



Cardiff University School of Computer
Science & Informatics

Game with a purpose

CM3203 - One Semester Individual Project Final Report

Author:

William Cooter

Supervisor:

Irena Spasic

Moderator:

Angelika Kimmig

10th of May, 2019

Abstract:

The game with a purpose is a tool to collect metadata that a computer would find difficult to generate, but that human would be well suited to. The metadata this project focuses on are the word associations synonyms, antonyms and hypernyms. This report will outline the uses and difficulties of this metadata, and present an approach to creating a game that collects this data from players.

Acknowledgements:

I would like to thank my supervisor Irena Spasic for her support throughout this project. I greatly appreciate the guidance you have provided to me over the past few months.

I would also like to thank any friends and family who tested the project by signing up and playing the game. Without you there would have been no credible data, and development would have been considerably more difficult.

Table of contents

1.	Introduction	1
2.	Background	3
3.	Specification and design	6
3.1.	Login system	6
3.1.1.	User perspective	6
3.1.2.	System behaviour	7
3.1.3.	Algorithms and architecture	8
3.1.4.	Constraints	9
3.2.	Matchmaking service	9
3.2.1.	User perspective	9
3.2.2.	System behaviour	10
3.2.3.	Constraints	12
3.3.	Multiplayer game	12
3.3.1.	User perspective	12
3.3.2.	System behaviour	14
3.4.	Result processing service	17
3.4.1.	User perspective	17
3.4.2.	System behaviour	18
3.4.3.	Constraints	18
3.5.	Single-player game	18
3.5.1.	User perspective	18
3.5.2.	System behaviour	19
3.6.	Website	19
3.6.1.	User perspective	19
3.6.2.	System behaviour	19
3.6.3.	Constraints	20
4.	Implementation	21
4.1.	Login system	21
4.1.1.	Database queries	21
4.1.2.	POST requests	21
4.1.3.	Emails	21
4.1.4.	Registering input validation	22
4.1.5.	Redirects	22
4.1.6.	Timeline of work	23
4.1.7.	Element-ui issue	23
4.2.	Matchmaking service	24
4.2.1.	Matchmaking.js	24
4.2.2.	checkMatches()	24
4.2.3.	getQueryParams()	24
4.3.	Multiplayer game	25
4.3.1.	Get game information and state	25
4.3.2.	Timer	26
4.3.3.	Submitting answers	27
4.3.4.	Skip word	29
4.3.5.	Quit game	30
4.3.6.	Results screen	30

4.3.7.	Issue with definitions	31
4.4.	Result processing service	31
4.4.1.	checkWords()	31
4.4.2.	resetValuesForTesting()	33
4.5.	Single-player game	33
4.6.	Website	33
4.6.1.	TOML file	33
4.6.2.	Issue	33
5.	Results and evaluation	34
5.1.	System working as (or not as) intended	34
5.1.1.	Generate metadata	34
5.1.2.	Fun and appealing	34
5.1.3.	Easy to use and available to all	35
5.2.	Comprehensible results	35
5.2.1.	Matches	35
5.2.2.	Game modes	36
5.3.	Confidence in results	36
5.4.	Testing	37
5.4.1.	Code testing	37
5.4.2.	Application testing	37
5.5.	Evaluate methodology and programming language	38
5.5.1.	Database	38
5.5.2.	Node	38
5.5.3.	Vue	38
6.	Future work	39
6.1.	Unrealised ideas	39
6.2.	Starting point for continuation of work	41
7.	Conclusion	43
8.	Reflection on learning	44

Table of figures

Due to the large volume of figures, all screenshots are included at the end of the main body so as to not break up the flow of the report. Diagrams in section 5 are included in their appropriate locations in the report.

1.	Website screenshots	45
1.1.	Login system screenshots	45
1.1.1.	Sign-up page without input	45
1.1.2.	Sign-up page with input	45
1.1.3.	Home page	46
1.1.4.	Forgotten password page	46
1.1.5.	Reset password page	46
1.1.6.	Account settings page	47
1.1.7.	Delete account page	47
1.1.8.	Website menu	48
1.1.9.	Sign-up confirmation email	48
1.1.10.	Confirmation email alert	48
1.1.11.	Login failure message	49
1.2.	Multiplayer game screenshots	49
1.2.1.	Multiplayer game menu	49
1.2.2.	Waiting mode	49
1.2.3.	Game countdown	50
1.2.4.	Game page	50
1.2.5.	Answers before submission	50
1.2.6.	Answers after submission	51
1.2.7.	Match notification	51
1.2.8.	Disabled input and skip button	51
1.2.9.	Skip alert	51
1.2.10.	Multiplayer game information box	52
1.2.11.	Progress bar green	52
1.2.12.	Progress bar amber	52
1.2.13.	Progress bar red	52
1.2.14.	Other player quit alert	53
1.2.15.	Multiplayer game results table	53
1.3.	Singleplayer game screenshots	53
1.3.1.	Single-player game information box	53
1.3.2.	Single-player game menu	54
2.	Code screenshots	55
2.1.	Login system code screenshots	55
2.1.1.	The database credentials being imported from the <code>.env</code> file into the file <code>'db.js'</code>	55
2.1.2.	A JavaScript promise in the file <code>'db.js'</code>	55
2.1.3.	The database query function being imported into the file <code>'server.js'</code>	55
2.1.4.	The axios API request function being imported into a Vue component	55

2.1.5.	The URL of a POST request being determined in the file 'auth.js'	55
2.1.6.	The email credentials being imported from the .env file into the file mail.js'	56
2.1.7.	The URL of a the API being determined in the file 'mail.js'	56
2.1.8.	An email template being read from the file 'mail.js'	56
2.1.9.	The password strength function in the file 'SignUp.vue'	57
2.1.10.	The confirm password function in the file 'SignUp.vue'	57
2.1.11.	The is a string function in the file 'SignUp.vue'	57
2.1.12.	The validation message for no input into a field	58
2.1.13.	The validation message for bad input into a field	58
2.1.14.	The validation message for correct input into a field	58
2.1.15.	The redirect function found in most Vue files	58
2.1.16.	An example of the redirect function in the HTML template	59
2.1.17.	The Vue router	59
2.1.18.	The redirect in new tab function in the file 'SignUp.vue'	60
2.1.19.	The redirect in new tab function in the HTML template in the file 'SignUp.vue'	60
2.1.20.	The hashing of a password in the file 'server.js'	60
2.1.21.	The creation of a sign up token in the file 'server.js'	60
2.1.22.	The authentication of a password in the file 'server.js'	61
2.1.23.	The creation of a JSON web token in the file 'server.js'	61
2.2.	Matchmaking service code screenshots	61
2.2.1.	The main function with variable time intervals in the file 'matchmaking.js'	61
2.2.2.	The file 'matchmaking.js' checking that a user's last heartbeat was no more than 5 seconds ago	61
2.2.3.	The grouping of users by their chosen game mode in the file 'matchmaking.js'	62
2.2.4.	The creation of a string to be used in an SQL statement in the file 'matchmaking.js'	62
2.2.5.	The file 'matchmaking.js' looping through each game mode and then every other user	62
2.3.	Multiplayer game code screenshots	62
2.3.1.	The addition of 1 hour to the data retrieved from the database in the file 'server.js'	62
2.3.2.	The current word index recovered from a partially completed game in the file 'server.js'	63
2.3.3.	The matched and passed counts recovered from a partially completed game in the file 'server.js'	63
2.3.4.	The already submitted answers recovered from a partially completed game in the file 'server.js'	63
2.3.5.	The count of the other player's answers recovered from a partially completed game in the file 'server.js'	63
2.3.6.	Assigning of the game information in the file 'Game.vue'	64
2.3.7.	The HTML timer component in the file 'Game.vue', able to call the methods startGame and delayGame	64
2.3.8.	The start game function in the file 'Game.vue'	64

2.3.9.	The delay game function in the file 'Game.vue'	65
2.3.10.	The watch section of the file 'TimerMultiplayer.vue'	65
2.3.11.	The increment time function getting the actual time in a UTC format in the file 'TimerMultiplayer.vue'	65
2.3.12.	The getDifference function in the file 'TimerMultiplayer.vue' ...	65
2.3.13.	Emission of 'start_game' in the file 'TimerMultiplayer.vue'	66
2.3.14.	The emission of 'delay_game' with the current time in the file 'TimerMultiplayer.vue'	66
2.3.15.	The file 'TimerMultiplayer.vue' redirecting the client to the game results page	66
2.3.16.	The inputted answer being broken into an array of words in the file 'Game.vue'	66
2.3.17.	The filtering of inputted words in the file 'Game.vue'	67
2.3.18.	The connection to either the 'queue' room or the room by the name of the token in th file 'server.js'	67
2.3.19.	The player number strings determined in the file 'server.js'	67
2.3.20.	The parsing of stringified arrays in the file 'server.js'	68
2.3.21.	The intersections between both player's answer arrays in the file 'server.js'	68
2.3.22.	The emission to the answer submitted socket if there was a match in the file 'server.js'	68
2.3.23.	The emission to the answer submitted socket if there was not a match in the file 'server.js'	68
2.3.24.	File 'Game.vue' listening to the answer submitted socket	68
2.3.25.	The element-ui alert telling the players that they have matched on a word in the file 'Game.vue'	69
2.3.26.	The file 'Game.vue' redirecting to the results page if they previous match was on the final word	69
2.3.27.	The file 'Game.vue' incrementing the matched count and executing the next word function if the previous match was not the last word	69
2.3.28.	The file 'Game.vue' emitting through the skip word socket	69
2.3.29.	The element-ui alert received by a player if the other player skipped a word in the file 'Game.vue'	70
2.3.30.	The emission to the other player skipped socket in the file 'server.js'	70
2.3.31.	The code executed by the file 'Game.vue' upon receiving an emission from the other player confirmed skip socket	70
2.3.32.	The file 'Game.vue' emitting to the quit game socket and redirecting to the results page	71
2.3.33.	The emission to the other player quit socket from the file 'server.js'	71
2.3.34.	The code executed by the file 'Game.vue' upon receiving an emission to the other player quit socket.	71
2.3.35.	The HTML code for the table in the file 'GameResults.vue' determining the icon at the start of the row	72
2.3.36.	The CSS of the colours for the background of the table in the file 'GameResults.vue'	72

2.4.	Result processing service code screenshots	73
2.4.1.	The main function of the file 'result_processing.js'	73
2.4.2.	The grouping of answers by their game mode in the file 'result_processing.js'	73
2.4.3.	The object created for each word in 'result_processing.js'	73
2.4.4.	The string created to be used in an SQL statement in the file 'result_processing.js'	73
2.4.5.	The SQL statement that recieves a string in the file 'result_processing.js'	74
2.4.6.	The conditions for a word to be made available in the single-player game mode in the file 'result_processing.js'	74
2.5.	Website code screenshots	74
2.5.1.	The contents of the file 'netlify.toml'	74
2.5.2.	The use of the mobile checking function found in all Vue components.	74
2.5.3.	The mobile check function in the file 'mobileCheck.js'	75
3.	Database screenshots	76
3.1.1.	The number of synonyms available for the single-player game mode	76
3.1.2.	The number of antonyms available for the single-player game mode	76
3.1.3.	The number of hypernyms available for the single-player game mode	77
3.1.4.	The number of multiplayer game that have been played	77
3.1.5.	The number of users who played the multiplayer game	78
3.1.6.	The four synonyms that are available in the single-player game mode	78
3.1.7.	The synonyms for the word "cold"	78
3.1.8.	The synonyms for the words "cold" and "tree"	79
3.1.9.	The number of matched antonyms	79
3.1.10.	The number of skipped antonyms	79
3.1.11.	The number of antonym games	79
3.1.12.	The number of uncompleted antonyms	80
3.1.13.	The number of matched hypernyms	80
3.1.14.	The number of skipped hypernyms	80
3.1.15.	The number of hypernym games	81
3.1.16.	The number of uncompleted hypernyms	81
3.1.17.	The number of matched synonyms	81
3.1.18.	The number of skipped synonyms	81
3.1.19.	The number of synonym games	82
3.1.20.	The number of uncompleted synonyms	82
4.	Miscellaneous screenshots	83
4.1.1.	Wordnet antonyms for the word "cold"	83
4.1.2.	The API (server) and services (matchmaking and result_processing) running on the Digitalocean server	83
4.1.3.	The address of the website	83
4.1.4.	The website's certificate	84
4.1.5.	The default colours used by element-ui	85

4.1.6.	The Vue components, notably the files 'Game.vue' and 'SinglePlayerGame.vue'	85
4.1.7.	The ESLint and Prettier files	85
4.1.8.	A copy of the .env file containing default values for the environment variables	86
5.	Diagrams	
5.1.1.	Websocket diagram for users in the matchmaking queue	11
5.1.2.	Matchmaking algorithm diagram	12
5.1.3.	Websocket diagram for answer submission	15
5.1.4.	Websocket diagram for skipping a word	16
5.1.5.	Websocket diagram for quitting the game	17

1. Introduction

The aim of the 'game with a purpose' project is to develop a tool that enables the generation of metadata that a computer would find hard to create, but that a human is well suited to. The tool is to be packaged as a game to collect the metadata of associations between words. There are three word associations that this project focuses on - synonyms (words with the same meaning, e.g. "cold" → "freezing"), antonyms (words the opposite meaning, e.g. "empty" → "full") and hypernyms (words with a more general meaning, e.g. "chair" → "furniture"). The goals of this project is to make a game that is both fun and appealing to play, while being efficient and effective at gathering data. The tool should also be easy to use accessible to all. The project consists of two games - a multiplayer game where two players agree on synonyms/antonyms/hypernyms for a set of words, and a single-player game that uses predefined associations to test a player's knowledge of the English language. The system is also to include a login system, requiring users to sign in before they can play either of the games.

The multiplayer game is designed with the intent of it being used by those who already possess a good understanding of the English language as both players may have to enter several synonyms/antonyms/hypernyms before they can agree on a match. On the other hand, the single-player game could be played by anyone with at least a very basic knowledge of English as the player would only need to enter one of the predefined word associations for each word they see. This could be useful for an educational purpose such as helping non-native English speakers test their knowledge of the language, or to help school children learn about word associations. The metadata that is generated by the tool could be used in its raw form by academics studying the English language, for word association dictionaries such as a thesaurus, or for an API that returns metadata for a specified word.

A login system for the game is the first priority of the project. The main goal of the login system is for it to be secure and safe, making sure that all user data is protected where necessary. To login, a user will have to use an email address and password as their credentials. A user must also have the ability to manage their account, allowing them to change their credentials and to delete their account if they so wish. To comply with the General Data Protection Regulation (GDPR), the system collects as little data as possible from a user, asking for only their forename, surname and email address. A user's data can also be erased by deleting their account, or by requesting that their data be removed in which case I would remove their account from the system. So that the users are aware that their data is being stored and they have the right for this data to be erased, a set of terms and conditions were created which must be agreed to upon registering an account (see Appendix D).

The single-player and multiplayer games are to be of a similar format - a player chooses to play one of three game modes (either synonyms, antonyms or hypernyms), and then has 2½ minutes to find 15 of these word associations. The words are shown one at a time, and the player will enter answers until a correct match is found. The player also has the option to skip a word if they so desire. The

key difference between the single-player game and the multiplayer game is how an answer is determined to be a match. In the multiplayer game there are two players simultaneously entering answers for the same word. Once both players have entered the same answer, then a match has been created and they are moved on to the next word. In the single-player game a player's answer has to be one of the predefined word associations to be a match. The inputs and matches of each game are to be recorded, along with whether a word was skipped or if the game was prematurely quit by a player. Once a game has finished, a result screen is shown to the player(s) containing a log of the game.

The approach taken to this project was to first analyse the brief and decide on a means of creating a multiplayer game. For two users to be able to play, they would either have to share a device, or be able to play from two separate devices. As the players would need to be entering answers simultaneously without knowing what the other player had entered, it was vital that they be able to play the multiplayer game from different devices. To connect the devices, a server and database were set up to communicate with each client device. The interface was then built as a web application for ease of access.

This project was built on the assumptions that players would be able to quickly and easily find synonyms, antonyms and hypernyms of words under a time limit. The multiplayer game also assumes that one of the associations that both players enter will be the same. However the largest assumption made by this project was that players would be honest when entering answers into the game, and not coordinate with another player to enter false answers into the multiplayer game.

To summarise, the main goals of this project have been to develop a tool that contains a login system that securely stores and transmits a user's data, a multiplayer game that collects associations between words, and a single-player game that uses predefined associations to test a player's knowledge of the English language.

2. Background

Some metadata is hard or impossible for a computer to generate. Objective topics with a lot of data allow a computer to define relationships between entities using either algorithms or artificial intelligence. In the case of word associations, a system could be fed data such as definitions and word types, however it would most likely be unreliable and each association it created would have to be confirmed by a human. This is because word associations are subjective, and hence are more suited to a human brain that is better at this type of understanding and processing. Word associations such as those which this project caters for are an example of this. This metadata can be used for multiple purposes, but particularly on the internet to create links between topics. For example, when somebody uses Google to search for a word, the search engine could also find sites using synonyms of the search term.

Likely stakeholders in this problem are those who require these word associations for their business/organisation. This could be those who study the English language at a higher level and want access to multiple associations for a single word. The data could also be useful to software developers wishing to create applications such as a thesaurus API (and hence need synonyms).

There are already several online tools that are able to give word associations upon request. These range from thesaurus websites such as Thesaurus.com¹ to word association APIs such as WordsAPI². However neither of these sites disclose how they acquire this metadata, leading to the assumption that they come from a static pre-made list. There are a few word association games on the internet such as the synonym game at Learning Games for Kids.com³, however these do not collect data in the same way as this project, and instead simply test a player's knowledge of the English language (alike to the single-player game of the project). The tool that this project aims to extend is the Princeton University lexical database 'Wordnet' - an API which contains word associations such as synonyms, antonyms and hypernyms. These associations were originally to be used in the single-player game as the criteria for a match, however upon investigation of the API it was decided that the Wordnet database was insufficient and would not be effective in supporting a single-player version of the game. For example the word "cold" only has the antonym "hot" [fig 4.1.1], meaning any other antonyms such as "warm" and "scorching" would not be accepted. Therefore the decision was made to use the relations generated by the multiplayer game to support the single-player game, rather than relying on the relations from Wordnet.

Once the decision had been made to create the game as a web application, the next step was to decide on a programming language. The obvious choices for the backend were PHP or Node.js. Though PHP is less complex, Node.js is much more up to date having been released in 2011 (Node.js, 2019) compared to 1995 (PHP, 2019). Node.js is also considerably faster and more lightweight than PHP, allowing more demanding processes to be run on a small server. Node.js is a JavaScript runtime

¹ <https://www.thesaurus.com/>

² <https://www.wordsapi.com/>

³ <https://www.learninggamesforkids.com/vocabulary-games/synonyms.html>

environment used to run JavaScript files (.js), and is commonly used for modern backend servers. To keep the project consistent, the frontend of the project was also run through a JavaScript framework. The two frameworks that had the greatest appeal were React and Vue.js. Despite having some limited previous experience with React, the decision was made to use Vue.js due to it being more up to date and compatible with the latest JavaScript version ES6.

The tools required for this project were a database to store all of the system and user's data, and the means to release the tool as a live website that anyone could access. While a non-relational database has advantages such as more efficient storage of JavaScript Object Notation (JSON) data structures (a common occurrence in web applications), a relational MySQL database was chosen as the initial schema of the database looked to be fairly consistent. This was hosted on an Amazon Web Services (AWS) server. To run the backend code such as login verification, a backend server was needed that could run a Node.js environment. There are many companies who sell space on a server, with the only notable differences between them being the price and then customer support levels. DigitalOcean provided a good balance of these two factors, and so a Ubuntu Node.js droplet was rented from them. To host the frontend of the system, a global deployment tool was needed. For this Netlify was chosen for two reasons - firstly it is free, and secondly the site can be deployed directly from a GitHub repository, automatically redeploying the website whenever a change is detected to the branch that is being deployed.

To manage version control of the project, GitHub was used to store the code repository. The branch 'dev' was used for production, while various other branches such as 'game' were used for development. These could then be merged back into the 'dev' branch. Before a merge was committed, Netlify would test the merge by building the frontend in a test environment. If the build was successful, the merge could be completed and Netlify would automatically deploy a fresh build of the frontend. The backend would have to be pulled to the server manually and any processes restarted. The GitHub repository of the project can be located at <https://github.com/WillCooter/final-year-project>.

Due to this project handling user data such as emails through the login system, it was vital that the Cardiff University Research Integrity training be completed (see Appendix A). This aided in the understanding of the responsibilities of storing user data and the need for privacy.

Finally, the packages used for this project are listed below. These can also be seen in the 'package.json' files in the 'frontend' and 'backend' directories of the code:

Languages:

- Backend written in Node v10.7.0 (Node.js, 2018)
- Frontend written in Vue v3.3.0 (Vue.js, 8th January 2019)
- MySQL v5.6.40 (MySQL, 21st January 2019)

Backend packages:

- Bcrypt v2.0.0 - to hash passwords (Bcrypt, April 2019)

- Crypto v1.0.1 - to create random tokens (Crypto, 2017)
- Express v4.16.3 - to allow GET and POST requests from the frontend (Express, November 2018)
- Jsonwebtoken v8.2.1 - to create web tokens after login (Jsonwebtoken, March 2019)
- Lodash v4.17.11 - data manipulation functions (Lodash, 12th September 2018)
- Moment v2.24.0 - functions involving time (Moment.js, January 2019)
- Mysql v2.16.0 (version of the package, not the MySQL) - to query the database
- Nodemailer 4.7.0 - to send emails from a gmail account (Nodemailer, 19th April 2019)
- Socket.io v2.2.0 - to use websockets in the API (Socket.io, 29th November 2018)

Frontend packages:

- Axios v0.18.0 - to send GET and POST requests to the backend API (Axios, August 2018)
- Element-ui v2.5.4 - frontend components such as buttons (Element-ui, 25th August 2019)
- Lodash v4.17.11 - data manipulation functions (Lodash, 12th September 2018)
- Moment v2.24.0 - functions involving time (Moment.js, January 2019)
- Password-strength-utility v1.1.6 - password strength when registering/changing password (Password-strength-utility, 2017)
- Socket.io-client v2.2.0 - allow websockets to connect to the API (Socket.io-client, December 2018)
- Vue v3.0.0 - vue client (Vue, 8th January 2019)

3. Specification and design

The development of the project can be broken down into 6 keys sections:

- A login system which must allow users to sign up with an email address, use it to log in with their chosen password, and give the user the ability to edit their credentials or to delete their account.
- A matchmaking system that can find 2 users wishing to play the multiplayer game, and start a game for them.
- A multiplayer game where 2 players have to agree on synonyms, antonyms or hypernyms for 15 words.
- A result processing system that can use the results of the multiplayer games to create answers for the single-player game.
- A single-player game where a player has to find predefined word associations for 15 words.
- Setting up the server and website.

