NEA

Participant – Gabriel Drozbik (id=338676)

# Table of contents

# Introduction and objectives: Page 3-5

# Analysis: Page 5-10

# Documented design: Page 10-22

# Technical solution: Page 0

# Testing: Page 0

# Evaluation: Page 0

**Analysis :**

Problem:

A lot of people (myself included) struggle with focus, remembering school content and studying because reading your business textbook and writing notes over and over becomes tiring really quickly; and the next best use of time appears. Memorization without any distinct emotion towards the material Is close to impossible.

Nearly all studying materials are locked behind a paywall, buying new textbooks every year are expensive, and to use any good learning platform, it’s required that you pay a monthly subscription.

When corporations can’t make you pay for their services, they’ll give up, yet will try to make money using you one way or another, whether it’s by inconvenience (and by paying, you’ll get a better experience), forcing you to sign up for a free trial using your credit card information, or outright profiting off of your data.

Research:

Currently, the best method for studying is spaced repetition, with the best platform being Anki, which is a free program. It revolves around using flashcards with questions or some sort of prompt, to which you answer out loud, or in your head – it doesn’t exactly explain the meaning behind the answers, and it’s purely used a tool for revising things you already know – that’s not to say that you can’t use it to learn, but like I stated above in the problems section, when you don’t have any other piece of information to connect to the one you’re trying to learn, it just won’t stick; which is why if you want to learn a language using Anki, you have to regularly consume content in that language as well. Once you have “your” answer, you click on a button which reveals the answer, and then you have an assortment of other options – times from which you have to choose (1minute, 4 minutes etc) – and whatever option you pick is how long it’ll take before you see that card again.

The most famous quiz platform is kahoot, with it’s multiplayer and session leader-board features. Kahoot is famous for its 4 options style, where you get a question, and have a 25% chance to get the answer right. It’s mostly characterised as nothing more than an educational game platform that teachers put on when there isn’t any more school work left to do, with difficulties between quizzes being inconsistent and options often being poorly made from a revision point of view; sometimes becoming a game of guess the odd one(s) out. For serious studying it’s a poor platform, because with enough intuition, you could win without knowing the content – which won’t help you in an exam where you don’t get any clues.

Research of the end user:

My end user is my little brother who’s in secondary school. He struggles to focus in classes that he doesn’t really enjoy, which can be seen with the discrepancy between his most recent mathematics grade (91%) and his religious education grade (42%). For now, this isn’t much of a problem, but the issue is that he’s able to get good maths grades out of pure passion, while the opposite shows for RE – and in the long term, there is not telling for how his interests in school subjects change.

Requirements:

Objectives:

The problem will be solved with a website - which is a lot simpler than having others download an electron program - and it will allow more for reach.

The idea of having a quiz application; that can be used offline, is appealing, if it wasn't for the fact that designing a package manager to distribute quiz content is overkill.

To make revision/learning actually enjoyable (even if the content is boring) or better, making it stand out in one way or another (with images for example) would help study without forgetting what you just read a minute ago, increasing efficiency.

Taking inspiration from git and GitHub, the idea of collaboration being possible will make the quality of content a lot higher; being able to add content onto others projects means that a complete source for learning can be produced and polished a lot sooner (with the creators permission). The ability to fork projects also means that outdated or half-completed content can be completed on the behalf of somebody else. With this system, licenses stating the rules for what you're allowed to do with someones work or not will be in place.

I want to learn JavaScript – a general purpose scripting language commonly used in web development - as quickly as possible so that I can migrate my code to typescript and use it instead. JavaScript is notorious for having horrible debugging, and If you make a mistake, you get a cryptic error in the browser built in terminal which really does nothing but point out that something doesn’t work. Typescript is the complete opposite, with easily understandable and sensible error messages. It shares a decent amount of syntax with JavaScript, so all JavaScript code (can be) valid typescript code (you have to enable it in the compiler configuration file – tsconfig.json), and it’s designed to be easy for migration from JavaScript. Typescript code compiles into JavaScript code which can then be run using node.js which is a JavaScript runtime environment.

The alternative to JavaScript or Typescript is PHP, which does a similar (if not the same) job, but it’s becoming less and less common – I would rather pick the other languages because documentation would be easier to find, and If I want to do web development after school, being able to work with JavaScript is more commonly needed (as far as I could tell from my research).

