly	3.1.	Passed
Vocab component - fetches vocab words on mount	Checks get-vocab-words route is invoked when component mounts	3.1.	Passed
Vocab component - renders vocab words after fetch	Checks fetched vocab words are rendered	3.1.	Passed

Selenium Integration Tests

Test names have had underscores removed to enhance readability

Test name	Description	Requirement	Result
test login	Checks that logging into the application brings client to Main container	1.2.	Passed
test navigate to about	Checks that clicking About link in toolbar brings client to About container	6.1.	Passed
test navigate to profile	Checks that clicking Profile link in toolbar brings client to Profile container	7.1.	Passed
test navigate back to home	Checks that clicking Home link in toolbar brings client back to Main container from Profile container	2.1.	Passed
test navigate to vocab	Checks that clicking a dropdown topic button takes client to vocab page	3.1.	Passed
test full sentence prompt shows on first dialogue	Checks that the prompt to use full sentences is displayed if it is the user's first dialogue	4.16.	Passed
test dialogue buttons	Checks the dialogue controls: show text, repeat, show hint and type response all behave as expected	4.2., 4.5., 4.6., 4.4., NF1.4., NF1.5.	Passed
test full dialogue	Runs through a full dialogue by requesting a hint, then inputting hint into input bar and submitting - until dialogue finishes	5.1., NF4.1.	Passed