3.1 Login system

3.1.1 User perspective

The login system was the first part of the project to be developed. This had to be secure enough to comply with GDPR, whilst also remaining easy to use so as not to discourage potential users with a long sign-up process. From a user's perspective, the first page they would see upon accessing the website is the sign-up page [fig 1.1.1 and fig 1.1.2]. This allows a user to register an account, taking only necessary details from them. These are the user's forename, surname and email address. The user is also required to enter a password and then to confirm it by typing it in again in a separate field of the form. If the strength of the password is not above a certain threshold it will be rejected. At the bottom of the page is a checkbox for the user to accept the terms and conditions of registering an account (see Appendix D), which can be accessed by clicking on the words 'terms and conditions', highlighted in blue. Once all fields have been filled with valid input (e.g. a valid email address) and the terms and conditions checkbox has been ticked, the user can register their account.

After registering an account, an email will be sent to the email address used by the user to sign up. This contains a unique link which upon being clicked takes the user back to the website and informs them that their account has been set up. The user can now log in with their email address and password, which will take them to the homepage of the website [fig 1.1.3]. If a user can not recall their password, there is a link on the login page that takes a user to the 'forgotten password' page [fig 1.1.4]. Here they can enter their email address, which will send an email to the account registered with that email address if it exists. This email will contain a unique link alike to the one sent for the account verification, except this time it takes the user to a page where they can enter a new password [fig 1.1.5]. Similar to when registering, the user will have to enter their new password twice and it will have to meet a certain strength criteria.

Once a user has logged in successfully, they are now able to manage their account. The two parts of an account a user can edit are their email address and password via the 'account settings' page [fig 1.1.6]. If a user wishes to change their email address, they must enter the new email address and their current password. A confirmation email is then sent to the new address containing a unique link, which upon being clicked confirms the change and logs the user out. The user must then log back in with their new address to continue using the account. For a user to change their password, they must enter their current password along with their new password and a repetition of their new password. This is also validated with the password strength criteria alike to when registering and when having forgotten one's password.

Should a user wish to delete their account, they must go to the 'delete account' page [fig 1.1.7], accessed by a button at the bottom of the 'account settings' page [fig 1.1.6]. Here they must re-enter their email address and password to confirm that they want to delete their account (as if they are logging in). If they are successful, the user is redirected back to the login page. If a user chooses to logout of their account (done via the menu in the top right of the page [fig 1.1.8]) then they will also be redirected to the login page, but their account will remain valid.

3.1.2 System behaviour

The system itself functions mostly through POST requests to send data to the system's API. The API then uses this data to query the database, and returns the appropriate data to instruct the client (the user's browser) on what to do next. When a user registers, the client sends a POST request to the system's API with the new credentials. This request is only sent if all fields of the form have been filled, the password and its confirmation are identical, the terms and conditions checkbox has been ticked, and the password has met a strength threshold determined by the package *password-strength-utility*. Upon receiving the credentials, the API checks that the email address is of a valid syntax, normalises the inputs (e.g. forname "wiLL-ia@m" would become "William"), and hashes the password using the *bcrypt* package. The API then creates a new entry in the `users` table of the database with all of the pre-mentioned data, along with the account creation time and date. After this, the API generates and sends an email to the inputted email address. The email contains a unique link using the *crypto* package to generate a random 20 byte token and converting it to hexadecimal [fig 1.1.9]. If the email address is already in use, an email is sent to the address asking whether they tried to register and informing the user that they already have an account. The API then sends a response to the client which generates an alert [fig 1.1.10] informing the user that a confirmation email has been sent to their email address. The alert is identical regardless of whether the email address is in use as this stops attackers from using this system to determine if a particular email address has an account. Once a user clicks the link in this email, upon creation of the page a POST request is sent to the API containing the token used to make the link unique. The API then updates the database to set that user as 'verified'.

When a user attempts to log in, a POST request is sent to the API with the inputted email and password. The first task is to check if there is an account with this email address. If an account exists, then the API uses the *bcrypt* package to compare the user's stored hash of their password against the password just entered. If all criteria are met, the user is redirected to the home page. If not, a response is returned to the client, not giving away which of the inputted credentials were wrong for security purposes [fig 1.1.11].

If a user cannot recall their password and chooses to reset it, submitting their email sends a POST request to the API with the inputted email address. The API then generates and sends an email to the inputted email address. The email contains a unique link generated using the *crypto* package to generate a random 20 byte token and converting it to hexadecimal. Once the link is followed, the user can input a new password. If these are identical and meet the determined *password-strength-utility* threshold, another POST request is sent to the API, which updates the user's password. Once this is completed, the API sends a response to the client, redirecting the client back to the login page.

In the account settings are the options for a user to change their email address or their password. If a user enters their password and a new email address, submitting these sends them to the API via another POST request. The API checks that the address is of the correct syntax, normalises it by changing all characters to lowercase, authenticates the password by using *bcrypt* to compare the inputted password against the stored hash of the user's password, and checks that the email address is not the user's current email address. The API then checks if the email address is already in use by another user. If it is, an email is sent to the address telling the user someone just tried to use their email and asking if it was them. If the email address is not in use, a confirmation email is sent with the same method as when registering for the first time - using a hex token to create a unique link. If instead a user is attempting to change their password, then a POST request is sent only if the new password and it's confirmation are identical and meet the *password-strength-utility* threshold. If the API is able to authenticate the user's current password, it uses *bcrypt* to create a hash of the new password, and then updates the database. It then returns a response to the client which determines the message displayed by the frontend (either a successful or unsuccessful password change).

Should a user wish to delete their account, the process is similar to logging in as it requires the email and password. However the API also checks that the email is the same as the one that the user logged in with (i.e. a user can only delete the account that they have logged in to). If all checks are valid then the database is updated and that user can no longer use those credentials to log in. A response is then sent to the client forcing it to remove it's login token and redirect to the login page.

3.1.3 Algorithms and architecture

The frontend of the application is written as Vue.js components (.vue). To create elements such as buttons and forms, the *element-ui* package is used. These are

more aesthetically pleasing than the standard HTML components, and provide additional functionality such as being able to set syntax rules for form inputs. This is used whenever the user has to enter a new email address as a preliminary check, however a regex test is also done on the API to guarantee the email address is of a valid format. In the login system all POST requests are bound to buttons that are disabled until the relevant form has been completely filled in. The API is an *express* app that can receive GET and POST requests sent to the API's http address. The API can then use the *MySQL* package to send queries to the database. The database credentials are stored in an *.env* file, which also contains the environment variables and credentials for the gmail account used to send emails. When the API receives a particular request, it contains code to be executed upon that request, before returning a JSON object to the client. Each JSON response will always contain a key 'status' with the value **true** or **false** to tell the client whether the request was a success.

3.1.4 Constraints

The most obvious constraint of the login system is that a user must have a valid and accessible email address to make an account. The only solution to this would be to have no login system at all, or to use a unique identifier such as in a username based system. However in the present day it is very common to own an email address, and it can help to prevent the use of bots. Having a login system also restricts the system as a user must sign up to play - a process that requires them to fill in several fields, confirm that they have read a page of terms and conditions, and then verify their email address. This process could deter some potential users from the game if they do not wish to give out their details. However it is good to have a login system as it can help to prevent bots using the site. It also allows results to be recorded for individual players, allowing for future in-game features such as a leaderboard and records of previous games.

3.2 Matchmaking service

3.2.1 User perspective

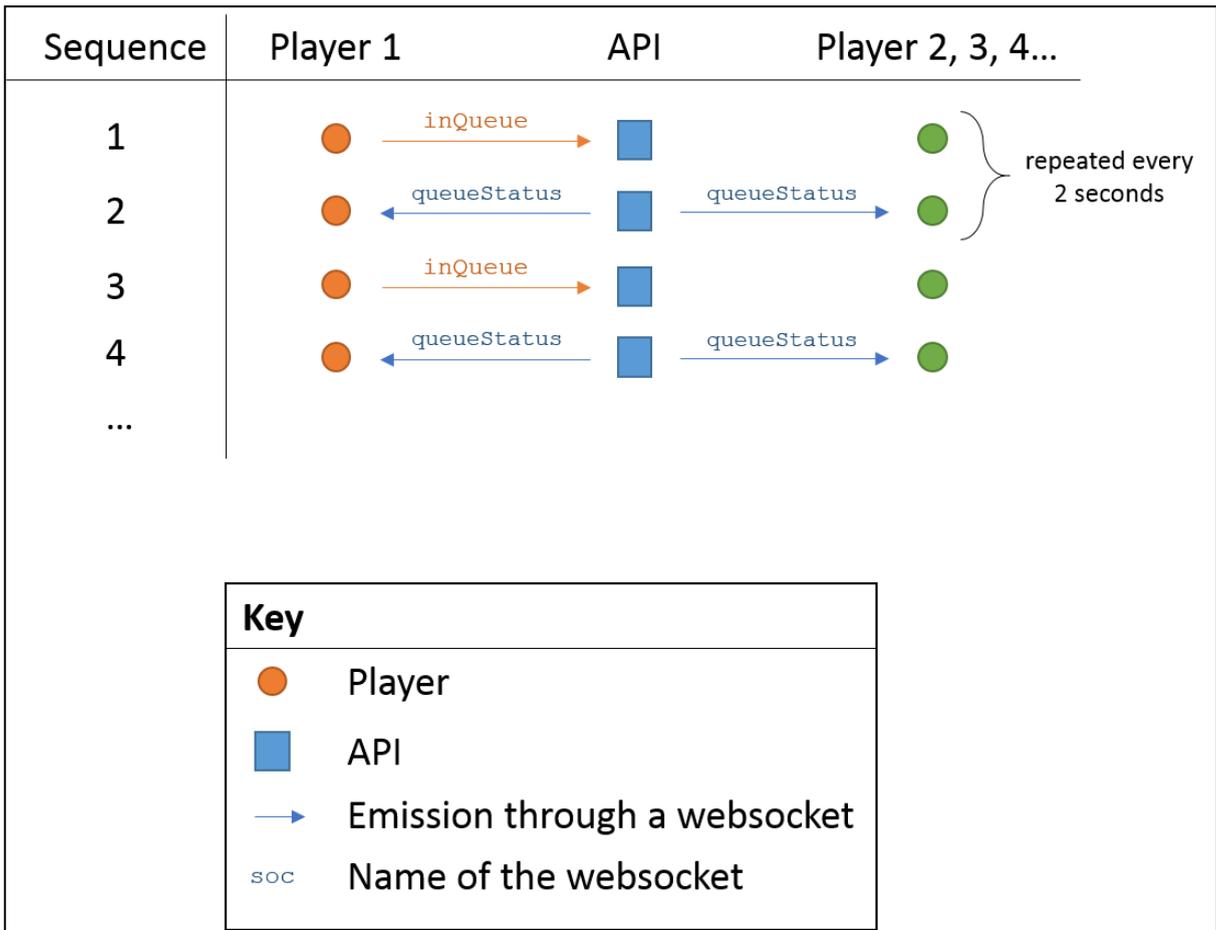
Once a user has registered an account and is logged in, they now have the option to play either the single-player game or the multiplayer game [fig 1.1.3]. If the user chooses to play the multiplayer game then they must first have another user to play with. The project brief instructed that a user be "automatically matched with a random partner". This means that the system must be able to match any users who wish to play the multiplayer game into pairs and create a game for them.

From a user's perspective, once they have selected to play the multiplayer game they are then presented with three game modes [fig 1.2.1] - synonyms, antonyms and hypernyms. Below each game mode is a counter to show the number of players who are currently queuing for this game mode, enabling the user to make their choice based on the likelihood of being immediately matched with another player. Once the user has selected a game mode, they join a queue and the page enters into a

waiting mode [fig 1.2.2]. As soon as the system finds another player queueing for the same game mode, the players are matched and are both redirected to their game.

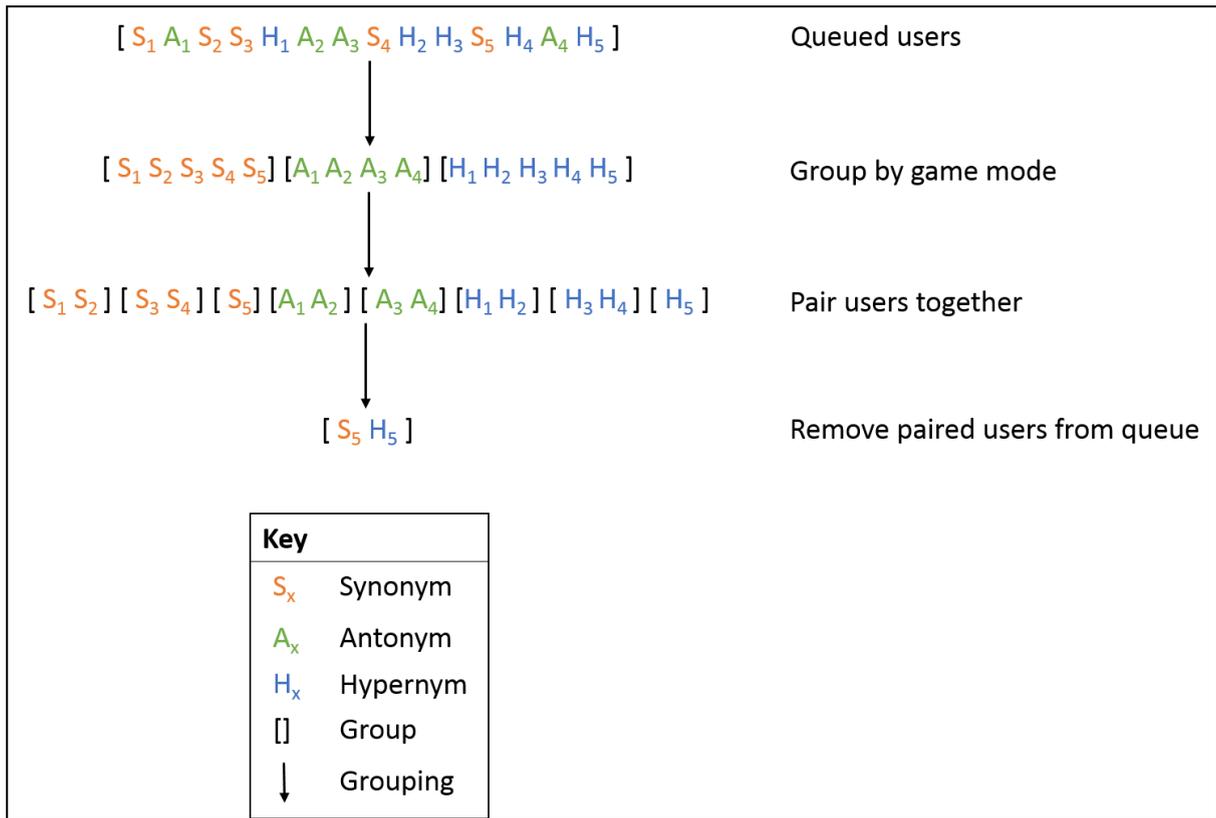
3.2.2 System behaviour

From the system's perspective there are two main things happening here. The first is the client displaying the number of players queueing for each game mode. When a user joins the queue for a game mode, a POST request is sent to the API adding the user to the `queued_users` table in the database. After this another POST request is sent every 2 seconds to the API as a "heartbeat" to let the API know that the user is still in the queue. Separately running on the Digitalocean droplet is a second file 'matchmaking.js' which shall be referred to as the 'matchmaking service' [fig 4.1.2]. This checks that the most recent heartbeat is no more than 5 seconds old, and if it is then it removes the user from the queue by marking their entry to the `queued_users` table as invalid. Also being sent every 2 seconds is an emission from the client to the API via a websocket. Websockets allow data to be pushed from the API to the frontend client without the frontend sending a request for the data. The client emits to the API that it is in the queue via the `inQueue` socket once every 2 seconds. The primary API then gets all users who are queued from the database and emits the data through the `queueStatus` socket, back to any frontend clients who are listening to this socket. The socket is only received by the multiplayer page, and uses the data to display the number of users queued for each game mode. All clients who are on this page will be listening to the `queueStatus` socket in the `queue` room [fig 5.1.1]. They will then receive the number of players in each game mode and update their values accordingly.



[Figure 5.1.1] Websocket diagram for users in the matchmaking queue

The second task that the system is performing is matching the queued users together. As well as maintaining the queue of users for the multiplayer game, the matchmaking service also pairs users together so that they can play the game. It does this by retrieving every valid entry from the `queued_users` table, splitting them into their respective game modes, and then matching every odd user with the next even user in the queue [fig 5.1.2]. These users are then marked as matched in the `queued_users` table, and new entries are added to the `multiplayer_games` and `multiplayer_answers` tables.



[Fig 5.1.2] Matchmaking algorithm diagram

3.2.3 Constraints

There are currently two known constraints of this solution. The first is that there is currently no way to leave to matchmaking queue except via refreshing the page. This is because the waiting mode [fig 1.2.2] is a fullscreen *element-ui* component that disables use of the page until a match has been found and the user has been redirected. The second constraint is an untested theory that some users who have a very poor internet connection may not be able to keep up with sending a heartbeat request every 2 seconds. If their connection goes down for more than 5 seconds, the matchmaking service will remove the user from the queue and they will have to rejoin the queue.

3.3 Multiplayer game

3.3.1 User perspective

Once the user has been matched with another player, they will both be redirected from the multiplayer menu page to the game. Both users will then see a countdown of up to 10 seconds before the game begins [fig 1.2.3]. This countdown will be in sync for both players, and the maximum value they see will depend on how quickly they are redirected to the game.

The game page [fig 1.2.4] has several pieces of information for the user, and several actions that the user can perform. The piece of information they are likely to see is

the word that they are meant to find synonyms/antonyms/hypernyms for. Below the input field is the definition of this word. It is important that the word is the largest text in the game as it should be the centre of the player's thoughts. When a player thinks of an answer, they can enter it into the input field, submitting the answer by either clicking the button at the end of the field, or by pressing the *enter* key on the keyboard. Multiple answers can be submitted at once if separated by spaces, and the answers are normalised by the system. If any of the answers entered are the same as the word itself or the same as any of the words in the definition, the answer will not be recorded. Those that are recorded will appear in the answers table in the bottom right of the screen [see fig 1.2.5 and fig 1.2.6 for an example].

Once the system detects that both users have entered the same answer for a word, they will both be moved on to the next word. They will also both receive a green notification [fig 1.2.7] at the top of the screen telling them that they matched and what the answer that they matched with was. The 'matched' counter on the right of the screen will also be incremented. Both users have the option to skip a word at any time (unless it is the final word). If player 1 skips a word, then their input field and the buttons on their screen will be disabled [fig 1.2.8], and they will have to wait for player 2 to confirm an alert telling them that the first player has skipped [fig 1.2.9]. Once the alert has been dismissed by player 2, both players will be moved on to the next word.

On the right of the screen is a box containing several pieces of information that are not critical to the user playing the game, but may help a user decide whether to skip a word or to persist with a word [fig 1.2.10]. Information displayed in the box includes the number of the current word and how many are left to go, the definition of the game mode (e.g. if playing synonyms, it will read "words with the same meaning, e.g. fast → quick"), the number of answers entered by the other player for this word, and the number of words that have been passed or matched so far during the game. All of these are updated in real time.

At the top of the screen is a progress bar that gradually decreases as the 2½ minutes of game time decreases. To the left of the progress bar is the remaining time written in minutes and seconds. The colour of the bar and text varies depending on the time left in the game - green for the first half of the game, amber for the third quarter, and red for the final quarter [see fig 1.2.11, fig 1.2.12 and fig 1.2.13].

Beneath the list of answers in the far bottom right of the screen is the quit button, allowing either player to quit the game at any time. This will force both players to the results page, giving the player who did not quit an alert telling them that the other player quit the game [fig 1.2.14]. This is one of 3 ways for the game to end. The second is for the users to skip and/or match their way through all 15 words in the game. The third way is for the players to run out of time. In all cases, the players are then redirected to the results page.

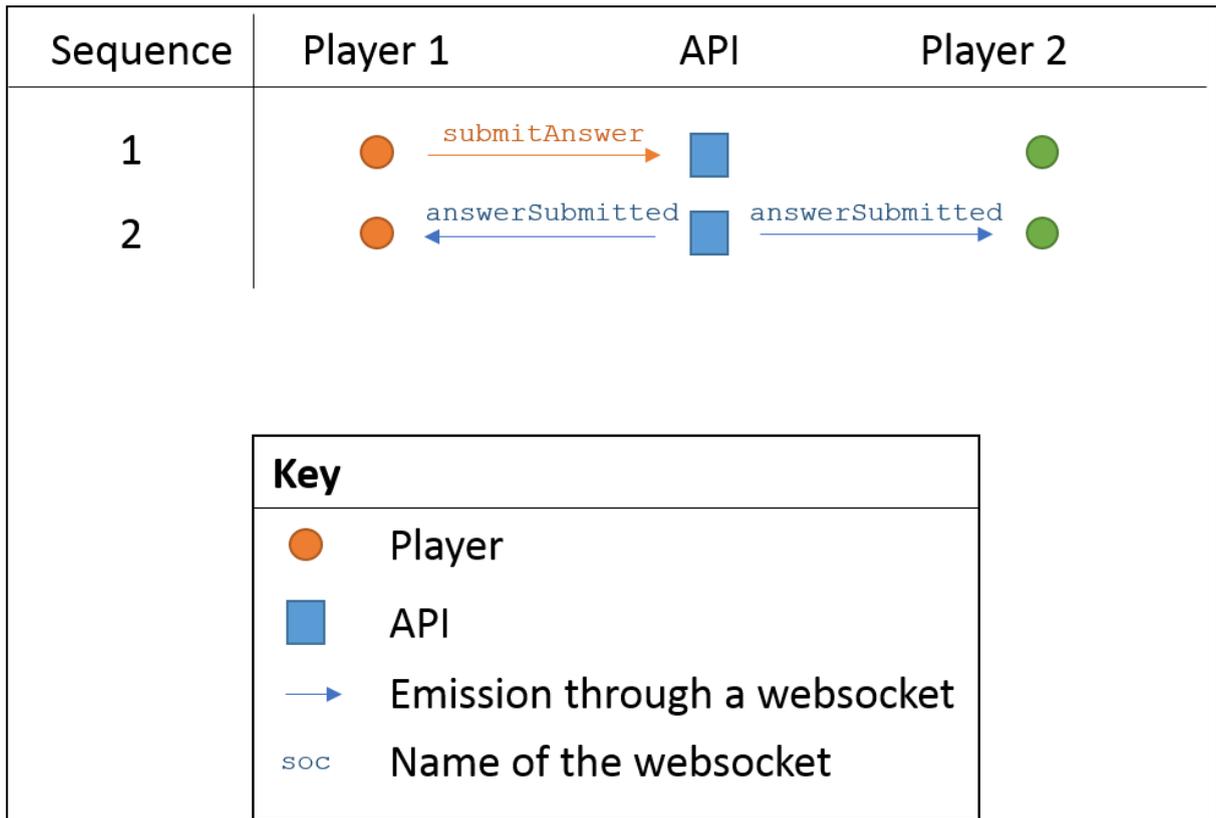
The results page displays the results of the game to both users in a table [fig 1.2.15]. The table contains a row for each word, also showing the match (if there was one) and both player's answers. The rows are colour coordinated - green if there was a

match, yellow if the word was skipped, and red if the word was never reached (either because the players ran out of time or one of them quit the game). At the top of the screen is a tally of each of these 3 outcomes.