In short:

* To make it easier to revise and study even when content is boring or when you find it difficult to focus,
* To make the quiz creation system easy and quick so that you can quickly get to revising if under time constraint,
* To allow flexibility in the way you learn, having dynamic quizzes, and have things easy configurable to your liking (which will help with productivity),
* To make it easy to collaborate with others, and to make the content on the website a pool that isn’t stagnant.

**Documented design:**

Objectives:

The first objective for the project is to make set up a simple markdown environment (using HTML and CSS) so that I can begin to work on the back-end.

The tactic is to work on the back-end as soon as possible which will be used as the core of the whole project, which can easily be used as a template for more complicated ideas.

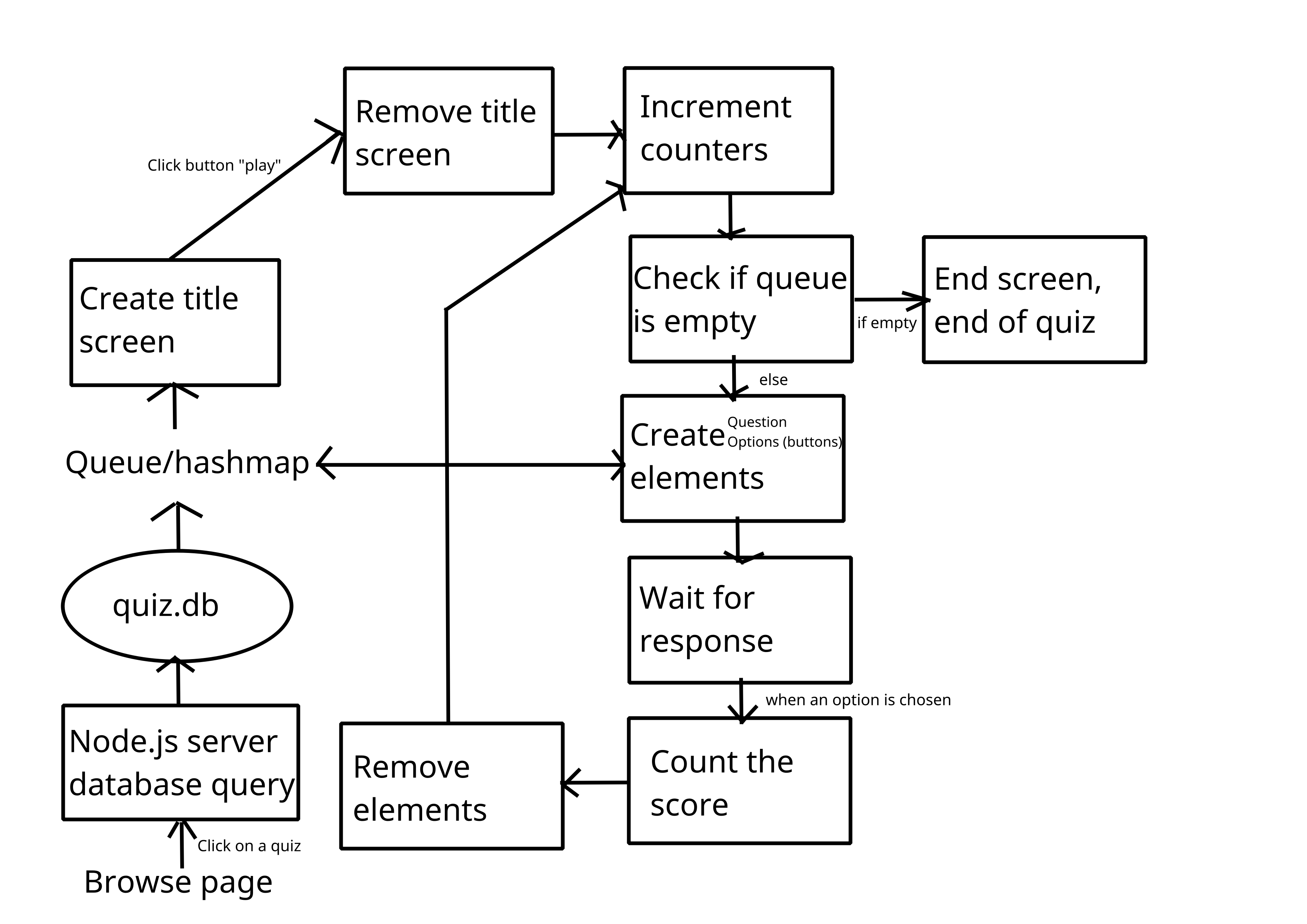
By back-end, I'm specifically talking about the quiz logic. After I polish it, I'll set up an SQL database, and do other things around the website - making a system of submitting and requesting data for example for quiz creation and account sign up.