3.3.2 System behaviour

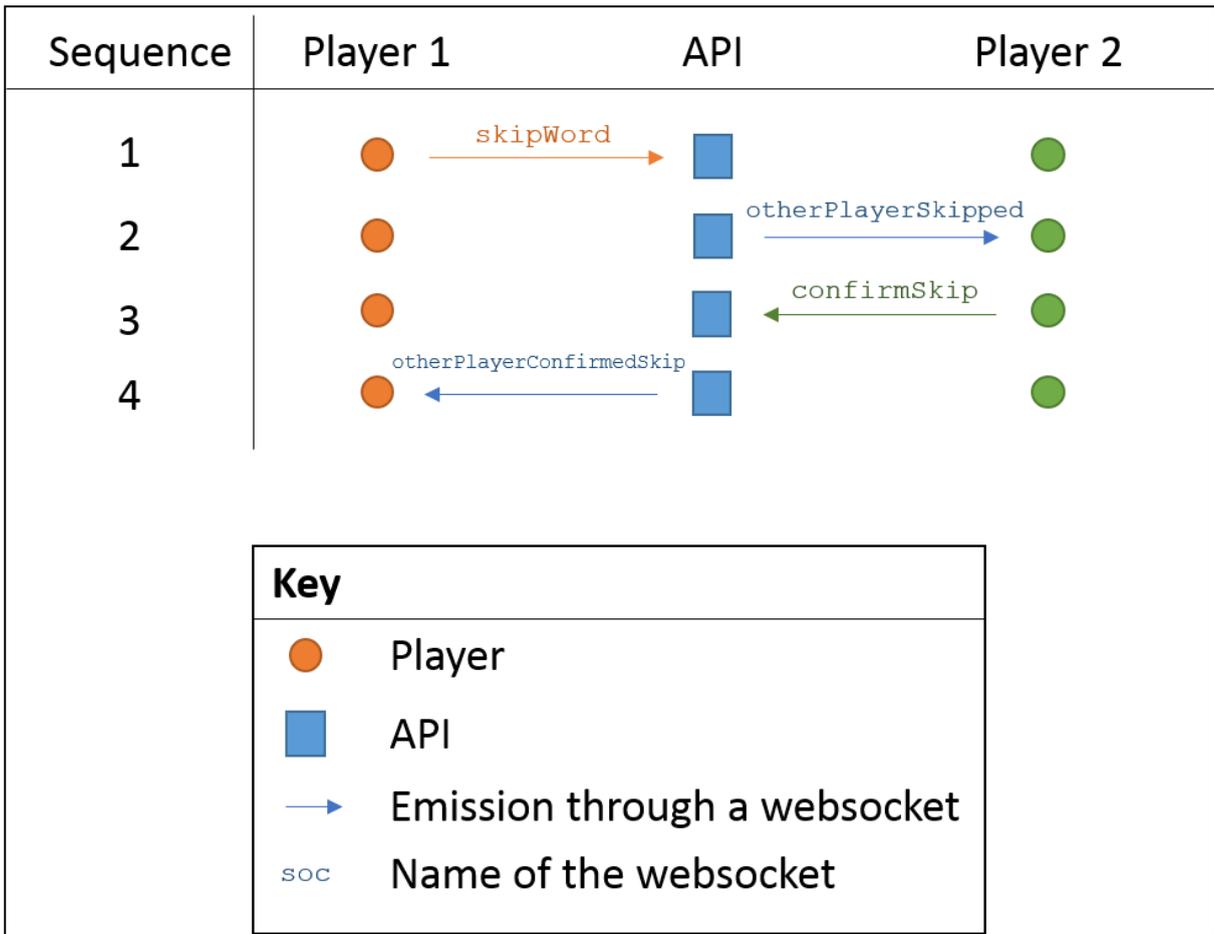
The page is created using a unique token generated by the matchmaking service when the match is made. This is used in the url of the game so as to give each game a unique link. When the page is created (either by redirecting from the matchmaking or by refreshing the page when in game), the client sends a POST request to the API asking for the game information. The API then retrieves the game information from the `multiplayer_games` table, as well as retrieving the game's current state from `multiplayer_answers` in case the game was already in progress and a user refreshed the page. This information is then returned to the client. One of the pieces of information that is retrieved is the end time of the game. This is a time in Coordinated Universal Time (UTC) decided upon by the matchmaking service, and is 2 minutes and 40 seconds after the match was made. Once both players have been redirected to the game after being matched, if there is more than 2 minutes and 30 seconds until the end of the game, the players will see a countdown on their screen. This means that both players will be in sync as the game will not start until a pre chosen UTC date and time 10 seconds after that match is made.

The words for the game are retrieved from the database by the API at the start of the game. These words are pre-selected at random by the matchmaking service. Once the client possess the words, they are stored in an array, and a counter value is used to index the array. Answers can then be entered into the input field, and when submitted they emit the word and that game's metadata to the `submitAnswer` websocket. So that multiple games can go on at once, each time a websocket is used in game, it is submitted to a room by the name of the game's unique token. When the API receives the broadcast that a word has been submitted, it compares it to all of the other player's previous answers. It then emits the result through the websocket `answerSubmitted` to **both** players [fig 5.2.1]. Depending on whether there was a match, 2 things can happen. If there was a match, both players receive a notification and are moved on to the next word. If there was not a match, then the player who did not submit the answer will have their count of answers submitted by the opposing player incremented. If a match is made on the final word in the game, then both users are redirected to the results page.



[Fig 5.2.1] Websocket diagram for answer submission

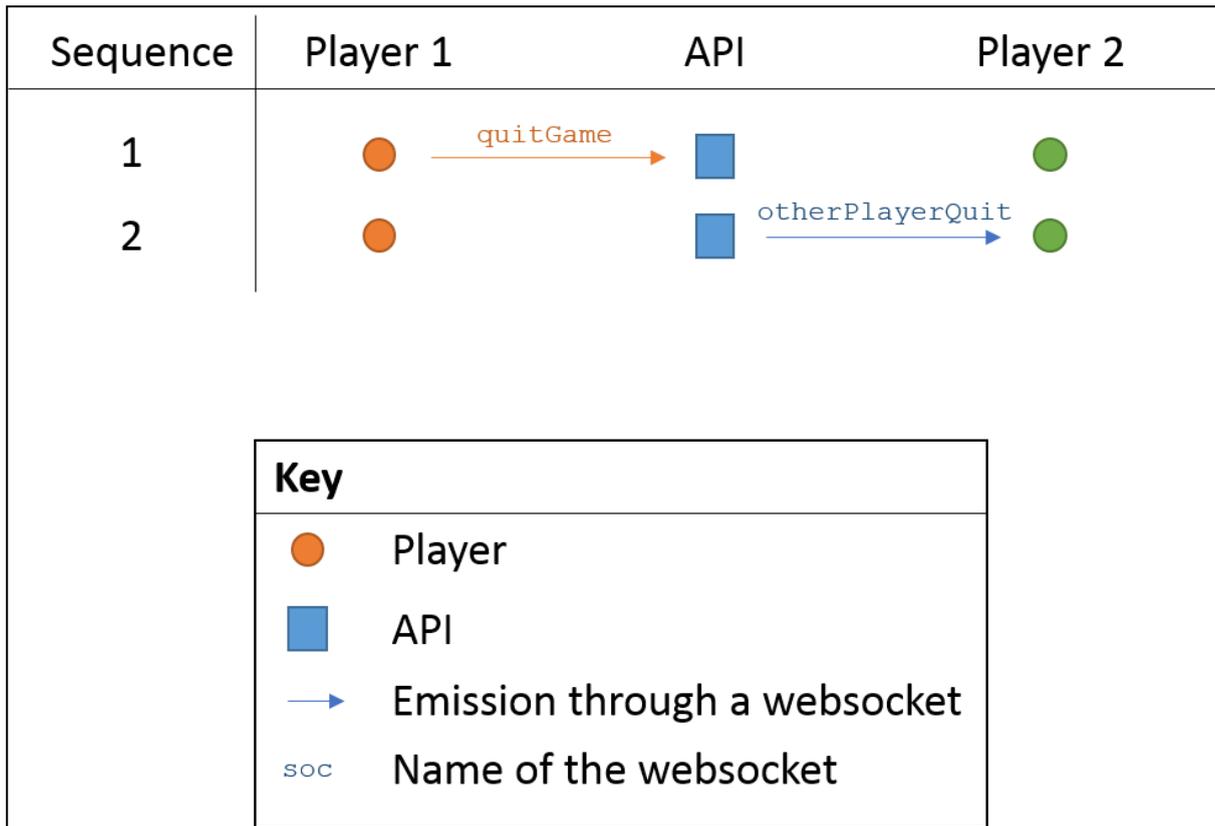
If a player (e.g. player 1) decides to skip a word by clicking the skip button, the game's metadata is emitted to the API using the `skipWord` websocket, once again in the room by the name of the game's unique token. Once the API receives this information, it updates the database accordingly and send out an emission through the websocket `otherPlayerSkipped`, which sends out to everyone in the room except for the sender of the original `skipWord`. As there are only 2 players in each room, it is just the opponent who receives the `otherPlayerSkipped` broadcast. Upon receiving this broadcast, player 2 will see an alert telling them that the other player has skipped the word. As soon as player 2 dismisses the alert, they are moved on to the next word, and a second broadcast is sent to the API, but this time it is through the `confirmSkip` socket. Once the API receives this, it send a final broadcast back to player 1 `otherPlayerConfirmedSkip`. This tells player 1 that the skip has been accepted, and it moves player 1 on to the next word [fig 5.2.2].



[Fig 5.2.2] Websocket diagram for skipping a word

The timer functions by comparing the actual UTC time to the game termination date once a second. It then calculates the difference and displays this as minutes and seconds. It also takes the total value in seconds and divides it by 150 (2½ minutes = 150 seconds) to get the percentage value for the progress bar. Once the number of minutes and seconds has reached 0, both users are redirected to the results page.

If a player (e.g. player 1) quits the game, the client emits the game's metadata to the API via the `quitGame` websocket. It then redirects player 1 to the results page. The API then updates the database, and emits to the `otherPlayerQuit` websocket to player 2. Upon receiving this, player 2 will be redirected to the results page and will see an alert telling them that the other player quit [fig 5.2.3].



[Fig 5.2.3] Websocket diagram for quitting the game

Upon creation of the results page, a POST request is sent to the API asking for the game's results. The API then gathers the relevant data from the `multiplayer_answers` table, and sends it back to the client, where it is displayed in a table.

3.4 Result processing service

3.4.1 User perspective

As mentioned in the 'background' (see Section 2), the lexical dictionary Wordnet which this project was originally designed to extend does not provide adequate word associations to support a single-player game. Instead a decision was made to use the results of the multiplayer game to support the single-player game. This means that the single-player game functions by showing words that have an adequate repository of synonyms/antonyms/hypernyms, and validating answers based on whether they are in this repository of word associations. From the user's perspective, when playing the single-player game, the answer to a word is either accepted or denied. Some answers that are more uncommon may not be accepted. An example of this situation would be if the word "cold" has the recorded synonyms "chilly", "freezing" and "bitter", but the player enters the word "icy". It is a synonym, but not one that the system is aware of, and hence it would not be registered as a match.

3.4.2 System behaviour

The third and final file running on the Digitalocean droplet is the result processing service (result_processing.js) [fig 4.1.2]. Once every 10 seconds, this service finds any previously unprocessed answers from the multiplayer game and puts them through a set of criteria to determine whether there is enough data for the word to be included in the single-player game. Currently the criteria for a word to be available in single-player are:

- The word must have occurred and a match found for the word at least 5 times in multiplayer games
- The word must have less than a 50% skip rate
- The word must have at least 2 unique matches

These criteria are currently relatively low, as a word could for example have 10 common synonyms but only 2 would get a match in the single-player game as they are the ones that have been recorded. However these criteria have been designed to be easily scalable so that when more multiplayer games have been played, the single-player game logic can also be improved.

3.4.3 Constraints

Currently the result processing service is able to take into account whether a word has been matched or passed, but it does not consider how fast these decisions were made. A match that is found on both player's first answer is considerably stronger than a match found after both players have entered several words and eventually got lucky. The system also does not consider the time that it takes for the answers to be entered, but could be programmed to take this into account. This would involve logging the time of every answer submission and at the end of the game calculating the relative times.

3.5 Single-player game

3.5.1 User perspective

From a user's perspective, the single-player game is almost identical to the multiplayer game. The only difference in the interface is that the game does not need to show the number of answers submitted by the other player as there is only one player [fig 1.3.1]. As the words for the single-player game rely on the results of the multiplayer game, there will always be a likelihood that there will not be 15 words available for each game mode. At the time of writing this, with the current criteria in place there are 4 synonyms [fig 3.1.1], 8 antonyms [fig 3.1.2] and 3 hypernyms [fig 3.1.3] that can be played in the single-player game. This is a problem that would become less important as the multiplayer game is played more and eventually there would be at least 15 words available for each game mode in single-player.

3.5.2 System behaviour

Due to there only being one player in the single-player game, there is no need for websockets. Instead all functions that require communication with the API are done via POST requests. The single-player menu where a user can select whether they want to play the synonym, antonym or hypernym gamemode does not need to use websockets as it does not need to show how many users are queueing for each gamemode. Instead it sends a POST request upon creation of the page to find out if there are words available for that gamemode in single-player. Currently, if there are less than 5 words available for a gamemode, the button is disabled and a user is unable to select that game mode [fig 1.3.2].

The functions that are run by the API for the single-player game are similar in most aspects to those for the multiplayer game, except that they do not have to deal with the complications of having 2 different players, and the tables accessed are `singleplayer_games` and `singleplayer_answers`. The main difference however is that there is no matchmaking function to decide which words will be in the game, generate a token and allocate an end time. Instead this is done by the API upon the user's selection of a gamemode using the `startSinglePlayerGame` POST request.

3.6 Website

3.6.1 User perspective

The game is available at the website <https://werdz.fun>. This is a short and easy to remember domain. The name of the game is "Werdz" - a comical misspelling of "words". The domain extension is .fun for two reasons - firstly it is cost effective compared to a domain name such as werdz.com, and secondly because it emphasises that the project is a game by using the word "fun" as the extension.

3.6.2 System behaviour

The frontend of the project is run at <https://werdz.fun> on Netlify, which automatically runs the command `yarn build` to build a production variant of the frontend. The backend of the project is run at <http://api.werdz.fun> on Digitalocean. To start or stop any of the three files running on the droplet, a developer must run first ssh onto the server. The files are run using pm2 - a Node.js process manager. To start a process, a developer must be in the correct directory and use the command `pm2 start file.js`, where 'file.js' can be replaced with a filename. After this, the process can be stopped with the command `pm2 stop file.js` and restarted with the command `pm2 restart file.js`. If changes are made to the code, then the developer must first use `git pull` to pull the changes from GitHub, and then run `pm2 restart all` to restart all processes.

To make sure that the website was secure, it was vital to obtain a certificate so that any data being sent from the client to the website would be encrypted [fig 4.1.4]. This was particularly important when dealing with user credentials such as emails and

passwords, particularly because the passwords are hashed after they have reached the API. This also develops trust with the users of the system as they will be able to see that the website has a secure 'https' address [fig 4.1.3], and their browser will not warn them that the site may be dangerous.

3.6.3 Constraints

The greatest constraint of the game being a web application is that it requires a steady internet connection to play. However it is very common in this day and age to have easy access to the internet via WiFi, ethernet or mobile data.

4. Implementation

4.1 Login System

To fully understand the implementation of this project, a basic understanding of the frontend framework Vue.js (see Appendix B) and the architecture of the database (see Appendix C) is required.

4.1.1 Database queries

The login system is the most code-heavy part of the project, with 10 different POST requests and 9 Vue pages. However the majority of this code is relatively simple, consisting mostly of POST requests which ask the API to check various credentials against those in the database, update the database accordingly, and return the appropriate values. For the API to be able to query the database, a universal database query function was created in the file 'db.js'. In this file, the database credentials are accessed from the `.env` [fig 2.1.1] containing the environment variables, and these are used to first connect to the database and then send the query. The function also handles errors and bad queries by using a JavaScript promise [fig 2.1.2]. Promises are used to make sure an asynchronous function returns a value by promising this value to the process that executed the function. The promise then resolves the query results if it has been successful, and rejects to the promise if there was an error. This function can then be imported and run anywhere in the backend [fig 2.1.3], but it must always be predated by `await` to tell the system to wait for the promise's response (e.g. `const user_id = await db.qry(`SELECT user_id FROM users WHERE email = ?`, [req.body.email])`). The *nodemailer* package already protects against SQL injection by checking any variables that inserted into the query using question marks. If the value has dangerous text that could be malicious, then the statement is not executed and an error is returned.

4.1.2 POST requests

For the client to be able to send POST requests to the API, the *axios* package is used. The function for a POST requests is imported from the file 'auth.js' in the frontend [fig 2.1.4]. Here the URL is defined based on the Node environment [fig 2.1.5] - if the environment is `development` then the base URL is '<http://localhost:8080/>', and if the environment is `production` then the base URL is '<https://api.werdz.fun/>'. The endpoint is then added (it is fed to the function from the vue page) along with the method (POST or GET) and the data object that is passed with the request. For this system only POST requests are used as they send the data as metadata whereas a GET request sends the data as part of the url, making it less secure.

4.1.3 Emails

A key part of the login system is the ability for the system to send emails to a user. There are 5 possible emails the system can send - a confirmation email for a user

signing up, a warning email telling a user someone just tried to sign up with the email address they use for their account, a confirmation email for when a user changes their email address, a warning email telling a user someone just tried to change their email to the email they use for their own account, and finally the password reset email. These are all specified in the 'mail.js' file, and all use the same `newMail()` function. This is similar to the database query function as it uses credentials from the `.env` file to send an email [fig 2.1.6]. It also has similarity to the POST request function as it must first use the Node environment to determine whether links in the emails should use the localhost address or website address [fig 2.1.7]. The email bodies are imported from HTML templates in the templates directory of the backend [fig 2.1.8], which have variables such as the recipient's name and unique links such as a password reset link.

4.1.4 Registering input validation

On the register page where users sign up for the first time, when defining the Vue component data, 3 validation methods are also specified for the input fields. The first is `passwordStrength()` [fig 2.1.9] - this uses the password strength score generated by *vue-password-strength-meter*, and provided the strength is above 1 (out of a maximum of 5) then the password is accepted. Originally the validation required the strength to be over 2, however upon feedback from users the requirements were too high compared to other websites. A password strength of over 1 still filters out most common passwords such as single words and common phrases. The second validation is `confirmPassword()` [fig 2.1.10] - this checks that the second entry of the password is identical to the first. The final validation is `isAString()` [fig 2.1.11] - this checks that the forename and surname fields do not contain any characters other than letters and spaces. It does this with a regex test of the form `const re = /^[a-z]+\s)*[a-z]+$/` - the first part `^[a-z]+\s)` allows the use of uppercase letter and spaces, and the second part `[a-z]` allows for the use of lowercase letters. When the data is declared after the validation methods, the rules for each input field are also defined. Each rule has a criteria, a warning message, and trigger to cause that message to be displayed. For example, the forename field has 2 rules - the first is that the field is required and will display a warning message on 'change', which means when the message will be displayed whenever the input field is empty. The second rule is the `isAString()` message defined earlier. The warning message for this is displayed on 'change' and on 'blur', meaning it will also be displayed when a user clicks out of the input field. These rules are then processed by the *element-ui* package, and changes the colour of the input fields accordingly [see fig 2.1.12, fig 2.1.13 and fig 2.1.14].

4.1.5 Redirects

The majority of the Vue components contain the method `redirect(location)` [fig 2.1.15 and fig 2.1.16] which redirects the user to another page on the site. The location is defined by the page's name in the routing index [fig 2.1.17]. However the sign up page contains a redirect to the page containing the sign up terms and conditions. Originally, clicking this link would redirect the user straight to this page, meaning that any input they had entered to the form would be lost when the user

returns to the page. The easiest way to solve this was to have the link open in a new tab. This is done by first defining the route as a variable, and then using the route's address in a new tab [fig 2.1.18 and fig 2.1.19].

4.1.6 Timeline of work

The first week of the project was spent setting up a development environment. This involved setting up an AWS relational database, creating a backend API using the *express* package, setting up the frontend Vue client, creating a *.env* file and config to allow the data inside it to be accessed, and setting up a GitHub repository for the project. Once the development environment was initiated, the next step was to set up communication between the frontend and backend via the 'auth.js' file (see Section 4.1.2). After this, the login system could start to be developed. The first task was to research valuable features of a login system, and those that appeared to be most vital were password hashing, confirmation emails, and giving the user the ability to manage their account post sign-up. The first page that was developed was the login and registering page. These were on the same page so that if a user entered details into one form and then switched to the other, when they return to the original form their input would still remain. Once a user had inputted their details, the API would hash the password using the *bcrypt* package [fig 2.1.20], and insert the data into the `users` table using a database query function. The API would also send an email to the inputted email address - either a confirmation email with a unique link (unique because of a token generated using the *crypto* package [fig 2.1.21]), or a warning email telling the owner of the account who uses that email address that someone had attempted to register an account with their credentials. The next task was for the login system to be able to confirm the user's details against those in the database. To check the validity of the inputted password, *bcrypt* has a function `compareSync()` which takes a plaintext password and a hash of a password and returns a boolean value for whether they match [fig 2.1.22]. The same method is used in account settings - when the user is changing their email/password or deleting their account, the plaintext password and hash are compared using *bcrypt* (e.g. `const authenticated = await bcrypt.compareSync(req.body.current_password, user.password)`). When a user logs in, the user's credentials are used to create a web token using the package *jsonwebtoken* [fig 2.1.23]. This token is returned to the frontend which sets the token in the local storage of the browser. Each time a page is loaded, the token is checked, and if invalid then the user is redirected back to the login page and the token value is set to null. The login system in total took around 3 weeks to complete including the initial setup of the development environment.

4.1.7 Element-ui issue

One major problem that emerged as the project developed was the lack of documentation for *element-ui*. This is because the package was originally developed in Chinese before the docs were translated to English. There were alternative frontend design packages, however *element-ui* had a lot of favourable characteristics such as the ability to create custom rules for input fields of a form.

4.2 Matchmaking Service

4.2.1 Matchmaking.js

Once the login system was completed, the next step was to create a matchmaking system that could pair users together who are queueing for a multiplayer game. For this to happen, a matchmaking service was created by running the file 'matchmaking.js' on the server [fig 4.1.2]. Once the matchmaking service is running, it recursively executes 2 predefined functions with a variable time interval between the executions. That time interval is a minimum of 2 seconds and a maximum of 20 seconds. The first function is `checkGames()` which checks over all multiplayer games in the `multiplayer_games` table to find any that are invalid, and removes them. The second function is `checkMatches()` which checks for users in the multiplayer queue, and matches them together if they chose the same game mode (synonyms, antonyms or hypernyms). Every time `checkMatches()` is executed, it returns a boolean dependant on whether there are any users in the queue. If there are, then the time interval of the function is reset back to 2 seconds. If there are no users in the queue, the time interval is incremented by 3 seconds each interval up to a maximum of 20 seconds. Once users are detected in the queue, the interval time is reset back to 2 seconds [fig 2.2.1].

4.2.2 checkMatches()

The `checkMatches()` function first selects all valid queued users from the `queued_users` table. If there are none, the function immediately returns **false**, causing the interval time to be incremented by 3 seconds. The next task for the matchmaking service is to separate the users with a valid heartbeat from those without one. This is done by checking whether each user's last heartbeat was less than 5 seconds ago [fig 2.2.2]. As the heartbeats are meant to occur once every 2 seconds from the client when a user is queued, this means that even if for some reason one heartbeat is missed, the user can still stay in the queue. The user ids of both groups are recorded, and those in with a dead heartbeat are set to removed in the `queued_users` table. If by this point there are no users with a valid heartbeat, the function returns **false** and the time interval is incremented. Next the valid users are grouped by their gamemode using the *lodash* package's function `_.groupBy()` which can group an array of objects by a key of the object [fig 2.2.3]. The grouped users are then fed into the function `getQueryParams()`.

4.2.3 getQueryParams()

As the code in the matchmaking service is run every 2 seconds (or more), having a low runtime is critical to making sure that the process intervals do not overlap. To make sure this is the case, the function needs to have as few asynchronous functions as possible. The majority of the asynchronous functions in the matchmaking service are database queries, and so to reduce the number of queries a non-standard method of inserting values into tables had to be used. Every time the matchmaking service matches 2 users together, several database queries have to be performed:

- The queued user's entries in the `queued_users` table must be updated
- Any valid previous multiplayer games with either player must be ended
- A new game must be inserted into the `multiplayer_games` table
- The game's words must be inserted into the `multiplayer_answers` table

If separate queries were performed for each match, then there would be 4 queries for each match, resulting in a large runtime if there are a lot of users queuing. To prevent this, the matchmaking service creates a string of values to be inserted into each table [fig 2.2.4], and then inserts them all at once. This means that no matter how many matches are made, there will only ever be 6 MySQL database queries. The function `getQueryParams()` does this by looping through each game mode, and then looping through half of the users in that game mode to match them together with the other half of the queued users [fig 2.2.5]. For each query, a string is created which has data appended to it in each loop. For example, after the loop a string `queued_values` may read `(1, 'William', 'Cooter'), (2, 'Bob', 'Jinska')`. The MySQL statement would then read `await db.qry(`INSERT INTO queued_users (id, forename, surname) VALUES ?`, [queued_values])`.

4.3 Multiplayer game

4.3.1 Get game information and state

After a pair of users have been matched together by the matchmaking service, they are redirected to a unique game link. As the page is created, a POST request is sent to the API requesting the information for that game. Before it can return any data, the API formats the time values correctly by adding 1 hour to the value from the database [fig 2.3.1]. This is because the time is saved to the database by the matchmaking service in the format `'YYYY-MM-DD HH:mm:ss'`, and in doing so the knowledge that it is a UTC timestamp is not retained. Therefore to get the time back into UTC, 1 hour must be added to the time. The API also has to set the placeholder of the input field depending on the chosen game mode, and determine the player number specified in the `multiplayer_games` table by comparing the user id of player 1 to the user id of the user requesting the game information. If they are the same, then that user is player 1. Otherwise they are player 2.

If the game page is reloaded part of the way through the game, the game's state must also be recovered. The first piece of information the API has to recover is the current word index in the array of words. This is calculated by filtering the answers for the game by those who have not been matched or passed using the `lodash_.filter()` function, and subtracting the length of this array from the length of the original array [fig 2.3.2]. This gives the index of the current word. For example, if no words have been matched or passed, the length of the filtered answers will be identical to the length of the unfiltered answers, so subtracting one from the other gives 0, i.e. the first word in the array. Other data that the API recovers includes the matched and passed count by filtering the answers using either the `matched` or `passed` attribute with the `lodash_.filter()` function [fig 2.3.3], the current word answers by looping through all the answers for **that** player at the current word index and recording them all as an array of objects in the form `[{answer:`

`ans}], {answer: ans}]` [fig 2.3.4], and the other player answer count by getting all the answers for the **other** player at the current word index and counting the length of that array [fig 2.3.5]. Once all of this data has been returned to the client, these values are set in their respective data items [fig 2.3.6].

4.3.2 Timer

The most complex part of the game is the 2½ minute timer. One of the items of data generated by the matchmaking service is the termination date of the game - a UTC timestamp 2 minutes and 40 seconds after the match is made. The multiplayer game Vue component imports another component 'Timer.vue', which can call 2 methods in the multiplayer game [fig 2.3.7]. The first is `startGame()` [fig 2.3.8], which tells the game to start and sets the data item `game_started` to **true**. The second is `delayGame()` [fig 2.3.9] which receives the current time in seconds until the end of the game, subtracts 150 from it (150 seconds = 2½ minutes), and displays this number as a countdown until the game begins. Originally the time functions using the *moment.js* package were not using UTC and were just using standard GMT time. However this presented the problem that users playing from non-GMT time zones were unable to play the game as the timer for the game uses the client's local time. For this reason the times across the system were all converted to UTC.

The timer component works by running the functions `compute()` and `incrementTime()`. In the 'watch' section of the component is the function `actualTime()` [fig 2.3.10], which watches the data value `actualTime` and executes the `compute()` function every time a change is detected in the `actualTime` value. This value is updated every 1 second by the `incrementTime()` [fig 2.3.11] function, and so the `compute()` function is also executed every 1 second. The `compute()` function gets the time difference between the end of the game and the current time using the `getDifference()` function [fig 2.3.12], and sets the variable `current_time` to this value in seconds. If the current time is less than or equal to the game length (150) then the timer emits `game_start` [fig 2.3.13] to the multiplayer game, causing it to run the `gameStart()` function. If the current time is greater than the game length then the timer emits `delay_game` [fig 2.3.14]. If the current number of minutes and seconds is 0, the timer redirects the user from the game to the results screen [fig 2.3.15]. When the timer is first initiated, the `current_time` in seconds is 0. So that the game does not immediately redirect to the results screen, the data value `timer_is_going` starts as **false**, and several functions require it to be **true** to execute. The first time the number of minutes and seconds is 0, the value is set to **true**. Then the second time the number of minutes and seconds is 0 (when the game actually finishes), the user is redirected. The `incrementTime()` function sets the actual time to the current time in UTC (generated using *moment.js*). It then sets a timeout of 1 second to call itself again, meaning that the function will run every 1 second. In this time it also determines the colour of the progress bar of the timer, sets the `current_time` value to be the number of seconds and the number of minutes multiplied by 60 to give the total number of seconds remaining, and finally uses the value to set the bar progress percentage.

The HTML template of the timer has 2 parts. Firstly on the left of the component is the time remaining in minutes and seconds displayed as text. Secondly there is a progress bar from the *element-ui* package, used to display the time remaining as a percentage of the initial 2 ½ minutes. The `incrementTime()` function manages the colour of the text and bar depending on the time remaining:

- If there is over half the allotted time remaining (2min 30sec - 1min 15sec) then they are green (#67C23A).
- If there is less than half but more than a quarter of the time remaining (1min 15sec - 0min 33sec) then they are amber (#E6A23C).
- If there is a less than a quarter of the time remaining (32sec or less) then they are red (#F56C6C).

These are the default colours for the components in the *element-ui* package [fig 4.1.5], so these were used to keep the app consistent.

4.3.3 Submitting answers

Once the game has begun, the main action a player can take is to submit synonyms/antonyms/hypernyms for the word that they are being shown. Answers are entered into the input field below the word. When an answer is submitted by a player (say player 1), a lot of data needs to be sent to the API. This includes the game id so that the API knows which game to reference in the database, the current word that the players are trying to find associations for, all of the answers that player has entered for this word, the player's user id, the player's player number (in this case it would be '1' for player 1), the game token, the current word index, and the maximum word index (the index of the final word in the game, e.g. for a game with 15 words, this value would be 14). All of this data is ready to be submitted immediately except for the answers that a player has submitted. These must first be validated against several criteria, which is done in the frontend as these answers need to be displayed in the 'previous answers' section of the page. The game can take multiple words in the same input so long as they are separated by spaces. To allow this, the initial processing of the answer string is to separate it into individual words. This is done with the `lodash_.words()` function which separates a string into an array of words [fig 2.3.16]. It also removes any punctuation. After this, the system loops through this array of answers, testing 4 criteria on each answer [fig 2.3.17]. These criteria are:

- The answer must not be already in the list of answers submitted for this word (i.e. not repeats). This is done using the `lodash_.filter()` function to filter out all answers that are not the new answer, and checking the length of this array. If the array has length, that means this answer has already been submitted, and the condition returns **false**.
- The answer must exist. If for some reason the `lodash_.words()` function returns an empty string, the condition returns **false**.
- The answer must not be the word that the players are trying to find synonyms/antonyms/hypernyms for. If the answer and the word are identical (even after the answer has been normalised), then the condition returns **false**.

- The answer must not be any of the words in the definition of the word. For example if the definition of the word “cold” is “low temperature”, then the words “low” and “temperature” will cause the condition to return **false**. This is done by using the *lodash* `_.words()` function on the definition, and then using the *lodash* function `_.includes()` to check if the array of words contains the answer.

If all 4 of these conditions return **true**, the word is added to the answers data item so that it can be displayed on screen, and also added to the answers in the data that is to be sent to the API. If the length of the answers is greater than 0 (i.e. there were some valid answers), then the data is emitted to the API using the `submitAnswer` websocket.

By submitting the game token with the data, the API connects to a websocket room by the name of that token [fig 2.3.18]. This means that for each game there is a unique websocket room for that game’s sockets to pass through. It also keeps the room private as the only users who have access to the token are the players of that game.

Once an answer has been submitted, the first thing the API does is use the submitted player number to define the variables `this_player_number` (which in this case would be the string “p1_answers”) and `other_player_number` (which would be the string “p2_answers”) [fig 2.3.19]. These strings are then used to define whose answers belong to which player when retrieving the answers submitted so far from the `multiplayer_answers` table. As these answers are currently in the form of stringified arrays, they must be converted back to real arrays using the code `JSON.parse(answers)`. This is wrapped in a try catch error handle incase for some reason the answers have not been saved correctly or there is an error in parsing them back to real arrays [fig 2.3.20]. There are now 2 arrays of words - `this_player_words` containing player 1’s answers, and `other_player_words` containing player 2’s answers.

The next step is to complete player 1’s answers by adding each new answer to the array `this_player_word`. Then any crossover can be found using the *lodash* `_.intersections()` function, which finds any identical elements in the 2 arrays [fig 2.3.21]. If any identical elements exist, the first one is taken as the matching answer.

If there was a match then the process is to firstly update the `multiplayer_answers` table, setting that word’s matched value to 1, recording the matched word, and updating the list of player 1’s answers to now include any new answers. Then, if the `current_word_index` value is equal to the `max_word_index` value (i.e. this is the last word in the game), any words in the `multiplayer_answers` table that are not marked as matched or passed are marked as incomplete. There is no reason that an answer should not be marked as one or the other, however on occasion for a reason that is still unknown this can happen, so this is a temporary fix until the issue is diagnosed and fixed. The table `multiplayer_games` is then updated to be completed and no longer valid. Finally

the API emits to the websocket `answerSubmitted` with the data that the status of the submission is **true** and the answer that was the match [fig 2.3.22]. This socket is emitted to **both** players.

If there was not a match then the process is firstly to update the `multiplayer_answers` table similarly to if there was a match, but the only attribute that is changed is player 1's answers which are updated with the new answers. The API then emits to the websocket `answerSubmitted` with the data that the status of the submission is **false**, player 1's user id, and the total number of answers submitted by each player for this word (`this_player_word_count` and `other_player_word_count`) [fig 2.3.23]. This socket is emitted to **both** players.

The client is constantly listening for the socket `answerSubmitted` in the room with the same name as that game's token [fig 2.3.24]. Both players will simultaneously receive the emission to this websocket and perform the same action, regardless of whether they were the one to send the initial emission to the `submitAnswer` socket (i.e. at this stage players 1 and 2 both do the same thing). When the client receives the data emitted by the API through the `answerSubmitted` socket, the first thing it does is to check the status.

If the status is **true**, then the players have found a matching synonym/antonym/hypernym for the word, and can move on to the next word. The game first displays an *element-ui* notification telling the players that they matched, and shows them the word that they matched on [fig 2.3.25]. If the current word index is equal to the maximum index of the list of words for the game (i.e. that was the last word in the game), then the players are redirected to the results screen [fig 2.3.26]. If it was not the last word in the game, then the data value `matched_count` is incremented (used to show how many words have been matched in the game so far), and the function `nextWord()` is executed [fig 2.3.27].

The function `nextWord()` moves the game on to the next word. This function is used when a match is found or a word is skipped. The function resets the input field, answers, and the count of the opponent's answers all back to their default values. It then increments the current word index, and if the game is moving on to the final word it disables the skip button.

If the status is **false**, then the total number of answers submitted by the other player is updated to the current value using either `this_player_word_count` or `other_player_word_count`, and the players continue submitting answers for that word.

4.3.4 Skip word

If neither of the player can think of an answer in common, either player can choose to skip the word. When a player (say player 1) skips a word, the data sent to the API is the game id, the current word, the user id of the player skipping the word, and the game token. These are emitted through the `skipWord` websocket [fig 2.3.28]. The skip and submit buttons for player 1 are then disabled, and the skip button text now

reads “waiting for other player to skip”. Player 1 will remain in this state until the skip has been confirmed by player 2.

When the API receives the emission to the `skipWord` socket, the relevant row of the `multiplayer_answers` table will be updated so that ‘skipped’ is **true**. The API will then emit to the socket `otherPlayerSkipped`, however unlike when submitting a word, this socket will only be received by members of that websocket room who did *not* send the initial `skipWord` emission. This means that player 1 will not receive the `otherPlayerSkipped` emission, and as player 2 is the only other member of that websocket room, they will be the one to receive the emission.

Upon receiving the emission, player 2 will be shown an *element-ui* alert telling them that the other player has skipped the word [fig 2.3.29]. Once the alert has been confirmed or dismissed, player 2’s `skipped_word_count` data item is incremented, their `nextWord()` function is executed, and an emission is made to the `confirmSkip` websocket. The API then emits the `otherPlayerConfirmedSkip` websocket [fig 2.3.30], but it only emits to the player who did not emit to the `confirmSkip` socket (i.e. the `otherPlayerConfirmedSkip` socket is emitted to player 1 - the player who initiated the skip). Player 1’s pass count is then incremented, their skip and submit buttons re-enabled, and the `nextWord()` function is executed [fig 2.3.31].

4.3.5 Quit game

If a player (say player 1) decides to quit the game using the quit button, the same data is sent to the API as if the user were skipping a word, but instead the data is emitted through the `quitGame` websocket. Player 1 is then redirected to the results screen and is no longer listening to any of the websockets [fig 2.3.32].

When the API receives the emission it makes 2 updates to the database. The first is to set ‘quit’ to be **true** in the `multiplayer_games` table, and the second is to set all remaining words for that game in the `multiplayer_answers` table to ‘uncompleted’. The API then emits to the websocket `otherPlayerQuit` [fig 2.3.33]. Upon receiving the emission to the `otherPlayerQuit` socket, player 2 is redirected to the results screen and is shown an *element-ui* alert telling them that the other player quit the game [fig 2.3.34].

4.3.6 Results screen

Once the players have been redirected to the results page (either by completing the game, quitting the game, or running out of time), the client sends a POST request to the API asking for the multiplayer results. The API then returns the answers both players gave, the matched word of each round if there was a match, and the total counts of matched, passed and uncompleted words. The results are displayed by the client in an *element-ui* table. The *element-ui* package is particularly useful here as it can display a JSON object as a table. After the data is returned from the POST request, the results of the query to the `multiplayer_answers` table is assigned to the data item `table_data`. This is then selected as the data for the *element-ui*

table, and the keys of the object are used to determine the values in the table attributes. For example, the first attribute is an icon that indicates whether the word was matched, skipped or uncompleted. Using Vue's "v-if" attribute in the row's HTML tag allows the table to variably show a green success icon if the data's matched attribute is **true**, an amber warning icon if the data's passed attribute is **true**, and a red error icon if the data's uncompleted attribute is **true** [fig 2.3.35]. Each row also has a background colour indicating its status. These are defined in the style section of the Vue framework [fig 2.3.36], and use the default lighter colour values of the *element-ui* package for consistency.

4.3.7 Issue with definitions

An interesting issue that arose while selecting words for the game was how to write the definitions of the word. Most definitions use synonyms, antonyms or hypernyms to define the word they are discussing, for example to define the word "tractor" one might say "a farm vehicle". However this gives away the hypernym "vehicle", so to avoid this the definitions had to be selected very carefully. In the case of the word "tractor", the chosen definition was "What a farmer drives".

4.4 Result processing

4.4.1 checkWords()

The result processing service executes the function `checkWords()` every 10 seconds to process results from new multiplayer games and use these results to improve the single-player game [fig 2.4.1]. The function starts by retrieving all non-processed answers from the `multiplayer_answers` table. It then groups these answers by their game mode using the *lodash* function `_.groupBy()` [fig 2.4.2]. Then for each of the 3 game modes (synonyms, antonyms and hypernyms) it creates an object where each key is a word, and each value is another object containing an array of the matched answers, and counts of the matches, skips and completions [fig 2.4.3], to look something like this JSON object:

```
{
  cold: {
    matched_words: ['freezing'],
    matched: 1,
    passed: 0,
    uncompleted: 0
  }
}
```

The service then creates a string of the words [fig 2.4.4] in the form (`'cold', 'hot', 'ugly'`), which it uses to query the relevant game mode table (e.g. the `synonyms` table if processing synonym games) [fig 2.4.5]. This table contains the total occurrences, matches and skips for each word in all recorded games. The matched and skipped values are added to their counterparts in the JSON object, and 2 more keys are created in the object. The first is occurrences,

which sums the previous occurrences, new matches, new passes and new incompletions. The second is the previous answers, which is all of the previous answers from the game mode table. The previous answers are in the form of a JSON object where the key is an answer and the value is its frequency. Once they have been parsed using `JSON.parse()` and appended to the original JSON object it will look something like this:

```
{
  cold: {
    matched_words: ['freezing'],
    matched: 4,
    passed: 1,
    uncompleted: 3,
    total_occurrences: 11,
    previous_answers: {
      'chilly': 2,
      'freezing': 4
    }
  }
}
```

This JSON is then used to update that game mode table so that the attributes `multiplayer_answers`, `multiplayer_occurrences`, `multiplayer_matches` and `multiplayer_passes` reflect both the previous results and the new results.

The next step is for the system to query that game mode table again and retrieve **all** rows of that database. Then for each row, the data is used to determine whether or not the word of that row should be available for the single-player game [fig 2.4.6]. As stated Section 3.4.2, the conditions for a word to be available are:

- Having at least 5 matches, determined simply by checking the number of multiplayer matches.
- Having a skip rate of less than 50%, determined by dividing the number of passes by the sum of passes and matches. It is important that the total occurrences is not used here as this includes occasions when the word was uncompleted because the players never got to that word in the game.
- Having at least 2 unique matches, determined using the `lodash_.keys()` function to determine how many keys the answers JSON object has.

If all 3 conditions are satisfied, the unique id of the row is appended to the `available_ids` string. If not satisfied, the unique id of the row is appended to the `unavailable_ids` string. Both strings will end up in the format (2, 3, 5) and will then be used for a query to the game mode table to update the 'singleplayer_availability' attribute to either **true** or **false**. Once all of this has been done for all 3 game modes, the final step is to mark the answers as processed in the `multiplayer_answers` table.

4.4.2 resetValuesForTesting()

The result processing service contains one other function `resetValuesForTesting()`. This resets all 3 game mode tables to their original form with all numerical values being set to 0, and the answers JSON being reset to an empty JSON `{}`. This is useful after testing the multiplayer game as testing it will create game instances that should be processed. For example, if testing the in-game skip button, the ratio of skips may shift over 50%, removing the word from the single-player game. To tackle this, all games and answers that have been played in testing are manually removed, and then the `resetValuesForTesting()` function is run. The service is then able to regenerate all of the values and JSON objects upon its next iteration.