I'll only begin to work on the front-end at the end , however - the buttons for the quiz will exist at the very beginning so that I can test the quiz logic I create.

Structure/hierarchy diagrams:

System flowcharts:

(flowchart for the quiz logic)

(count the score – to evaluate if the users choice is correct or wrong, and then increment score is correct)

Data flow diagrams:

user action -> script -> server.js -> database -> server.js -> ts script (ex. Game)

^ This is also a correction for the flowchart above.

Object-oriented design:

Database design:

There will be two databases, one in which account logins are stored, and another with quiz content. SQLite is going to be used so a database will be in a .db file format.

The account database will be straight forward, with just one table. The quiz database will be more complicated, and it'll have interlinked tables - so the title for the quiz is in one table, the questions in another, and so on for the rest of the data (options, answers, settings etc).

(I’ll put an illustration of quiz.db)

Algorithms:

Binary search will be used to look up quizzes in the database, which realistically, the whole website will depend on; to find and play a quiz, a search of some kind will have to be done. Different filters in the browse section will also exist. A hashing algorithm will also be used for the database to make look-ups quicker. An encryption algorithm will be used on the account logins (passwords) to ensure a degree of security.

Data structures and advanced technologies:

In the quiz logic ([src/play.ts] below), there will be use of

a queue so data needed will be popped onto the queue from the

database, and pushed once the user moves onto the next question.

File structures:

- docs/

Nea-Student-project-Tracker.txt

Nea-Student-project-Tracker.docx

- public/

- CSS/

style.css

- database/

quiz.db

account.db

- fonts/

Ubuntu-Light.woff2

Ubuntu-Medium.woff2

Ubuntu-Regular.woff2

- js/

(All the compiled typescript code (in src) goes here)

- particles/

(particles\_app.js)

particles.json

- src/

create.ts

form.ts

play.ts

server.ts

particles\_app.ts

- views/

account.html

browse.html

create.html

index.html

login.html

play.html

signup.html

signup-success.html

submit-quiz-metadata.html

(there are more HTML files than is required, I was trying to learn html POST functions, but I never got around to finish that)

HCI (Human-computer interaction) / Screen design:

Hardware selection:

I'm making use of standard hardware (I'm not using a raspberry-pi or anything), however, the operating system I'm using should be considered.

Using a distribution of Linux on my laptop (arch Linux), file paths use forward slashes, unlike windows back slashes. Statistically, if the

website was to be ran on a server, it would most likely use Debian or another Linux distribution, and file paths wouldn't be a problem.

Still, if the website was to be run on windows, I should make sure that the file paths work.

Fully articulated design:

**Documented Design:**

In JavaScript/Typescript, comments are written using “//”;

Server.ts:

import express = require('express'); // Importing the HTTP server library

const app = express(); // Defining the server node

import path = require('path'); // Importing external files to be used

app.use(express.static(path.join(\_\_dirname, 'public')));

import bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: true }))

import Database = require('better-sqlite3'); // Importing the SQLite

const quiz\_db = new Database('public/database/quiz.db', { verbose: console.log });

const account\_db = new Database('public/database/account.db', { verbose: console.log });

quiz\_db.pragma('journal\_mode = WAL'); // In the documentation, this was recommended to use, but I’m not sure if it’s really needed or not

account\_db.pragma('journal\_mode = WAL');

const PORT = 8000; // Using port 8000, so the website will run at http://localhost/8000

app.listen(PORT, () => {

console.log("Server's started at http://localhost:8000");

})