4.5 Single-player game

As stated Section 3.5.2, there are only minor differences between the single-player and multiplayer games. The major differences in the code are the lack of websockets, reduced complexity and lack of player numbers.

4.6 Website

4.6.1 TOML file

The only code that is exclusively for getting the website live is the file 'netlify.toml' [fig 2.5.1]. This contains 2 command variables that affect how the frontend is built and where in the directory it is to be built from.

4.6.2 Issue

An issue that was common to all parts of the site was that a lot of the *element-ui* components would not perform well on a mobile device. There were also some issues with websockets when running the website through the iPhone safari browser. As there was limited time for this project, spending valuable time trying to fix this problem would have been a mistake, and so instead the decision was made to stop support entirely for mobile devices and have the website only available to non-mobile platforms. This is done using the `mobileCheck()` function from the file 'mobileCheck.js'. All Vue components import this function and execute it upon creation of the page [fig 2.5.2]. If the function detects that the site is being viewed on a mobile device, the user is redirected to an information screen informing the user that they cannot access the site on their current device [fig 2.5.3].

5. Results and evaluation

5.1 System working as (or not as) intended

The results of this project can be found in files 'synonyms.csv', 'antonyms.csv' and 'hypernyms.csv'. These are exact copies of the corresponding tables in the database. The other database tables are irrelevant to the results or contain user information, and are therefore not included.

5.1.1 Generate metadata

The main goal of this project was to create a system that would use human computation to generate difficult metadata - in this case synonyms, antonyms and hypernyms. To this extent the project has been a success, as seen in the game mode tables of the database. At the time of writing this, 44 multiplayer games have been played by 16 players [fig 3.1.4 and fig 3.1.5], giving the result processing service enough data to let several words from each game mode pass the criteria for being available in the single-player game. For example, in the synonyms table the words "cold", "house", "eat" and "finger" all have at least 5 matches, 2 unique matches, and a skip rate of less than 50% [fig 3.1.6]. This shows that despite the criteria for words available for single-player being relatively low (at present), the system works and can be self sufficient in providing words for the single-player game. This is important due to the choice not to use the Wordnet API, and in my opinion presents what will eventually be a superior alternative once enough data has been collected. The game also helps to generate the strength of the association between words with the frequency of a match. For example, the word "cold" has had 6 matches with the synonym "freezing" and 2 matches with the synonym "chilly" [fig 3.1.7]. I have not yet developed an algorithm to determine the strengths of the relations, but going just by frequency one could say that the word "freezing" is a stronger synonym than the word "chilly".

5.1.2 Fun and appealing

A secondary goal of the project was to make the game both fun and appealing to play. The game itself is not a highly appealing concept for many to repeatedly play as it offers only a limited experience. On average, users played 2-3 games each, trying out each game mode once and then not returning to play again. There are several reasons why this could be happening. The first is that the game does not incite competitiveness. As both players are working together to find matches for words, there is no competition between the users, and therefore the sense of achievement at completing the game is merely against the system and not against the other player. The second reason is that the act of finding these word associations is not an engaging or exciting task to most players. With the rapid development of mobile and console games in the 21st century, humans are becoming increasingly used to having fun and unique games at their fingertips. The game developed in this project is comparatively dull and uninteresting to the majority of users. However, if the game were to be used as a tool for specific groups of people it may have more success. The multiplayer and single-player game could be used by school children,

particularly those learning the basics of English in primary school. The games could also be used by those who are studying English as a foreign language and wish to test or improve their skills. The single-player game could be used as a tool by speech and language therapists to help patients who have difficulty with word associations, for an example an elderly patient who has had a stroke and is attempting to relearn words that they have forgotten. In these cases the game functions as an educational tool with the primary focus of teaching and testing rather than being engaging and exciting.

5.1.3 Easy to use and available to all

Another secondary goal of the project was to make the tool easy to use and available to all. As per the brief, the game requires a user to login before they can play the game. This makes the tool considerably less easy to use as the user must spend anywhere for 2 to 10 minutes setting up their account (depending largely on the ease of use of their email provider - an external factor which the project cannot influence). This presents the issue that during the sign up or login process a user may lose interest in playing the game. It also makes the game hard to access for children who are unlikely to have an email address of their own, and if playing at school in an English lesson would not have a parent available to sign up for them. It also makes the game harder to play for those who are less technologically capable such as the elderly, who may find it difficult to complete the sign up process. The game itself however is fairly straightforward and easy to play - the most important information such as the word to find answers for and the time remaining are the largest items on the screen, and there are only 4 options available to the user (input, submit, skip and quit). Regardless, a user's first game will likely be of poor quality as they will have to become accustomed to the interface. In terms of accessibility the project has been a success, with the website running indefinitely at the address (<https://werdz.fun>). The initial plan was to run the servers and database through a Raspberry Pi, however this would have been unreliable (higher risk of the server going down from a power cut), slow (latency would depend on the location of the user relative to the Pi) and unscalable (unless multiple Pi's were to be run in parallel). It would also have over-complicated the project having to learn how to configure a Raspberry Pi and setup a server, leaving less time to work on the game and the login system.

5.2 Comprehensible results

5.2.1 Matches

The main results of the project are stored in the game mode tables. In the multiplayer game there were a total of 70 matches in the synonym game mode with 34 unique answers, 80 matches in the antonym game mode with 35 unique answers, and 34 matches in the hypernym game mode with 19 unique answers. Each game mode currently has 21 words available for the multiplayer game, however due to the words being chosen at random for each game there is an imbalance in the number of games each word has featured in. For example, the word "cold" has occurred in 13 synonym games, but the word "tree" has only occurred once [fig 3.1.8]. This

means it is hard to give an overall success rate, however despite this it is still obvious that the results show a multitude of word associations have been generated.

5.2.2 Game modes

The synonym game mode was played 20 times, with 70 matches, 11 skips and 59 incompletions [fig 3.1.17, fig 3.1.18, fig 3.1.19 and fig 3.1.20]. This means that the match rate of words has been exactly 50%. The ratio is likely to be lower as a word will usually have multiple obvious synonyms, and so the players may not be able to find one and either have to skip the word, or spend so much time on it that they run out of time and are unable to complete the game. This explains why the number of incompletions is so high. So far this is just a theory, however this could be tested by also recording the length of time spent on each round of a game.

The antonym game mode was played 14 times, with 80 matches, 6 skips and 5 incompletions [fig 3.1.9, fig 3.1.10, fig 3.1.11 and fig 3.1.12]. This is a much better ratio than synonyms with an 88% match rate. This is likely because antonyms are easier to find due there usually being just one obvious choice. For example, the word “light” could have several antonyms such as “gloomy”, “dim” and “overcast”, but the obvious antonym that most users would immediately enter is “dark”. Therefore it is much more likely that a match will be found in a short period of time.

The hypernym game mode was played 9 times, with 34 matches, 5 skips and 16 incompletions [fig 3.1.13, fig 3.1.14, fig 3.1.15 and fig 3.1.16]. This gives a match rate of 62% - not as low as synonyms, but also not as high as antonyms. Hypernyms are similar to antonyms in that there will often be one obvious choice, however the thought process of trying to classify a word is more complicated than the thought process of finding the opposite of a word. Therefore finding hypernyms of a word will usually take longer, leading to a relatively high number of incompletions due to running out of time.

5.3 Confidence in results

The results so far have been generated by friends and family members, however if the game were to be released to the public there would be the opportunity for users to exploit the system if they discover how it works. In the multiplayer game, so long as both players type in the same answer then it will be matched and become a synonym/antonym/hypernym for the word, regardless if it actually has that association. This means that 2 players could type in any answer they like as long as it is the same answer. This would not only make the game redundant, but would generate incorrect data. It would also corrupt the single-player game as the false answer that was entered could then potentially be used as an answer after the result processing server has analysed it. There are measures that could be undertaken to possibly prevent the results of cheating players leaking into the overall results and single-player game, however it is hard to be foolproof (see Section 6.1).

5.4 Testing

5.4.1 Code testing

Due to the large scope of this project, the decision was made to forsake unit testing in the interest of developing the project as much as possible in the allocated time span. In most parts of the code this was acceptable, however in hindsight testing would have been very useful for the more complicated algorithms such as those run by the matchmaking and result processing services. Both of these files use the unconventional method of creating a large string and using it to execute one SQL query rather than several. Because of this the code is both difficult to read and understand, so unit testing would have been appropriate. The testing of JavaScript files could be done using *chai*⁴, a testing library that is run through the Node.js environment. A file would be created with the same name as the file that is being tested, but with the extension *.test.js* (e.g. 'matchmaking.js' would be tested with the file 'matchmaking.test.js'). The original file's functions would then be imported into the test file. Each function would have an input JSON and expected output JSON defined for it in the test. The input would then be fed into the function, and the output would be compared to the expected output. When the test file is run in the Node environment, the *chai* package is able to identify which parts of the output JSON are different from the expected output and highlight them in the terminal. This would have been particularly useful for developers wishing to either continue the project, or salvage the matchmaking/result processing services for use in another project.

Testing of the frontend is much less formal when compared to the backend. When a change is made to a Vue component, the development build of the frontend does not need to be re-run, and the build is able to automatically integrate the changes. In future an official testing environment could be used such as Puppeteer⁵ - a tool for testing individual components and processes in Google Chrome.

5.4.2 Application testing

To test the functionality of the system and to gather data, the system was tested by 25 users, with 16 of them playing the game [fig 3.1.5]. The games were not done under controlled conditions as the priority at the time was to generate enough data for the development of the single-player game. The participants were instructed to make contact with any bugs, issues or general opinions on the game and login system. This feedback was then used to develop the user-friendliness of the game. For example, the criteria for a new password was originally a password that had been scored at least 3/5 by the *password-strength-utility* package. Several users complained that this was too strict, disallowing passwords that other sites would usually accept. Therefore the criteria was changed from a minimum of 3/5 to a minimum of 2/5.

The data generated by the game (the word associations) is difficult to test from a programming aspect, otherwise there would be no need for this project. As there are

⁴ <https://www.npmjs.com/package/chai>

⁵ <https://github.com/GoogleChrome/puppeteer>

currently only 21 words per game mode, there is still a small enough results pool that they can be checked manually by a human.

5.5 Evaluate methodology and programming languages

5.5.1 Database

The decision to use the relational database language MySQL over a non-relational database language such as MongoDB was a success. MongoDB stores data in JSON-like documents which may have been useful in some cases due to the backend of the system being written in JavaScript. It would have also allowed easier and more appropriate storage of JSON objects such as the array of objects in the words attribute of the `multiplayer_games` table. However most of the tables have a clear schema that is suited better to a relational database. The option to have both a relational and a non-relational database is not valid for 2 reasons. Firstly it would have over complicated the project at an early stage, taking time away from developing the system to learn how to set up and use a non-relational database such as MongoDB. Secondly it would mean that some data fetches that could be done in one MySQL query would now need to be done in two queries - one to each database (for example having to fetch a game from the multiplayer games in a MongoDB database, and then use the player's id to get their email addresses from the MySQL database).

5.5.2 Node

The decision to use Node.js as the backend environment was also a success. I already had some knowledge of JavaScript and was familiar with some of the essential packages such as *lodash*, allowing me to quickly get to work and not waste time learning a new language.

5.5.3 Vue

Using the Vue.js frontend framework was particularly useful due to JavaScript being an asynchronous programming language. This means that the system can run parts of the code separately from the primary application thread, which is particularly useful for functions that deal with time such as the timer component. It allows a function to run in parallel to the main process, and once it has either completed or failed its task, will notify the main thread and return any data that it has promised.

6. Future work

6.1 Unrealised ideas

One of the key issues with the game was its lack of competitiveness. In the multiplayer game, the players are competing against the system. However a more competitive game where the players compete against each other may be more appealing. This could be similar to the single-player game, however a round is won by the first player to enter a valid synonym/antonym/hypernym. The player with the most rounds won at the end of the game would be the winner. There could also be a 'pointless' variety of the game, where the aim is to enter the most obscure answer that they can think of. For example, if the word was "off", then the antonym "on" would be a low scoring answer, whereas the word "activated" would be a high scoring answer. There could also be more word association game modes such as 'hyponyms' - the opposite of hypernyms (i.e. if a hypernym is a more general meaning, then a hypernym is a more specific example, e.g. furniture → chair).

Another incentive for users to play the game could be a leaderboard system. This would display all players in a table with the rank of a player being determined by how many games they have played and how many word associations they have made. This could be accompanied by user names for an account, and the option to add an image to a user's account as a 'profile picture'.

To make the game more accessible, a change to the system could be made so that users are not required to login or have an account to play the game. This would make the game much more available, but would also increase the likelihood of players creating bots to play the game. The login system could still exist as an option, and only users with an account would be displayed on a leaderboard. There could also be other perks to creating an account such as being able to rematch the last player that a user and played with, or choose to play with a friend rather than being matched with another random player.

When a user first signs up, a tutorial telling them how to play the game could help first time players. This could be an embedded video, or an interactive walk through of the single-player game. Once completed, the multiplayer game would become available to the user. To verify that a player is genuine and not a bot, a user could be forced to play an arbitrary number of single-player games before they can access the multiplayer game. If the player is unable to complete any single-player games then they are either a bot, or they are a user who is not proficient enough in the English language to be able to play the multiplayer game successfully.

Currently the result processing service has no way to evaluate whether an answer is actually a valid synonym, antonym or hypernym. If user's are abusing the system by coordinating an answer, a few fail-safes could be implemented:

- If a player enters the same answer for multiple rounds in a game, their results are not processed.

- If a game's matched words are repeated or below a certain length (e.g. always 1 character), they are not processed.
- An answer is not processed until it has been entered as a synonym, antonym or hypernym by several different users.
- Create a blacklist of inappropriate or offensive words that if entered as an answer cause the rest of that user's answers to not be processed, and for that user to be potentially banned.
- Further develop the matchmaking service to not match users who are playing the game from the same IP address.

The current criteria for a word to be included in the single-player game is relatively low. Once more results have been obtained from the multiplayer game, the criteria could be made stricter, for example:

- Increasing the minimum number of matches for a word from 5 to 20.
- Decreasing the maximum skip rate of a word from 50% to 10%.
- Increasing the minimum number of unique matches for a word from 2 to 5.

Another possible future development could be to separate the databases into a relational database for login system and queueing, and a non-relational database for the games and answers. Once the project has been further developed, there may be scope to implement such a separation. This may be more efficient for storing tables such as `multitplayer_games` and `multiplayer_answers` due to them already containing stringified JSON objects that would be better stored in a non-relational database.

At its current stage, the game is ready for a controlled testing phase. This would involve taking a select group of individuals and having them play multiple games with each other. This would serve 2 purposes:

- It would help to discover any remaining bugs in the game and login system.
- It would generate more data, allowing the single-player game to be further developed based on the results.

Currently there have been 42 multiplayer games, but I estimate that I would need a minimum of 200 games played by at least 30 users before the single-player game could be tested.

Despite the game being available to all, the data that the game has generated is still only accessible from the database. To make the results public, this data could be published in several different ways. The first option could be in a public database that is updated from the original database. The original database would not be made public for security as it also contains user details such as email addresses and names. Another form in which the data could be accessed is through an API that allows developers to get data for a word in a JSON format. The data that would be returned could look something like this:

```

{
  word: "cold",
  synonyms: {
    freezing: 6,
    chilly: 2,
    subzero: 1,
  },
  antonyms: {
    hot: 2,
    warm: 1,
  },
}

```

From a business perspective, the most obvious goal is to attract users to sign up and play the game. For this to happen, the first step would be making the public aware of the game's existence. This could be done through advertising, for example by creating a Google Ads campaign to have the game show up as an advert on appropriate Google searches. To cover the costs of a Google Ads campaign, as well as to cover the costs of the domain name, servers and database, the game could contain advertising space for other companies and organisations who would pay an advertising fee.

6.2 Starting point for continuation of work

For a developer to continue the project, they would have 2 options - to use the source code to set up their own website with their own domain, servers and database; or to use the ones that have already been set up. If the latter option were to be chosen, the credentials of these tools would have to be transferred as they are not stored in the GitHub repository. The developer would also have to create a new `.env` file to store some of these credentials and environment variables in the backend of their local repository [fig 4.1.8].

If a developer was to start with just the source code, once they have created a database they can then use the command line function `npm run create_db_tables`. This executes the code in the file 'create_db_tables.js', which creates all of the tables that the database needs for the system to function correctly.

To keep the code consistent, I have employed several coding guidelines for the sake of consistency:

- Included in the source code are several ESLint and Prettier files [fig 4.1.7]. These can be used in the text editor Visual Studio Code (VS Code) along with the corresponding VS Code extensions to format the code to a consistent format such as tabs being 4 spaces.
- Using snake case for variables, e.g. `user_id` rather than `userID`.
- Using camel case for functions, e.g. `getSnacks()` rather than `get_snacks()`.

- Using uppercase for global variables, e.g. `PORT = 8080` rather than `port = 8080`.
- Using ES6 JavaScript arrow functions, e.g. `const test = () => {return}` rather than `const test = function() {return}`.

There are a few inconsistencies within the architecture of the directory. For example, the Vue component for the multiplayer game is named 'Game.vue', but the Vue component for the single-player game is named 'SinglePlayerGame.vue' [fig 4.1.6]. A future piece of work could be to tidy up the directories and eliminate any inconsistencies in file names. In this example, I would rename the file 'Game.vue' to 'MultiPlayerGame.vue'. There is also a relatively large inconsistency in how the skipping of a word in a game is referenced. In some places such the file 'Game.vue', the button is referred to as the 'skip word button'. However in the database, the skipping of a word is referred to as 'passing', for example in the `multiplayer_answers` table, the attribute to determine whether a word was skipped is labeled 'passed'. This should be noted by any developers using the project.

7. Conclusion

The main aim of this project was to create a data generation tool and package it as a game. The data to be collected was synonyms, antonyms and hypernyms of words. To make the tool a success, the secondary goals of the project were to make the game fun and easy to play, while being easily accessible and available to all. I believe I have achieved these goals to the best of my ability, however there are limitations to how appealing a word association game can be. At its current state I would be willing to start a controlled testing phase of the game.

The results of the multiplayer game have shown that the system functions as a tool to generate synonyms, antonyms and hypernyms of words, and the relative strength of these associations. It has also shown that all 3 of these game modes perform differently and can be treated as such.

8. Reflection on learning

Over this project I have developed many technical skills that I lacked prior to this semester. These include the knowledge gained on the subject of JavaScript, experience using the frontend framework Vue.js, and a better understanding of relational databases. I have also developed a much larger understanding of the ways in which web applications function, now being knowledgeable in servers, websockets, POST requests and many other parts of the web which until now were just terms I had heard thrown around by colleagues and lecturers.

In terms for personal skills, I feel like my ability to manage a project has transcended throughout this semester. I am now far more aware of the uses of GitHub for large projects such as this, and have a much greater appreciation for the development process. This project has taught me to plan ahead in whatever part of development I am working - to consider all eventualities before going anywhere near a keyboard, which in the long run saved time that would have been spent fixing bugs and optimising algorithms. In future I would create a much more extensive plan than the 2 page initial plan that was first created, setting out all available resources before deciding which to use in each location.

1.0.0 - Website screenshots:

The screenshot shows a sign-up form with a header containing a "Log in" link and a blue "Register" button. The form fields are: Forename (placeholder: e.g. Bob), Surname (placeholder: e.g. Bob), Email (placeholder: e.g. bob@example.co.uk), New password (placeholder: Enter New Password), and Confirm password (placeholder: Confirm New Password). Below the fields is a checkbox for terms and conditions, which is unchecked. At the bottom are "Register" and "Reset" buttons.

[Fig 1.1.1] Sign-up page without input

The screenshot shows the same sign-up form as Fig 1.1.1, but with input. The "Log in" link is now a button. The "Register" button is highlighted in blue. The form fields contain: Forename: William, Surname: Cooter, Email: cooterw@cardiff.ac.uk, New password: a masked password, and Confirm password: a masked password. The terms and conditions checkbox is now checked. The "Register" button is highlighted in blue, and the "Reset" button is greyed out.

[Fig 1.1.2] Sign-up page with input

Hello Will!

Choose game mode

Singleplayer	Multiplayer
Play by yourself	Match and play with another player

[Fig 1.1.3] Home page

Forgotten Password

Email:

[Fig 1.1.4] Forgotten password page

Reset Password

New password:

Confirm password:

[Fig 1.1.5] Reset password page

Account Settings

Update your email address

Current password:

New email address:

Change your password

Current password:

New password:

Confirm password:

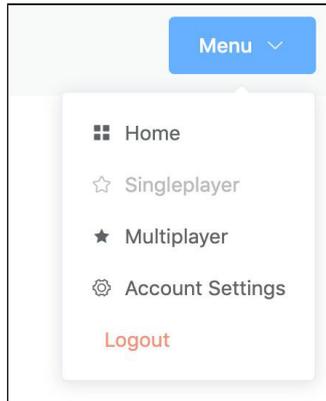
[Fig 1.1.6] Account settings page

Delete Account

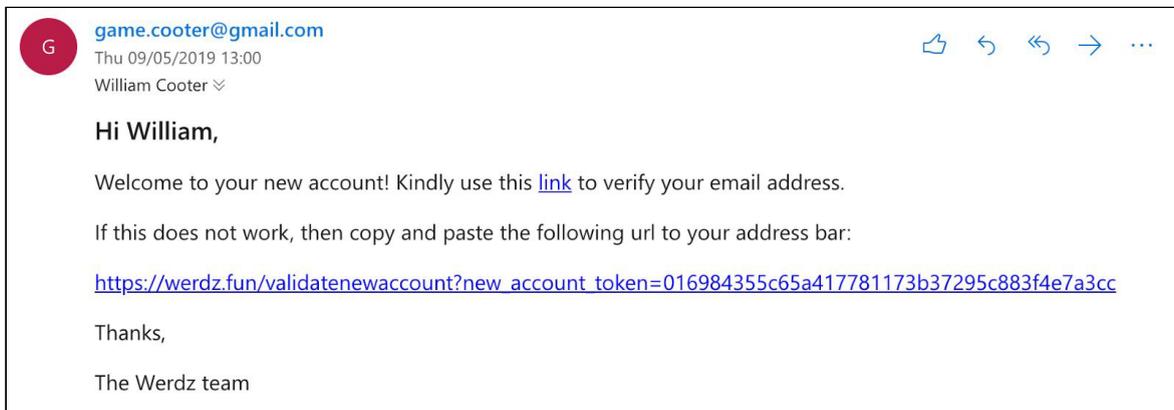
Email:

Password:

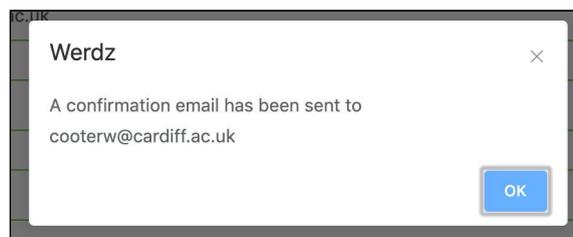
[Fig 1.1.7] Delete account page



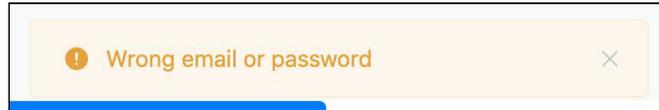
[Fig 1.1.8] Website menu



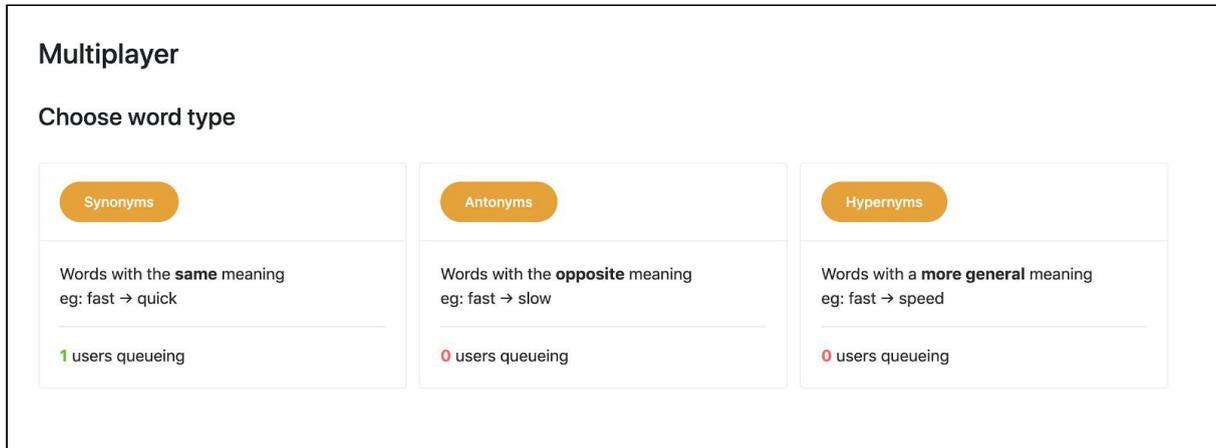
[Fig 1.1.9] Sign-up confirmation email



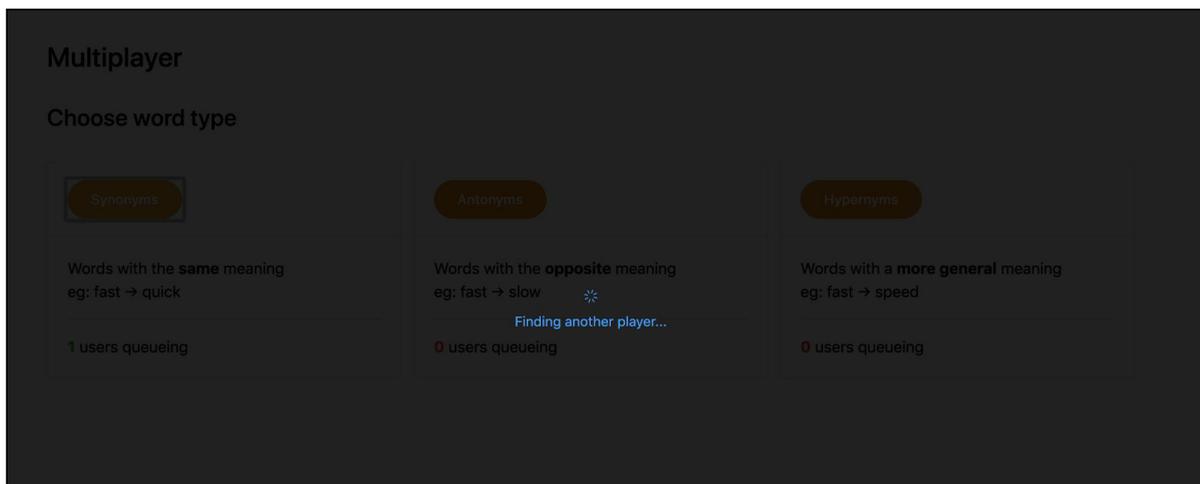
[Fig 1.1.10] Confirmation email alert



[Fig 1.1.11] Login failure message



[Fig 1.2.1] Multiplayer game menu



[Fig 1.2.2] Waiting mode

Game starts in
3
seconds

[Fig 1.2.3] Game countdown

The screenshot shows a game interface for the word "Book". At the top left, a red timer indicates "7 sec" with a progress bar. Below the word "Book", there is a text input field with the placeholder "Please input a synonym..." and a right-pointing arrow button. Underneath the input field is the definition: "Something one might take out from a library" and a "Skip word" button. On the right side, a panel titled "Synonym 3 of 10" displays "Words with the same meaning eg: fast → quick" and "The other player has submitted 1 answers". Below this panel are two status indicators: "Matched: 1" (green) and "Passed: 1" (orange). At the bottom right, there is an "Answers" section with the word "paper" entered in a text field and a "Quit" button.

[Fig 1.2.4] Game page

The screenshot shows the word "Clean" at the top. Below it is a text input field containing the text "CLEAN Dirty answer|". Below the input field, the text "Not dirty" is displayed.

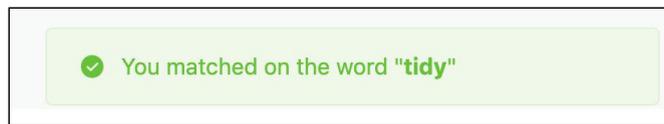
[Fig 1.2.5] Answers before submission



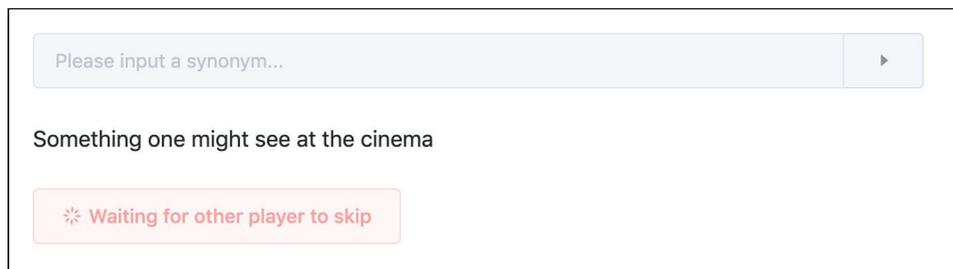
Answers

answer

[Fig 1.2.6] Answers after submission



[Fig 1.2.7] Match notification

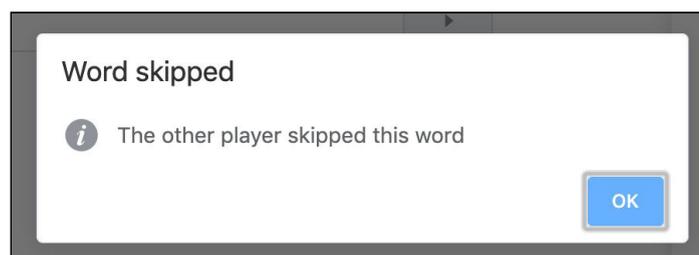


Please input a synonym... ▶

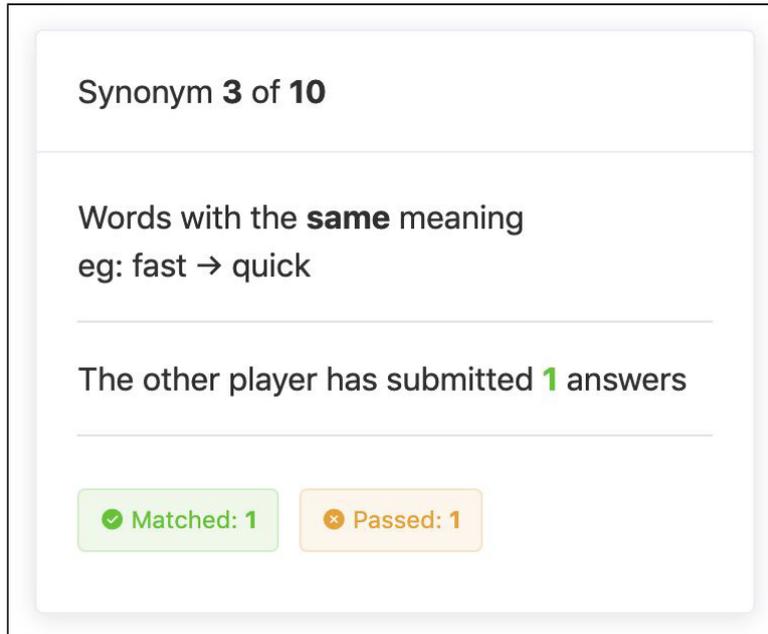
Something one might see at the cinema

* Waiting for other player to skip

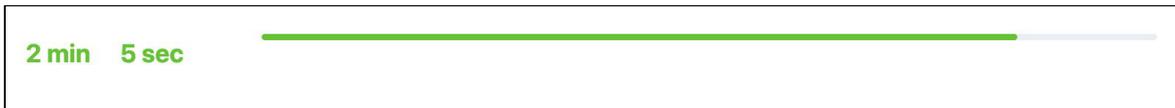
[Fig 1.2.8] Disabled input and skip button



[Fig 1.2.9] Skip alert



[Fig 1.2.10] Multiplayer game information box



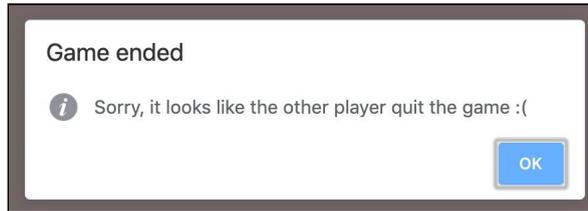
[Fig 1.2.11] Progress bar green



[Fig 1.2.12] Progress bar amber



[Fig 1.2.13] Progress bar red



[Fig 1.2.14] Other player quit alert

Results:
✔ Matched: 1
⚠ Passed: 1
❌ Uncompleted: 8

Word	Match	Your answers	Opponent's answers
✔ Clean	tidy	answer, tidy	tidy
⚠ Film			
❌ Book		paper	papaer

[Fig 1.2.15] Multiplayer game results table

Antonym 4 of 8

Words with the **opposite** meaning
eg: fast → slow

✔ Matched: 2
⚠ Passed: 1

[Fig 1.3.1] Single-player game information box



[Fig 1.3.2] Single-player game menu

2.0.0 - Code screenshots:

```
const { DB_HOST, DB_USER, DB_PASSWORD, DB_DATABASE } = require('../config')
```

[Fig 2.1.1] The database credentials being imported from the `.env` file into the file `'db.js'`

```
return new Promise(async (resolve, reject) => {
```

[Fig 2.1.2] A JavaScript promise in the file `'db.js'`

```
const db = require('./db')
```

[Fig 2.1.3] The database query function being imported into the file `'server.js'`

```
import { apiRequest } from '../api/auth'
```

[Fig 2.1.4] The axios API request function being imported into a Vue component

```
let url = ''
if (process.env.NODE_ENV === 'development') {
  url = `http://localhost:8080/${endpoint}`
} else if (process.env.NODE_ENV === 'production') {
  url = `https://api.werdz.fun/${endpoint}`
}
```

[Fig 2.1.5] The URL of a POST request being determined in the file `'auth.js'`

```
const { EMAIL_SERVICE, EMAIL_USER, EMAIL_PASS, NODE_ENV } = require('../config')
```

[Fig 2.1.6] The email credentials being imported from the `.env` file into the file `mail.js`

```
const determineNodeEnv = () => {  
  if (NODE_ENV === 'development') {  
    return 'http://localhost:3000'  
  } else if (NODE_ENV === 'production') {  
    return 'https://werdz.fun'  
  }  
}
```

[Fig 2.1.7] The URL of a the API being determined in the file `'mail.js'`

```
await fs.readFile(  
  __dirname + '/api/templates/change-email-confirmation.html',  
  'utf8',  
  async (err, data) => {
```

[Fig 2.1.8] An email template being read from the file `'mail.js'`

```

const passwordStrength = (rule, value, callback) => {
  if (this.password_score < 2) {
    callback(new Error('Not strong enough'))
  } else {
    callback()
  }
}

```

[Fig 2.1.9] The password strength function in the file 'SignUp.vue'

```

const confirmPassword = (rule, value, callback) => {
  if (value !== this.register_model.new_password) {
    callback(new Error('Passwords must match'))
  } else {
    callback()
  }
}

```

[Fig 2.1.10] The confirm password function in the file 'SignUp.vue'

```

const isAString = (rule, value, callback) => {
  const re = /^[a-z]+\s*[a-z]+$/
  if (re.test(String(value).toLowerCase())) {
    callback()
  } else {
    callback(new Error('Names can only contain letters and spaces'))
  }
}

```

[Fig 2.1.11] The is a string function in the file 'SignUp.vue'

* Forename:

Please input your first name

[Fig 2.1.12] The validation message for no input into the forename field

* Forename:

Names can only contain letters and spaces

[Fig 2.1.13] The validation message for bad input into the forename field

* Forename:

[Fig 2.1.14] The validation message for correct input into the forename field

```
redirect(location) {  
  |   this.$router.push({ name: location })  
  },  
}
```

[Fig 2.1.15] The redirect function found in most Vue files

```
<!-- DELETE ACCOUNT -->
<el-button
  type="danger"
  plain
  style="float:right;"
  v-on:click="redirect('DeleteAccount')"
>Delete Account</el-button
>
```

[Fig 2.1.16] An example of the redirect function in the HTML template

```
import SignUp from '@components/SignUp'
import Home from '@components/Home'

Vue.use(Router)

export default new Router({
  mode: 'history',
  routes: [
    {
      path: '/',
      name: 'SignUp',
      component: SignUp,
    },
    {
      path: '/home',
      name: 'Home',
      component: Home,
    },
  ],
})
```

[Fig 2.1.17] The Vue router

```

redirectInNewTab(location) {
  const routeData = this.$router.resolve({ name: location })
  window.open(routeData.href, '_blank')
},

```

[Fig 2.1.18] The redirect in new tab function in the file 'SignUp.vue'

```

<!-- TERMS AND CONDITIONS -->
<el-form-item>
  <el-checkbox v-model="register_model.terms_agreed"></el-checkbox>By
  checking this box you declare that you have read and agree to the
  <el-button
    type="text"
    style="color:#426cb9;"
    v-on:click="redirectInNewTab('TermsAndConditions')">
    >terms and conditions</el-button>
</el-form-item>

```

[Fig 2.1.19] The redirect in new tab function in the HTML template in the file 'SignUp.vue'

```

const hash = await bcrypt.hash(req.body.password, saltRounds)

```

[Fig 2.1.20] The hashing of a password in the file 'server.js'

```

const token_val = await crypto.randomBytes(20)
const token = token_val.toString('hex')

```

[Fig 2.1.21] The creation of a sign up token in the file 'server.js'

```
const authenticated = await bcrypt.compareSync(req.body.password, user.password)
```

[Fig 2.1.22] The authentication of a password in the file 'server.js'

```
const token = jwt.sign(payload, app.get('appSecret'), {  
  expiresIn: '24h',  
})
```

[Fig 2.1.23] The creation of a JSON web token in the file 'server.js'

```
const main = async () => {  
  await checkGames()  
  const users_exist = await checkMatches()  
  if (!users_exist && reset_time < 21000) {  
    reset_time += 3000  
  } else if (users_exist) {  
    reset_time = 2000  
  }  
  await setTimeout(() => {  
    main()  
  }, reset_time)  
}
```

[Fig 2.2.1] The main function with variable time intervals in the file 'matchmaking.js'

```
if (!user.last_heartbeat ||  
  moment(moment().utc().format('YYYY-MM-DD HH:mm:ss')).diff(user.last_heartbeat, 'seconds') > 5  
) {
```

[Fig 2.2.2] The file 'matchmaking.js' checking that a user's last heartbeat was no more than 5 seconds ago

```
const grouped_users = _.groupBy(alive_queued_users, 'game_mode')
```

[Fig 2.2.3] The grouping of users by their chosen game mode in the file 'matchmaking.js'

```
const temp3 = `(${grouped_users[key][i].user_id},  
  ${grouped_users[key][i + 1].user_id},  
  '${grouped_users[key][i].game_mode}',  
  '${token}',  
  '${moment().utc().format('YYYY-MM-DD HH:mm:ss')}}',  
  '${moment().utc().add(160, 'seconds').format('YYYY-MM-DD HH:mm:ss')}}',  
  '${words}')`,\n`
```

[Fig 2.2.4] The creation of a string to be used in an SQL statement in the file 'matchmaking.js'

```
for (let i = 0; i < game_modes.length; i++) {  
  const key = game_modes[i]  
  grouped_users[key] = _.sortBy(grouped_users[key], 'initialisation_date')  
  for (let i = 0; i < grouped_users[key].length - 1; i += 2) {
```

[Fig 2.2.5] The file 'matchmaking.js' looping through each game mode and then every other user

```
game.termination_date = moment(game.termination_date)  
  .add(1, 'hours')  
  .format('YYYY-MM-DD HH:mm:ss')
```

[Fig 2.3.1] The addition of 1 hour to the data retrieved from the database in the file 'server.js'

```
const current_word_index =
  answers.length - _.sortBy(_.filter(answers, { matched: 0, passed: 0 }), ['id']).length
```

[Fig 2.3.2] The current word index recovered from a partially completed game in the file 'server.js'

```
const matched_answers_count = _.filter(answers, { matched: 1 }).length
const passed_answers_count = _.filter(answers, { passed: 1 }).length
```

[Fig 2.3.3] The matched and passed counts recovered from a partially completed game in the file 'server.js'

```
const current_word_answers = []
JSON.parse(_.sortBy(answers, ['id'])[current_word_index].this_player_answers).forEach(
  ans => {
    current_word_answers.push({ answer: ans })
  }
)
```

[Fig 2.3.4] The already submitted answers recovered from a partially completed game in the file 'server.js'

```
const other_player_answer_count = JSON.parse(
  _.sortBy(answers, ['id'])[current_word_index].other_player_answers
).length
```

[Fig 2.3.5] The count of the other player's answers recovered from a partially completed game in the file 'server.js'

```

res.data.status ? (this.game = res.data.game) : this.$router.push({ name: 'Home' })

this.player_no = res.data.player_no
this.current_word_index = res.data.current_word_index
this.matched_count = res.data.matched_answers_count
this.passed_count = res.data.passed_answers_count
this.answers = res.data.current_word_answers
this.no_of_opponent_answers = res.data.other_player_answer_count
this.input_placeholder = res.data.input_placeholder

```

[Fig 2.3.6] The assigning of the game information in the file 'Game.vue'

```

<TimerMultiplayer
  v-bind:date="game.termination_date"
  v-bind:game_id="game.id"
  v-bind:token="token"
  style="float:center;"
  @start_game="startGame"
  @delay_game="delayGame"
></TimerMultiplayer>

```

[Fig 2.3.7] The HTML timer component in the file 'Game.vue', able to call the methods startGame and delayGame

```

startGame() {
  |   |   this.game_started = true
  },

```

[Fig 2.3.8] The start game function in the file 'Game.vue'

```

delayGame(time) {
  | |   this.time_until_start = time - 150
  },

```

[Fig 2.3.9] The delay game function in the file 'Game.vue'

```

watch: {
  |   actualTime() {
  |     |   this.compute()
  |     },
  },

```

[Fig 2.3.10] The watch section of the file 'TimerMultiplayer.vue'

```

incrementTime() {
  |   this.actualTime = moment()
  |     .utc()
  |     .format('X')

```

[Fig 2.3.11] The increment time function getting the actual time in a UTC format in the file 'TimerMultiplayer.vue'

```

getDifference() {
  |   return this.termination_date - this.actualTime
  },

async compute() {
  |   const duration = moment.duration(this.getDifference(), 'seconds')

```

[Fig 2.3.12] The get difference function in the file 'TimerMultiplayer.vue'

```
if (this.timer_is_going && current_time <= this.game_length) {  
  this.$emit('start_game')  
  this.time_has_started = true
```

[Fig 2.3.13] The emission of 'start_game' in the file 'TimerMultiplayer.vue'

```
} else if (this.timer_is_going && current_time > this.game_length) {  
  this.$emit('delay_game', current_time)
```

[Fig 2.3.14] The emission of 'delay_game' with the current time in the file 'TimerMultiplayer.vue'

```
if (!this.minutes && !this.seconds) {  
  if (this.timer_is_going) {  
    const data = { game_id: this.game_id }  
    await apiRequest('post', 'finishGameMulti', data)  
    this.$router.push({  
      name: 'GameResults',  
      query: { token: this.token },  
    })
```

[Fig 2.3.15] The file 'TimerMultiplayer.vue' redirecting the client to the game results page

```
const words = _.words(_.toLowerCase(this.input))
```

[Fig 2.3.16] The inputted answer being broken into an array of words in the file 'Game.vue'

```

words.forEach(word => {
  const x = { answer: word }
  const cond1 = !_.filter(this.answers, x).length
  const cond2 = word.length
  const cond3 = word !== _.toLower(this.game.words[this.current_word_index].word)
  const cond4 = !_.words(
    | | _.toLower(this.game.words[this.current_word_index].definition)
  ).includes(_.toLower(word))
  if (cond1 && cond2 && cond3 && cond4) {
    | | this.answers.push(x)
    | | data.answers.push(word)
  }
})

```

[Fig 2.3.17] The filtering of inputted words in the file 'Game.vue'

```

io.on('connection', socket => {
  const token = socket.request._query['token']

  if (token) {
    | socket.join(token)
  } else {
    | socket.join('queue')
  }
}