// Setting up the different parts of my website using routing

// Example: "/play" -> "http://localhost:8000/play"

app.get("/", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'index.html'));

})

app.get("/browse", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'browse.html'));

})

app.get("/play", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'play.html'));

})

app.get("/create", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'create.html'));

})

app.get("/account", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'account.html'));

})

app.get("/signup", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'signup.html'));

})

app.get("/signup", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'signup.html'));

})

// Possibly unnecessary

app.get("/signup-success", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'signup-success.html'));

})

app.get("/login", (req, res) => {

res.sendFile(path.join(\_\_dirname, 'views', 'login.html'));

})

// This doesn’t work, here I’m trying to learn to post data to the server, and then to the database

app.post('/submit-quiz-metadata', (req, res) => {

res.render('the\_template', { name: req.body.name });

//console.log('Recieve data from client:', userData);

// SQL logic below:

const insert = quiz\_db.prepare('INSERT INTO Meta (value1, value2, value3) VALUES(?, ?, ?)');

insert.run('value1', 'value2', 'value3');

quiz\_db.close();

});

// Making the tables for quiz.db (if they don’t exist)

quiz\_db.exec(`

CREATE TABLE IF NOT EXISTS Meta(

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT,

creator TEXT

);

CREATE TABLE IF NOT EXISTS Settings(

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT UNIQUE,

value TEXT

);

CREATE TABLE IF NOT EXISTS Questions(

id INTEGER PRIMARY KEY AUTOINCREMENT,

question\_text TEXT,

category TEXT

);

CREATE TABLE IF NOT EXISTS Options(

id INTEGER PRIMARY KEY AUTOINCREMENT,

question\_id INTEGER,

option\_text TEXT,

FOREIGN KEY (question\_id) REFERENCES Questions(id)

);

CREATE TABLE IF NOT EXISTS Answers(

id INTEGER PRIMARY KEY AUTOINCREMENT,

question\_id INTEGER,

option\_id INTEGER,

FOREIGN KEY (question\_id) REFERENCES Questions(id),

FOREIGN KEY (option\_id) REFERENCES Options(id)

);

`);

quiz\_db.close();

// Making a table for the logins

account\_db.exec(`

CREATE TABLE IF NOT EXISTS Logins(

id INTEGER PRIMARY KEY AUTOINCREMENT,

[username] TEXT UNIQUE,

[password] TEXT

);

`);

account\_db.close();

play.ts:

// I used a class because I wanted to try out OOP, but it isn’t really useful here

class quizMeta {

name: string = "Python quiz";

author: string = "Gabriel";

button\_text: string = "Play";

}

// Making an object, qm for quiz meta (information about the quiz - metadata)

const qm = new quizMeta();

// Setting up elements using the class

const gameTitle = document.getElementById("game\_title");

const gameTitleChild = document.createTextNode(qm.name);

gameTitle!.appendChild(gameTitleChild);

const gameCreator = document.getElementById("game\_creator");

const gameCreatorChild = document.createTextNode(qm.author);

gameCreator!.appendChild(gameCreatorChild);

const startButton = document.getElementById("start\_button");

const startButtonChild = document.createTextNode(qm.button\_text);

startButton!.appendChild(startButtonChild);

// Used another class, again, this doesn’t really improve anything besides organization

class quizContent {

// q = questions, o = options, a = answers

q: Array<string> = ["Click 1", "Click 2", "Click 3!"];

o: Array<string> = ["One", "Two", "Three", "Four"];

a: Array<string> = ["One", "Two", "Three", "Four"];

}

const qz = new quizContent();

// Pre-quiz screen;

// the function gets called if the HTML start button gets clicked,

// this function is only used at the beginning.

function start\_quiz() {

// Clearing the pre-quiz screen

document.getElementById("game\_title")!.remove();

// The exclamation mark removes the error stating that -

// ('game\_title' is 'possibly null').

document.getElementById("game\_creator")!.remove();

document.getElementById("start\_button")!.remove();

let round = 0; // Setting up the counters – round and score

let score = 0;

verification(round, score);

}

// Field verification:

// To check if there are questions left,

// if not, the end screen will be triggered.

function verification (round: number, score: number) {

round ++; // Incrementation

// Was used for testing:

// console.log(round);

// If the number of rounds is higher than the number of questions, the quiz ends

if ((qz.q[round - 1]) == null){ //

console.log("The game has finished");

end\_screen(round, score);

}

else{

object\_creation(round, score);

}

}

// Quiz

function object\_creation (round: number, score: number) {

const buttonContainer = document.getElementById("button\_container");

/\* Display quiz\_q[x]\*/

const newH2 = document.createElement("h3");

const q = document.createTextNode(qz.q[round - 1]);

newH2.appendChild(q);

newH2.id = "question";

document.body.insertBefore(newH2, buttonContainer);

/\* Display quiz\_a[x]\*/

/\* Need to add the onclick element/property to the buttons \*/

const a = document.createElement("BUTTON");

const a\_test = (qz.o[0]);

const a\_node = document.createTextNode(a\_test);

a.appendChild(a\_node);

a.id = 'choice\_a';

buttonContainer!.appendChild(a);

const b = document.createElement("BUTTON");

const b\_test = (qz.o[1]);

const b\_node = document.createTextNode(b\_test);

b.appendChild(b\_node);

b.id = 'choice\_b';

buttonContainer!.appendChild(b);

const c = document.createElement("BUTTON");

const c\_test = (qz.o[2]);

const c\_node = document.createTextNode(c\_test);

c.appendChild(c\_node);

c.id = 'choice\_c';

buttonContainer!.appendChild(c);

const d = document.createElement("BUTTON");

const d\_test = (qz.o[3]);

const d\_node = document.createTextNode(d\_test);

d.appendChild(d\_node);

d.id = 'choice\_d';

buttonContainer!.appendChild(d);

waiting\_for\_ans(a, b , c, d, a\_test, b\_test, c\_test, d\_test, round, score);

}

/\* Testing for the answer\*/

function waiting\_for\_ans(a: HTMLElement, b: HTMLElement, c: HTMLElement, d: HTMLElement, a\_test: string, b\_test: string, c\_test: string, d\_test: string, round: number, score: number) {

function listen() {

async function handleClick(choice: string, round: number, score: number) {

evaluate(choice, round, score);

}

a.addEventListener('click', () => handleClick(a\_test, round, score));

b.addEventListener('click', () => handleClick(b\_test, round, score));

c.addEventListener('click', () => handleClick(c\_test, round, score));

d.addEventListener('click', () => handleClick(d\_test, round, score));

}

listen();

}

// Test the answer

function evaluate(choice: string, round: number, score: number) {

let answer: string = (qz.a[round - 1]);

if (choice == answer) {

score++;

console.log('Correct');

console.log(score + '/' + round);

purge\_screen(round, score);

}

else {

console.log("Wrong");

console.log(score + '/' + round);

purge\_screen(round, score)

}

}

function purge\_screen(round: number, score: number) {

document.getElementById("question")!.remove();

document.getElementById("choice\_a")!.remove();

document.getElementById("choice\_b")!.remove();

document.getElementById("choice\_c")!.remove();

document.getElementById("choice\_d")!.remove();

// Restarting the loop of functions

verification(round, score);

}

function end\_screen(round: number, score: number){

const endScreenContainer = document.getElementById('end\_screen\_container');

const end\_banner = document.createElement('h3');

const end\_banner\_node = document.createTextNode("Congratulations!");

end\_banner.appendChild(end\_banner\_node);

end\_banner.id = "endBanner";

endScreenContainer!.appendChild(end\_banner);

// I haven't made a score tracker yet

const end\_score = document.createElement('h4');

const end\_score\_node = document.createTextNode("Your score is " + score + "/" + (round - 1) + "!");

end\_score.appendChild(end\_score\_node);

end\_score.id = "endScore";

endScreenContainer!.appendChild(end\_score);

}

/\* Structure:

\* Displaying a "Play" screen (before the quiz),

\* Removing the existing objects,

\* Adding the buttons,

\* Adding text for the quiz,

\* Waiting for the answer,

\* Checking for incorrect/correct answer,

\* Keeping track of the score

\* Displaying so,

\* Next question (looping),

\* When the quiz is done:

\* Removing all the existing objects

\* Adding a quiz screen (with scores)

\*/