```

[Fig 2.3.18] The connection to either the 'queue' room or the room by the name of the token in the file 'server.js'

```

// THE SUBMITTING PLAYER'S PLAYER NUMBER
const this_player_no_answers = req.player_no === 1 ? 'p1_answers' : 'p2_answers'
// THE OTHER PLAYER'S PLAYER NUMBER
const other_player_no_answers = req.player_no === 1 ? 'p2_answers' : 'p1_answers'

```

[Fig 2.3.19] The player number strings being determined in the file 'server.js'

```

try {
  this_players_words = JSON.parse(answers.this_player)
  other_players_words = JSON.parse(answers.other_player)
} catch (e) {
  throw e
}

```

[Fig 2.3.20] The parsing of stringified arrays in the file 'server.js'

```

const matches = _.intersection(this_players_words, other_players_words)

```

[Fig 2.3.21] The intersections between both player's answer arrays in the file 'server.js'

```

io.in(req.game_token).emit('answerSubmitted', {
  // EMIT TO BOTH PLAYERS
  status: true,
  word: match.answer,
})

```

[Fig 2.3.22] The emission to the answer submitted socket if there was a match in the file 'server.js'

```

io.in(req.game_token).emit('answerSubmitted', {
  status: false,
  this_player_id: req.user_id,
  this_player_word_count: this_players_words.length,
  other_player_word_count: other_players_words.length,
})

```

[Fig 2.3.23] The emission to the answer submitted socket if there was not a match in the file 'server.js'

```

this.socket.on('answerSubmitted', async res => {

```

[Fig 2.3.24] The file 'Game.vue' listening to the answer submitted socket

```
this.$message({
  dangerouslyUseHTMLString: true,
  message: `You matched on the word "<strong>${res.word}</strong>"`,
  type: 'success',
})
```

[Fig 2.3.25] The element-ui alert telling the players that they have matched on a word in the file 'Game.vue'

```
if (this.current_word_index === this.game.words.length - 1) {
  this.$router.push({
    name: 'GameResults',
    query: { token: this.token },
  })
}
```

[Fig 2.3.26] The file 'Game.vue' redirecting to the results page if they previous match was on the final word

```
} else {
  this.matched_count++
  this.nextWord()
}
```

[Fig 2.3.27] The file 'Game.vue' incrementing the matched count and executing the next word function if the previous match was not the last word

```
this.socket.emit('skipWord', data)
```

[Fig 2.3.28] The file 'Game.vue' emitting through the skip word socket

```

this.$alert('The other player skipped this word', 'Word skipped', {
  confirmButtonText: 'OK',
  closeOnClickModal: false,
  showClose: false,
  type: 'info',
  callback: () => {
    this.socket.emit('confirmSkip', data)
    this.passed_count++
    this.nextWord()
  },
})

```

[Fig 2.3.29] The element-ui alert received by a player if the other player skipped a word in the file 'Game.vue'

```

socket.to(req.game_token).emit('otherPlayerSkipped', {
  // EMIT ONLY TO OTHER PLAYER
  user_id: req.user_id,
  status: true,
})

```

[Fig 2.3.30] The emission to the other player skipped socket in the file 'server.js'

```

this.socket.on('otherPlayerConfirmedSkipped', async res => {
  // WHEN THE OTHER PLAYER CONFIRMS THE SKIP
  if (res.status && res.user_id !== this.user.user_id) {
    this.passed_count++
    this.skip_button_loading = false
    this.submit_disabled = false
    this.skip_button_text = 'Skip word'
    this.nextWord()
  }
})

```

[Fig 2.3.31] The code executed by the file 'Game.vue' upon receiving an emission from the other player confirmed skip socket

```

this.socket.emit('quitGame', data)
this.$router.push({
  name: 'GameResults',
  query: { token: this.token },
})

```

[Fig 2.3.32] The file 'Game.vue' emitting to the quit game socket and redirecting to the results page

```

socket.to(req.game_token).emit('otherPlayerQuit', {
  // EMIT ONLY TO OTHER PLAYER
  user_id: req.user_id,
  status: true,
})

```

[Fig 2.3.33] The emission to the other player quit socket from the file 'server.js'

```

this.socket.on('otherPlayerQuit', async res => {
  // WHEN THE OTHER PLAYER QUILTS THE GAME
  if (res.status && res.user_id !== this.user.user_id) {
    this.$router.push({
      name: 'GameResults',
      query: { token: this.token },
    })
    this.$alert(
      'Sorry, it looks like the other player quit the game :(',
      'Game ended',
      {
        confirmButtonText: 'OK',
        closeOnClickModal: false,
        showClose: false,
        type: 'info',
      }
    )
  }
})

```

[Fig 2.3.34] The code executed by the file 'Game.vue' upon receiving an emission to the other player quit socket.

```

<el-table-column prop="matched" width="40">
  <template slot-scope="scope">
    <i
      v-if="scope.row.matched"
      class="el-icon-success"
      style="color:#67C23A;"
    ></i>
    <i
      v-if="scope.row.passed"
      class="el-icon-warning"
      style="color:#E6A23C;"
    ></i>
    <i
      v-if="scope.row.uncompleted"
      class="el-icon-error"
      style="color:#F56C6C;"
    ></i>
  </template>
</el-table-column>

```

[Fig 2.3.35] The HTML code for the table in the file 'GameResults.vue' determining the icon at the start of the row

```

<style>
.el-table .danger-row {
  background: #fde6e6;
}

.el-table .warning-row {
  background: #fdf5e6;
}

.el-table .success-row {
  background: #f0f9eb;
}
</style>

```

[Fig 2.3.36] The CSS of the colours for the background of the table in the file 'GameResults.vue'

```

const reset_time = 1000 * 10 // 10 seconds

const main = async () => {
  checkWords()
  await setTimeout(() => {
    main()
  }, reset_time)
}

```

[Fig 2.4.1] The main function of the file 'result_processing.js'

```

const grouped_answers = _.groupBy(answers, 'game_mode')

```

[Fig 2.4.2] The grouping of answers by their game mode in the file 'result_processing.js'

```

words[answer.word] = {
  matched_words: [answer.matched_word],
  matched: answer.matched,
  passed: answer.passed,
  uncompleted: answer.uncompleted,
}

```

[Fig 2.4.3] The object created for each word in the file 'result_processing.js'

```

let words_as_array = ``
Object.keys(words).forEach(word => {
  words_as_array += `${word}`, `
  // remove nulls from the list of answers to a word
  words[word].matched_words = _.compact(words[word].matched_words)
})
words_as_array = words_as_array.slice(0, -2) + ``

```

[Fig 2.4.4] The string created to be used in an SQL statement in the file 'result_processing.js'

```

// get the previous answers to a word in it's game mode
const previous_answers = await db.qry(
  `SELECT word, answers, multiplayer_occurrences, multiplayer_matches, multiplayer_passes
  FROM ${game_mode_type.table}
  WHERE word IN ${words_as_array}`
)

```

[Fig 2.4.5] The SQL statement that receives a string in the file 'result_processing.js'

```

const cond1 = word.multiplayer_matches > 4
const cond2 =
  word.multiplayer_passes / (word.multiplayer_passes + word.multiplayer_matches) < 0.5
const cond3 = _.keys(JSON.parse(word.answers)).length > 1

```

[Fig 2.4.6] The conditions for a word to be made available in the single-player game mode in the file 'result_processing.js'

```

[build]
  base = "front-end/"

[[redirects]]
  from = "/*"
  to = "/index.html"
  status = 200

```

[Fig 2.5.1] The contents of the file 'netlify.toml'

```

created() {
  mobileCheck() ? this.$router.push({ name: 'MobileRedirect' }) : console.log()
}

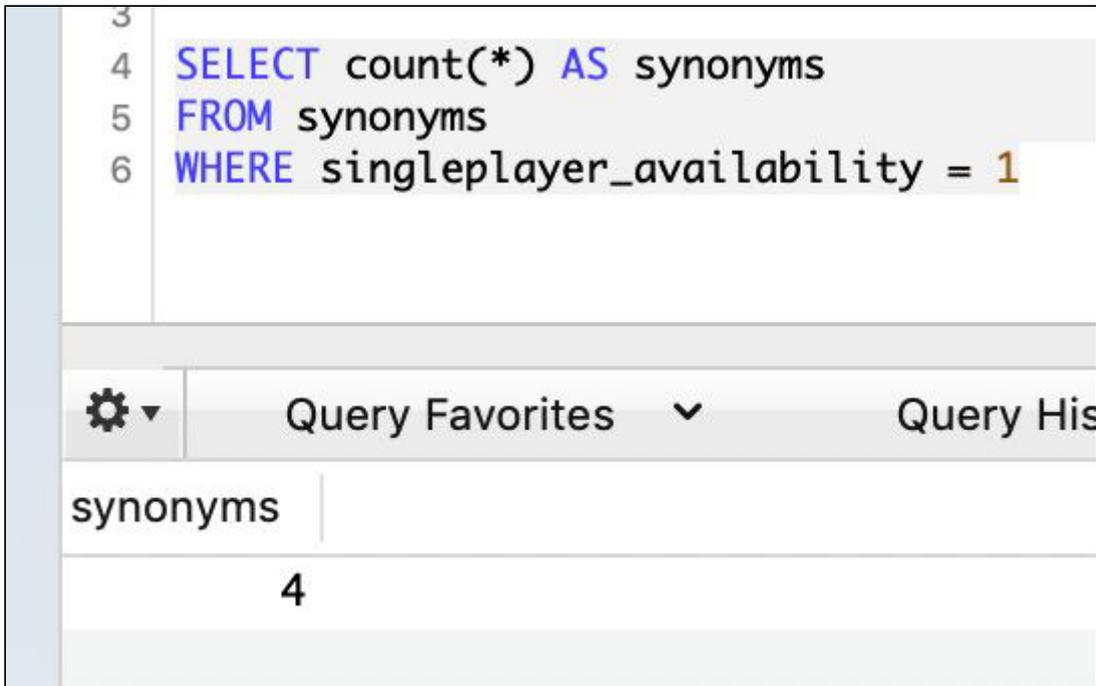
```

[Fig 2.5.2] The use of the mobile checking function found in all Vue components.

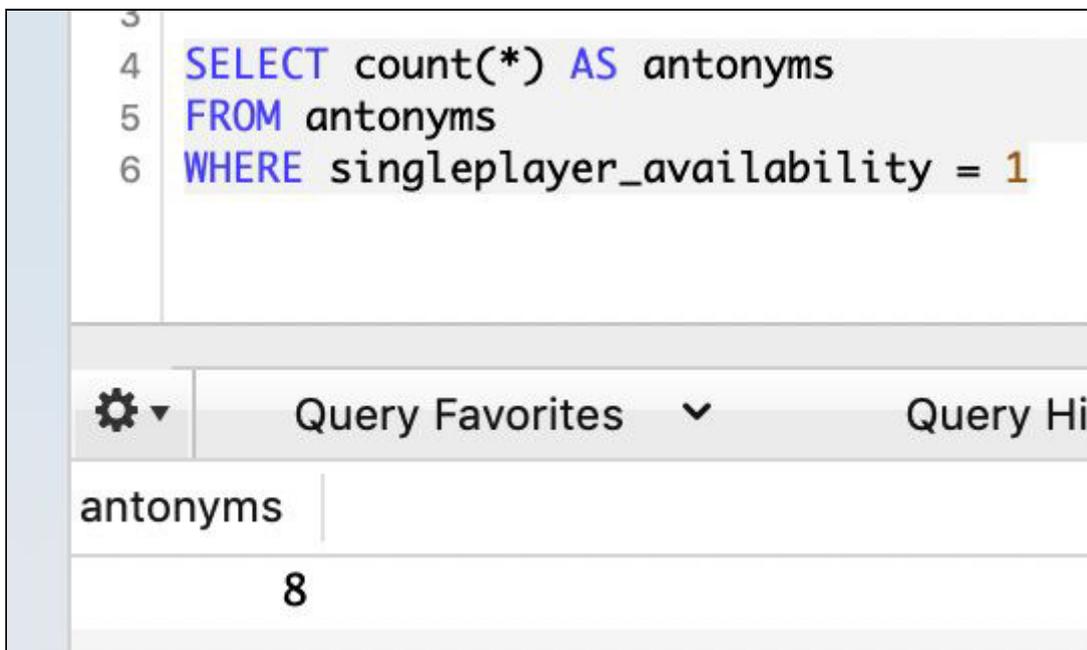
```
export const mobileCheck = () => {  
  if (  
    typeof window.orientation !== 'undefined' ||  
    navigator.userAgent.indexOf('IEMobile') !== -1  
  ) {  
    return true  
  } else {  
    return false  
  }  
}
```

[Fig 2.5.3] The mobile check function in the file 'mobileCheck.js'

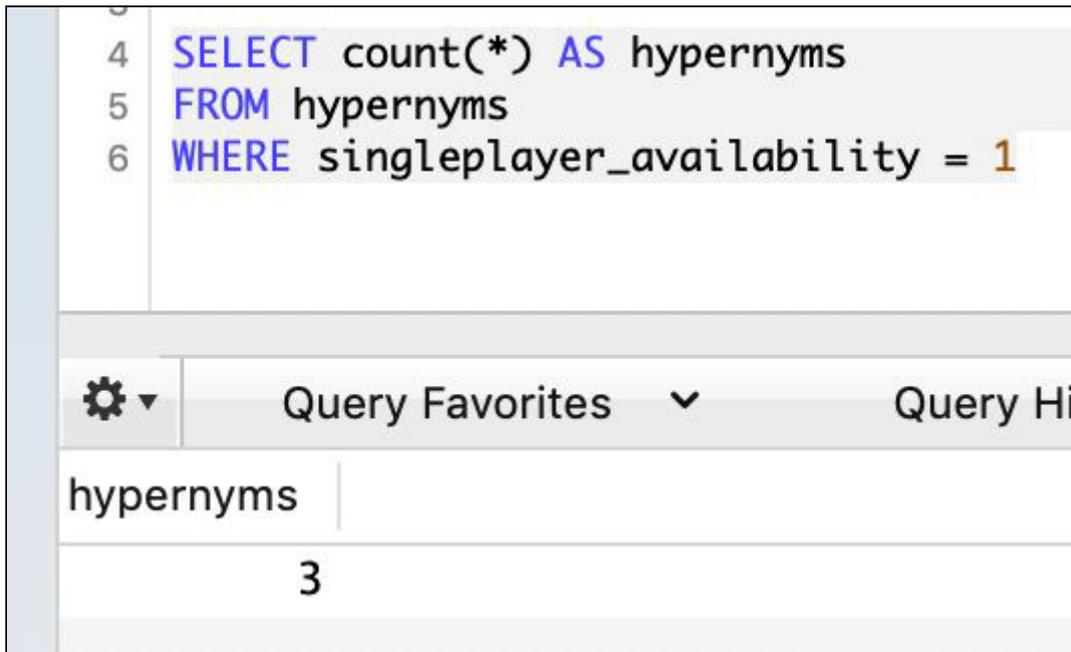
3.0.0 - Database screenshots:



[Fig 3.1.1] The number of synonyms available for the single-player game mode



[Fig 3.1.2] The number of antonyms available for the single-player game mode



[Fig 3.1.3] The number of hypernyms available for the single-player game mode



[Fig 3.1.4] The number of multiplayer game that have been played

```

1
2 SELECT COUNT(user_id) AS users
3 FROM users
4 WHERE user_id IN (
5     SELECT DISTINCT(p1_user_id)
6     FROM multiplayer_games
7 ) OR user_id IN (
8     SELECT DISTINCT(p2_user_id)
9     FROM multiplayer_games
10 );

```

Query Favorites

users

16

[Fig 3.1.5] The number of users who have played the multiplayer game

word	definition	answers	multiplayer_availability	multiplayer_occurrences	multiplayer_matches	multiplayer_passes	singleplayer_availability
Cold	Low temperature	{"freezing":6,"chilly":2,"subzero":1}	1	12	9	1	1
House	A place to live in	{"home":8,"cottage":1}	1	13	9	2	1
Eat	Intake food	{"digest":1,"devour":1,"consume":2,"munch":1}	1	10	5	3	1
Finger	A part of your hand	{"index":2,"thumb":1,"digit":3}	1	12	6	1	1

[Fig 3.1.6] The four synonyms that are available in the single-player game mode

word	definition	answers
Cold	Low temperature	{"freezing":6,"chilly":2,"subzero":1}

[Fig 3.1.7] The synonyms for the word "cold"

word	definition	answers	multiplayer_availability	multiplayer_occurrences	multiplayer_matches	multiplayer_passes
Cold	Low temperature	{"freezing":6,"chilly":2,"subzero":1}	1	12	9	1
Tree	A large thing that grows	{}	1	1	0	1

[Fig 3.1.8] The synonyms for the words “cold” and “tree”

```
5 SELECT COUNT(*) as ant_matched FROM multiplayer_answers WHERE game_mode = 'ANT' AND matched = 1;
```

Query Favorites Query History

ant_matched
80

[Fig 3.1.9] The number of matched antonyms

```
5 SELECT COUNT(*) as ant_passed FROM multiplayer_answers WHERE game_mode = 'ANT' AND passed = 1;
```

Query Favorites Query History

ant_passed
6

[Fig 3.1.10] The number of skipped antonyms

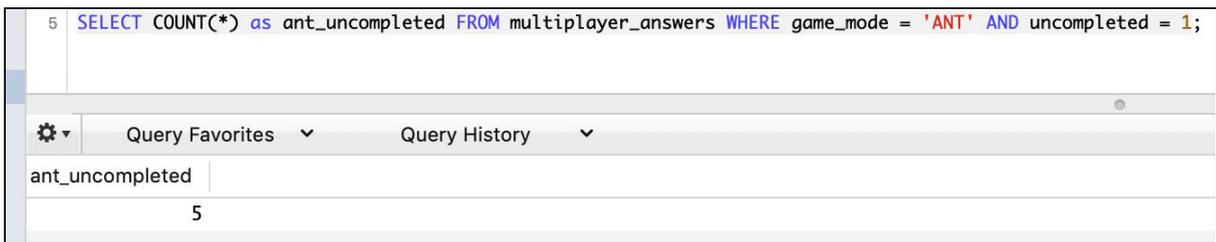
```
5 SELECT COUNT(*) as ant FROM multiplayer_games WHERE game_mode = 'ANT';
```

Query Favorites Query History

ant
14

[Fig 3.1.11] The number of antonym games

```
5 SELECT COUNT(*) as ant_uncompleted FROM multiplayer_answers WHERE game_mode = 'ANT' AND uncompleted = 1;
```

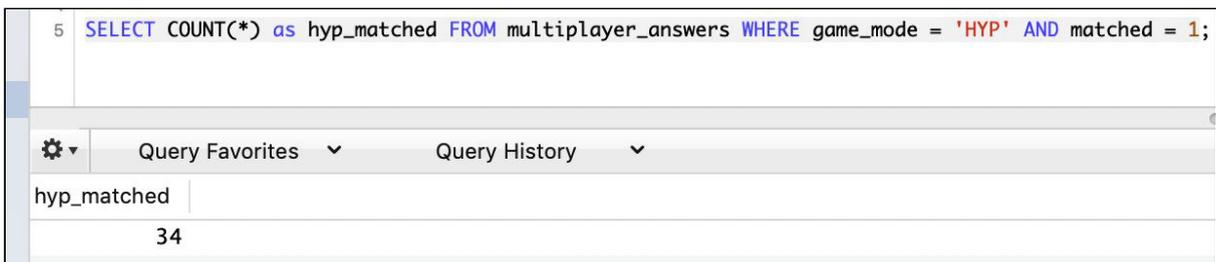


The screenshot shows a database query interface. At the top, a SQL query is entered: `5 SELECT COUNT(*) as ant_uncompleted FROM multiplayer_answers WHERE game_mode = 'ANT' AND uncompleted = 1;`. Below the query editor, there are two tabs: "Query Favorites" and "Query History". The results are displayed in a table with one column labeled "ant_uncompleted" and one row containing the value "5".

ant_uncompleted
5

[Fig 3.1.12] The number of uncompleted antonyms

```
5 SELECT COUNT(*) as hyp_matched FROM multiplayer_answers WHERE game_mode = 'HYP' AND matched = 1;
```

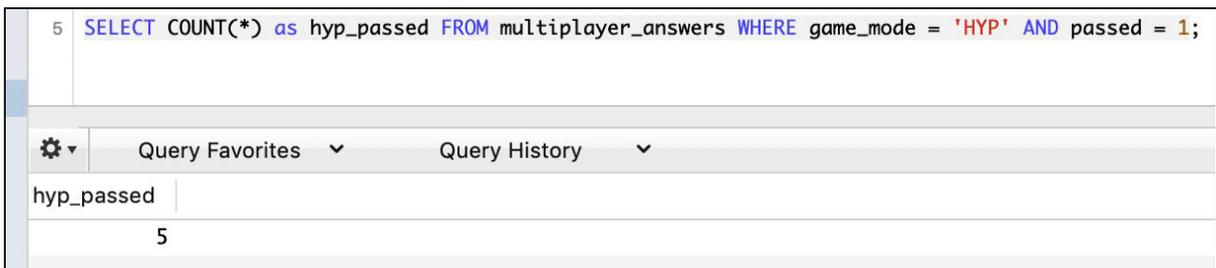


The screenshot shows a database query interface. At the top, a SQL query is entered: `5 SELECT COUNT(*) as hyp_matched FROM multiplayer_answers WHERE game_mode = 'HYP' AND matched = 1;`. Below the query editor, there are two tabs: "Query Favorites" and "Query History". The results are displayed in a table with one column labeled "hyp_matched" and one row containing the value "34".

hyp_matched
34

[Fig 3.1.13] The number of matched hypernyms

```
5 SELECT COUNT(*) as hyp_passed FROM multiplayer_answers WHERE game_mode = 'HYP' AND passed = 1;
```



The screenshot shows a database query interface. At the top, a SQL query is entered: `5 SELECT COUNT(*) as hyp_passed FROM multiplayer_answers WHERE game_mode = 'HYP' AND passed = 1;`. Below the query editor, there are two tabs: "Query Favorites" and "Query History". The results are displayed in a table with one column labeled "hyp_passed" and one row containing the value "5".

hyp_passed
5

[Fig 3.1.14] The number of skipped hypernyms

```
4
5 SELECT COUNT(*) as hyp FROM multiplayer_games WHERE game_mode = 'HYP';
```

Query Favorites Query History

hyp
9

[Fig 3.1.15] The number of hypernym games

```
5 SELECT COUNT(*) as hyp_uncompleted FROM multiplayer_answers WHERE game_mode = 'HYP' AND uncompleted = 1;
```

Query Favorites Query History

hyp_uncompleted
16

[Fig 3.1.16] The number of uncompleted hypernyms

```
5 SELECT COUNT(*) as syn_matched FROM multiplayer_answers WHERE game_mode = 'SYN' AND matched = 1;
```

Query Favorites Query History

syn_matched
70

[Fig 3.1.17] The number of matched synonyms

```
5 SELECT COUNT(*) as syn_passed FROM multiplayer_answers WHERE game_mode = 'SYN' AND passed = 1;
```

Query Favorites Query History

syn_passed
11

[Fig 3.1.18] The number of skipped synonyms

```
4
5 SELECT COUNT(*) as syn FROM multiplayer_games WHERE game_mode = 'SYN';
```

Query Favorites Query History

syn
20

[Fig 3.1.19] The number of synonym games

```
5 SELECT COUNT(*) as syn_uncompleted FROM multiplayer_answers WHERE game_mode = 'SYN' AND uncompleted = 1;
```

Query Favorites Query History

syn_uncompleted
59

[Fig 3.1.20] The number of uncompleted synonyms

4.0.0 - Miscellaneous screenshots:

- **S: (adj) cold**
 - see also
 - similar to
 - attribute
 - antonym
 - **W: (adj) hot** [Opposed to: **cold**]
 - derivationally related form

[Fig 4.1.1] Wordnet antonyms for the word “cold”

```
will@werdz:~$ pm2 status
```

App name	id	version	mode	pid	status	restart	uptime	cpu	mem	user	watching
matchmaking	1	1.0.0	fork	12645	online	59	6D	0.2%	59.2 MB	will	disabled
result_processing	2	1.0.0	fork	1898	online	10	9D	0.3%	58.4 MB	will	disabled
server	0	1.0.0	fork	1881	online	383	9D	0.2%	74.7 MB	will	disabled

[Fig 4.1.2] The API (server) and services (matchmaking and result_processing) running on the Digitalocean server



[Fig 4.1.3] The address of the website



werdz.fun
 Issued by: Let's Encrypt Authority X3
 Expires: Thursday, 23 May 2019 at 14:58:30 British Summer Time
 This certificate is valid

▼ Details

Subject Name
 Common Name: werdz.fun

Issuer Name
 Country or Region: US
 Organisation: Let's Encrypt
 Common Name: Let's Encrypt Authority X3

Serial Number
 04 6F C4 0E 53 3D 30 93 C3 18 0F 80 E4 38 AF 9B A1 B2
 Version: 3

Signature Algorithm
 SHA-256 with RSA Encryption (1.2.840.113549.1.1.1)
 Parameters: None

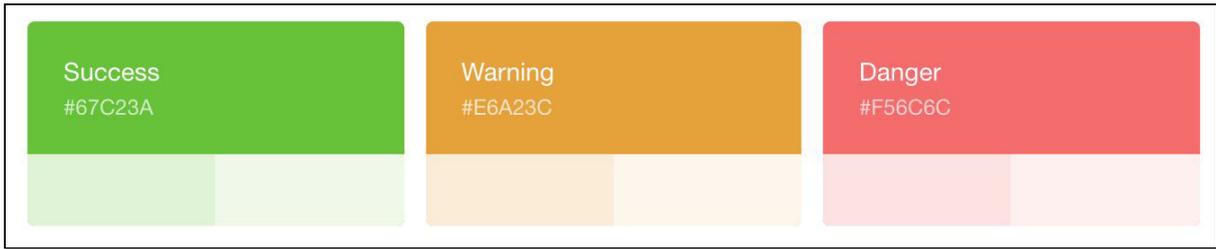
Not Valid Before
 Friday, 22 February 2019 at 13:58:30 Greenwich Mean Time
Not Valid After
 Thursday, 23 May 2019 at 14:58:30 British Summer Time

Public Key Info
 Algorithm: RSA Encryption (1.2.840.113549.1.1.1)
 Parameters: None
 Public Key: 256 bytes: BF 9C B3 61 BF D9 E0 6F ...
 Exponent: 65537
 Key Size: 2,048 bits
 Key Usage: Encrypt, Verify, Wrap, Derive
 Signature: 256 bytes: 7E 5E C4 7A A4 61 AC EE ...

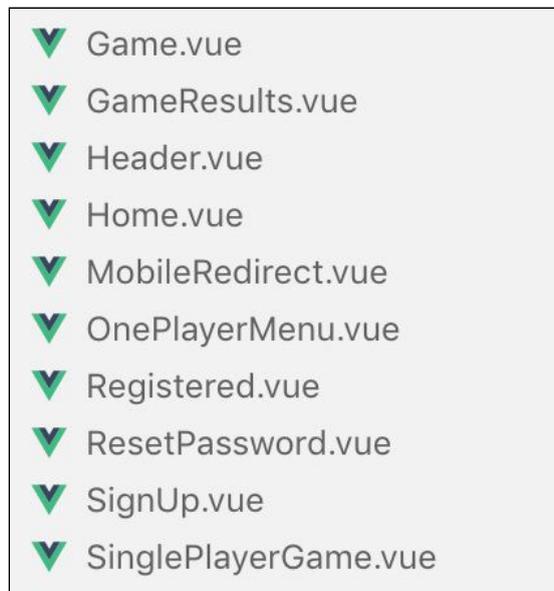
Extensions
 Extension: Key Usage (2.5.29.15)
 Critical: YES
 Usage: Digital Signature, Key Encipherment
 Extension: Basic Constraints (2.5.29.19)
 Critical: YES
 Certificate Authority: NO
 Extension: Extended Key Usage (2.5.29.37)
 Critical: NO
 Purpose #1: Server Authentication (1.3.6.1.5.5.7.3.1)
 Purpose #2: Client Authentication (1.3.6.1.5.5.7.3.2)
 Extension: Subject Key Identifier (2.5.29.14)
 Critical: NO
 Key ID: 30 DC 2E 34 B4 FA AA DD 0E 98 81 9D F9 08 89 C8 43 ED 6B A1
 Extension: Authority Key Identifier (2.5.29.35)
 Critical: NO
 Key ID: AB 4A 6A 63 04 7D DD BA E6 D1 39 B7 A6 45 65 EF F3 A8 EC A1
 Extension: Subject Alternative Name (2.5.29.17)
 Critical: NO
 DNS Name: werdz.fun
 DNS Name: www.werdz.fun
 Extension: Certificate Policies (2.5.29.32)
 Critical: NO
 Extension: Embedded Signed Certificate Timestamp List (1.3.6.1.4.1.11129.2.4.2)
 Critical: NO
 SCT Version: 1
 Log Key ID: 74 7E DA 83 31 AD 33 10 91 21 9C CE 25 4F 42 70 C2 BF FD 5E 42 20 08 C6 37 35 79 E6 10 7B CC 56
 Timestamp: Friday, 22 February 2019 at 14:58:30 Greenwich Mean Time
 Signature Algorithm: SHA-256 ECDSA
 Signature: 71 bytes: 30 45 02 20 0C F0 93 9B ...
 Extension: Certificate Authority Information Access (1.3.6.1.5.5.7.1.1)
 Critical: NO
 Method #1: Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)
 URI: <http://ocsp.int-x3.letsencrypt.org>
 Method #2: CA Issuers (1.3.6.1.5.5.7.48.2)
 URI: <http://cert.int-x3.letsencrypt.org/>

Fingerprints
 SHA-256: 32 E2 44 13 01 6D C9 33 B2 9E B2 B1 D1 99 49 B8 AD 78 E8 69 A9 01 1D D3 57 28 12 64 3B 3A 77 9D
 SHA-1: 10 47 D4 B2 AC E3 90 04 BE 99 E9 2C 9A 9A E4 80 7E 73 2C F7

[Fig 4.1.4] The website's certificate



[Fig 4.1.5] The default colours used by element-ui



[Fig 4.1.6] The Vue components, notably the files 'Game.vue' and 'SinglePlayerGame.vue'



[Fig 4.1.7] The ESLint and Prettier files

```
NODE_ENV = "development"
DB_HOST = "example.host.com"
DB_USER = "username"
DB_PASSWORD = "password"
DB_DATABASE = "database_name"
EMAIL_SERVICE = "gmail"
EMAIL_USER = "example@gmail.com"
EMAIL_PASS = "password"
```

[Fig 4.1.8] A copy of the *.env* file containing default values for the environment variables

Terminology guide:

API - Unless otherwise specified, this is referring to the Express app in the file 'server.js'. All POST requests and websockets are routed through the API.

Server - This refers to the Digitalocean droplet on which Node environments can be run. The files run on the server are 'server.js', 'matchmaking.js' and 'result_processing.js'.

Service - The files 'matchmaking.js' and 'result_processing.js' are run on the server and are referred to as "services".

Backend - The server side of the project repository.

Frontend - The client side of the project repository.

Normalise - Manipulating a string to a predefined standard format, for example a name would be lowercase except for the first letter.

Client - The user's browser.

User - The human using the system.

Player - The human playing the game.

Attacker - A human attempting to disrupt the system or retrieve sensitive information.

Glossary:

Antonym - a word association where 2 words have opposite meanings, for example “happy” is an antonym of “sad”.

API - an Application Programming Interface is a set of routines, protocols and tools for building software.

AWS - Amazon Web Services, a cloud computing platform.

Component - a building block of a Vue interface.

ESLint - a configurable linting tool for JavaScript code.

ES6 - the latest update to JavaScript.

Express - a web application framework for Node.js.

GDPR - the General Data Protection Regulation, a set of rules and regulations on data protection and privacy for the EU.

GET - a HTTP method used to request data from a specified source.

GitHub - a web based version control service.

HTML - Hypertext Markup Language, the standard markup language for web applications.

Hypernym - a word association where one words is a more general meaning of the other, for example “building” is a hypernym of “hospital”.

IP address - Internet Protocol address, a unique identifier for each device using the internet.

JavaScript - a programming language most commonly found in web applications.

JSON - JavaScript Object Notation, a human readable data object consisting of key:value pairs.

Key - the identifier of a data value in a JavaScript object.

Localhost - a host running on a local machine.

Metadata - data that provides information about other data.

MySQL - a relational database management system.

Node.js - a runtime environment for JavaScript files.

Object - see *JSON*.

PHP - Hypertext Preprocessor, a general purpose programming language originally designed for web applications.

Pm2 - a process manager for Node.js.

POST - a HTTP method used to send to a server.

Process - an executing instance of an application.

Raspberry Pi - a small affordable computer for small projects.

React - a JavaScript library for building user interfaces.

Server - a device that provides functionality for other devices/clients.

SQL injection - a technique whereby code is inserted into an SQL query for malicious purposes.

Synonym - a word association where 2 words have the same meaning, for example “fast” is a synonym of “quick”.

Thread - see *process*.

URL - Uniform Resource Locator, a web address.

UTC - Coordinated Universal Time, a primary standard time standard that is not adjusted for daylight savings.

Value - the data value of a JavaScript object, identified by a key.

VS Code - Visual Studio Code, a source code editor.

Vue.js - a JavaScript framework for building user interfaces.

Websocket - a computer communication protocol provided over a single TCP connection.

Word association - a link between 2 or more words.

Appendices

Appendix A: Research Integrity Training



Appendix B: Vue.js guide

To understand the implementation of this project, a basic understanding of the frontend framework Vue.js is required. A Vue component (any .vue file) has 3 sections - the 'template' section containing the HTML, the 'script' section containing the JavaScript functions, and the 'style' section containing any css. Data items and methods defined in the script can be referenced/called in the template. For example, to display a user's first name on a page, the template would contain the line `<p>{{user.forename}}</p>`. The script section is where the majority of the work happens. Here components and packages can be imported such as the *vue-password-strength-meter* component. This is followed by a JSON object containing 8 keys (or less if not all). These are as follows:

- Name: this is simply a string containing the name of the Vue file.
- Components: an object for defining the imported components so that they can be used in the HTML template.
- Data: this returns an object containing all of the data items that will be used throughout the script and template. For a data item to be referenced in the rest of the script, it must be predated by 'this' tag (eg: this.user.forename).
- Created: this is a function that is executed upon creation of the page, before the HTML items on the page are created. This can be useful for functions such as redirecting to another page if the user's login token is invalid.
- Mounted: this is a function that is executed upon creation of the page, after the HTML items on the page are created. This can be useful to access a component immediately after it has been rendered.
- Computed: this is an object containing multiple functions that are executed every time a change is detected in the computed function data. This can be useful to keep the page up to date according to the environment - for example disabling a button if there is no input into a form.
- Watch: this acts like the computed object, however a function only executes when a data item that shares its name changes. For example if the data item user changed, then the function user() in the watched object would be executed.
- Methods: this is an object containing multiple functions that can be called from the HTML template and from other places in the script by predating it with 'this'. This can be useful for executing JavaScript code upon an action, for example sending a POST request to the primary server when the user clicks a button.
- Destroyed: this is a function that is executed when the page is destroyed (eg: the user is redirected to another page). This can be useful for ending an ongoing process on a page, such as the heartbeat requests when queueing for the multiplayer game.

```
2
3 <template>
4   <div>
5     <el-header/>
6   </div>
7 </template>
8
9 <script>
10 import Header from './Header'
11
12 export default {
13   name: 'Template',
14   components: {
15     Header,
16   },
17   data() {
18     return {
19       user: null,
20     }
21   },
22   created() {},
23   mounted() {},
24   computed: {},
25   methods: {},
26   watch: {},
27   destroyed() {},
28 }
29 </script>
30
31 <style>
32 </style>
33
```

Appendix C: Database architecture

- 'Users' table: a table that contains all of a user's details - a unique id, a unique email address, a forename, a surname, the hash of a password, the account creation date, the last login date, a boolean value for the account's verification, a boolean value for whether the account has been deleted, the date of the deletion (if there was one).
- 'Sign_up_requests' table: a table that stores all requests to sign up to the website - a unique id, the user id of the user signing up, a token for that sign up (the one used for the unique link), a boolean value for validity of the sign up request, another boolean for completion of the request, and the completion date.
- 'Email_change_requests' table: a table that store all requests for a user to change their email address - identical to 'sign_up_requests', however it also has an attribute for the requested email address.
- 'Password_reset_requests' table: a table to store all requests for a user to recover their password - identical to 'sign_up_requests'
- 'Queued_users' table: a table to store users who are queued for the multiplayer game - a unique id, the user id, gamemode, a boolean for validity of the queued user, the initialisation date (when the user joined the queue), a boolean for whether the matchmaking system removed the user from the queue due to a dead heartbeat, a boolean value for matched if the system found another player to match with, the match date, the id of the match in the 'queued_users' table, the user id of the user that was matched with, the date of the last heartbeat, and the game token.
- 'Multiplayer_games' table: a table to store the data of each multiplayer game - a unique id, the user id of player 1, the user id of player 2, the game mode, the game token, the initialisation date, the termination date, the words and their definitions for the game stored as a stringified array of javascript objects (eg: [{"word":"Cold","def":"Low temperature"}, {"word":"House","def":"A place to live in"}]), and 4 booleans for the games validity, completion, removed and quitted.
- 'Multiplayer_answers' table: a table to store the results of each word from each multiplayer game - a unique id, the game id, the game mode, the word, player 1 and player 2's answers stored as a stringified arrays (eg: ["chilly", "freezing"]), a boolean for whether a match was found, the match if there was one, and booleans for whether the word has been passed, uncompleted, and processed (processed being whether the result processing server has taken into account this result yet).
- 'Singleplayer_games' table: a table to store the data of each singleplayer game - identical to 'multiplayer_games' except that there is only one player, so only one user id needs to be stored
- 'Singleplayer_answers' table: a table to store the results of each word for each singleplayer game - identical to 'multiplayer_answers' except that there is only one player, so only one array of answers needs to be stored. There is also no boolean processed attribute as the singleplayer answers are not being processed by the result processing server.

- 'Synonyms', 'antonyms' and 'hypernyms' tables: tables to store the words for each game mode and the processed data of each word - a unique id, the word, the definition, the answers that have occurred and their frequency stored as a stringified javascript object (eg: {"cold":5,"freezing":1}), a boolean value for the words availability in the multiplayer game, the total occurrences of that word in the multiplayer game, the number of times a match has been found for that word, the number of times that word has been skipped, and whether that word is available in the singleplayer game mode.

Appendix D: Website terms and conditions

Last updated: 27th of February, 2019

Please read these Terms and Conditions ("Terms", "Terms and Conditions") carefully before using the werdz.fun website (the "Service") operated by William Cooter ("us", "we", or "our"). Your access to and use of the Service is conditioned on your acceptance of and compliance with these Terms. These Terms apply to all visitors, users and others who access or use the Service. By accessing or using the Service you agree to be bound by these Terms. If you disagree with any part of the terms then you may not access the Service.

Accounts

When you create an account with us, you must provide us information that is accurate, complete, and current at all times. Failure to do so constitutes a breach of the Terms, which may result in immediate termination of your account on our Service. You are responsible for safeguarding the password that you use to access the Service and for any activities or actions under your password, whether your password is with our Service or a third-party service. You agree not to disclose your password to any third party. You must notify us immediately upon becoming aware of any breach of security or unauthorized use of your account.

Links To Other Websites

Our Service may contain links to third-party web sites or services that are not owned or controlled by us. We have no control over, and assumes no responsibility for, the content, privacy policies, or practices of any third party web sites or services. You further acknowledge and agree that we shall not be responsible or liable, directly or indirectly, for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods or services available on or through any such web sites or services. We strongly advise you to read the terms and conditions and privacy policies of any third-party web sites or services that you visit.

Termination

We may terminate or suspend access to our Service immediately, without prior notice or liability, for any reason whatsoever, including without limitation if you breach the Terms. All provisions of the Terms which by their nature should survive termination shall survive termination, including, without limitation, ownership provisions, warranty disclaimers, indemnity and limitations of liability. We may terminate or suspend your account immediately, without prior notice or liability, for any reason whatsoever, including without limitation if you breach the Terms. Upon termination, your right to use the Service will immediately cease. If you wish to terminate your account, you may simply discontinue using the Service. All provisions of the Terms which by their nature should survive termination shall survive termination, including, without limitation, ownership provisions, warranty disclaimers, indemnity and limitations of liability.

Governing Law

These Terms shall be governed and construed in accordance with the laws of United Kingdom, without regard to its conflict of law provisions. Our failure to enforce any right or provision of these Terms will not be considered a waiver of those rights. If any provision of these Terms is held to be invalid or unenforceable by a court, the remaining provisions of these Terms will remain in effect. These Terms constitute the entire agreement between us regarding our Service, and supersede and replace any prior agreements we might have between us regarding the Service.

Changes

We reserve the right, at our sole discretion, to modify or replace these Terms at any time. If a revision is material we will try to provide at least 30 days notice prior to any new terms taking effect. What constitutes a material change will be determined at our sole discretion. By continuing to access or use our Service after those revisions become effective, you agree to be bound by the revised terms. If you do not agree to the new terms, please stop using the Service.

Contact Us

If you have any questions about these Terms, please contact us at game.cooter@gmail.com.

1. Node.js, 2019. *Previous Releases*. Available at <https://nodejs.org/en/download/releases/> [Accessed on: 06/05/2019]
2. PHP, 2019. *History of PHP: PHP Tools, FI, Construction Kit, and PHP/FI*. Available at https://www.php.net/manual/en/history.php.php?fbclid=IwAR3vyifU4P4ZHSCnIvSDilq5WAtTZi7G5WSPABh_I59SQ3j5gz_4uHPOJ5c [Accessed on: 02/02/2019]
3. Node.js, 18th July 2018. Node.js v10.7.0. Available at <https://nodejs.org/en/download/releases/> [Accessed on: 02/02/2019]
4. Vue.js, 8th January 2019. Vue-cli v3.3.0. Available at <https://cli.vuejs.org/guide/installation.html> [Accessed on 09/02/2019]
5. MySQL, 21st January 2019. MySQL v5.6.40. Available at <https://dev.mysql.com/downloads/mysql/5.6.html> [Accessed on 02/02/2019]
6. Bcrypt, April 2019. Bcrypt v2.0.0. Available at <https://www.npmjs.com/package/bcrypt/v/2.0.0> [Accessed on 17/02/2019]
7. Crypto, 2017. Crypto v1.0.1. Available at <https://www.npmjs.com/package/crypto> [Accessed on 17/02/2019]
8. Express, November 2018. Express v4.16.3. Available at <https://www.npmjs.com/package/express/v/4.16.3> [Accessed 04/02/2019]
9. Jsonwebtoken, March 2019. Jsonwebtoken v8.2.1. Available at <https://www.npmjs.com/package/jsonwebtoken/v/8.2.1> [Accessed 22/02/19]
10. Lodash, 12th September 2018. Lodash v4.17.11. Available at <https://lodash.com/> [Accessed 24/02/19]
11. Moment.js, January 2019. Moment.js v2.24.0. Available at <https://momentjs.com/> [Accessed 03/03/19]
12. Nodemailer, 19th April 2019. Nodemailer v4.7.0. Available at <https://www.npmjs.com/package/nodemailer/v/4.7.0> [Accessed 24/02/19]
13. Socket.io, 29th November 2018. Socket.io v2.2.0. Available at <https://socket.io/docs/> [Accessed 17/03/19]
14. Axios, August 2018. Axios v0.18.0. Available at <https://www.npmjs.com/package/axios> [Accessed 05/02/19]
15. Element-ui, 25th August 2019. Element-ui v2.5.4. Available at <https://element.eleme.io/?ref=madewithvuejs.com#/en-US/component/installation> [Accessed 28/02/19]
16. Password-strength-utility, 2017. Password-strength-utility v1.1.6. Available at <https://www.npmjs.com/package/password-strength-utility> [Accessed 17/02/2019]
17. Socket.io-client, December 2018. Socket.io-client v2.2.0. Available at <https://www.npmjs.com/package/socket.io-client> [Accessed 17/03/19]
18. Vue, 8th January 2019. Vue-cli v3.3.0. Available at <https://cli.vuejs.org/guide/installation.html> [Accessed on 09/02/2019]