

## Contents

Implementation .....	2
Limitations, Future Improvements and Final Reflection .....	20

## Implementation

The purpose of this document is to discuss the final deliverable I have created. Now that the development work on the project has finished, I will be discussing the solution and how I chose to implement the requirements specified by the client. I will cover the technologies and tools used, the inner workings of the code and I will also evaluate how well the code I produced meets the requirements presented to me.

The solution was built using a variety of different open-source technologies available, which I will discuss in further detail below.

For the front-end development I used Vue.js 3 and Tailwind CSS. Vue is a JavaScript framework which is lightweight and makes creating single-page applications incredibly easy. It has many features which I found to be useful, such as the Composition API which simplifies the code writing process even more so and makes scalability and maintainability much easier. Another benefit of Vue is the use of components to break up code into logical pieces where each main aspect of functionality is in its own file.

Tailwind is a CSS framework that makes styling elements on the page extremely quick via its class-based approach where you simply write the styling in-line, as part of the element setup. Page design or styling is not one of my strong points, but with Tailwind it simplified the process a lot and made it very easy to create a site where I was pleased with the end-result.

For the state management, I decided to use Pinia. This is the latest iteration of the global state management framework built specifically for Vue applications. I decided on this because of its ease of implementation and the fact it is the latest version.

Finally, I decided to use TypeScript rather than vanilla JavaScript as this allowed me to ensure all my data types were safe and would prevent me from running into many issues caused by the loosely typed JavaScript.

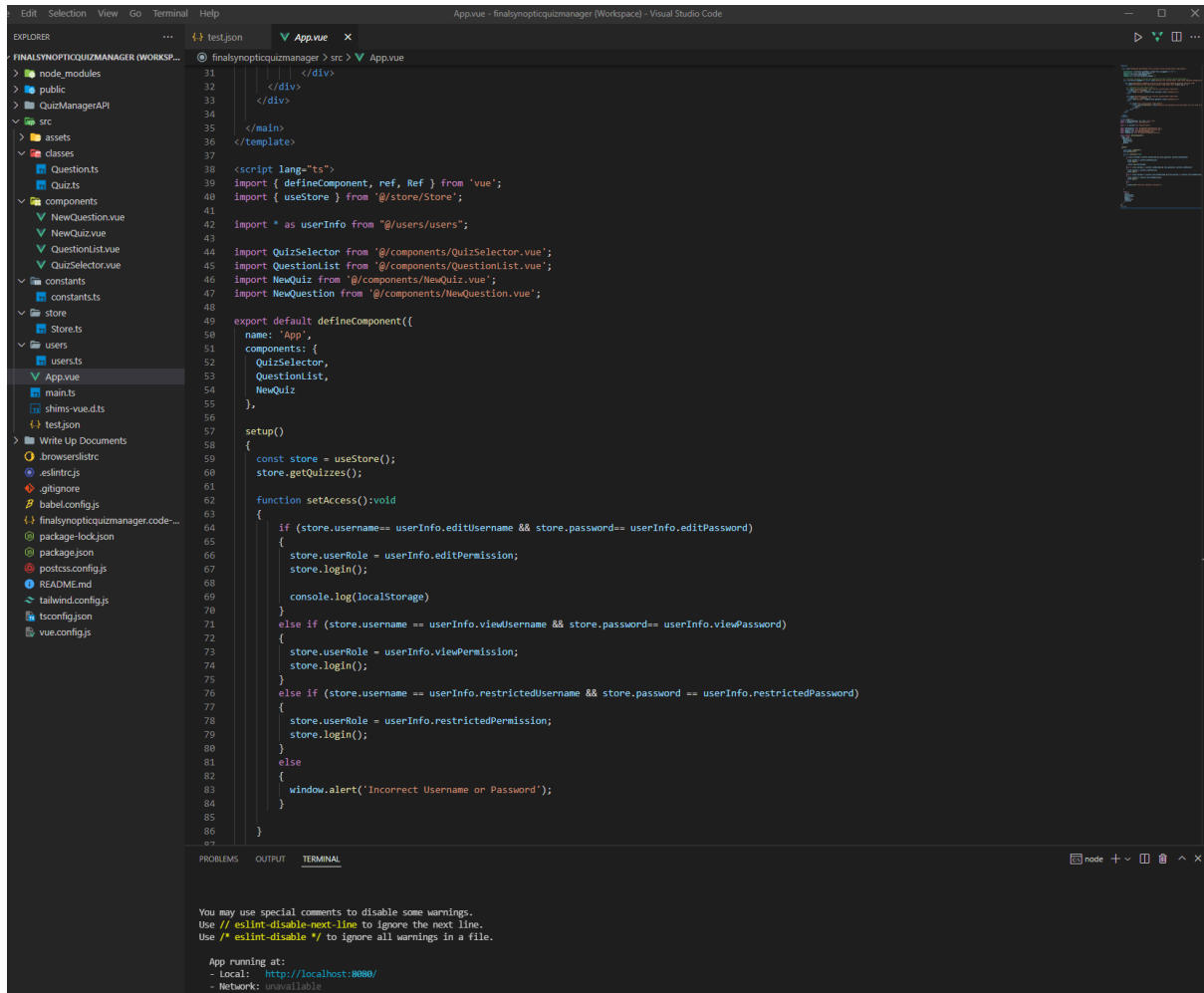
With regards to the back-end, I ran into several issues which forced me to completely re-think how I would approach this. I had originally planned to use SQL Server as my database and had built my designs around this, but unfortunately, I had technical issues which prevented me from doing so.

In the end I settled on using MongoDB. More specifically, I chose MongoDB Atlas for my data store, which is their cloud-based solution and required no software downloads to get up and running. MongoDB is regarded as a NoSQL database and rather than being built around tables and the relationships between those tables, MongoDB is a non-relational database built around what they call 'Collections' which include 'Documents'.

To pass data from the front end to the back end I built my own API from scratch. There were many options available to me for doing this. Typically, people would use what is known as a MEVN stack. The acronym **MEVN** stands for *MongoDB*, *Express.js*, *VueJS*, *Node.js*.

However, I decided to substitute Express and Node for C# .NET Core. More specifically, I built my API using .NET Framework 6, which is the very latest iteration and still in preview rather than a full release. The reasoning behind this is due to the release of the Minimal API framework. This is used to allow the creation of a basic API very quickly, without the large amounts of boilerplate code that .NET APIs are known for. Another reason I chose this route

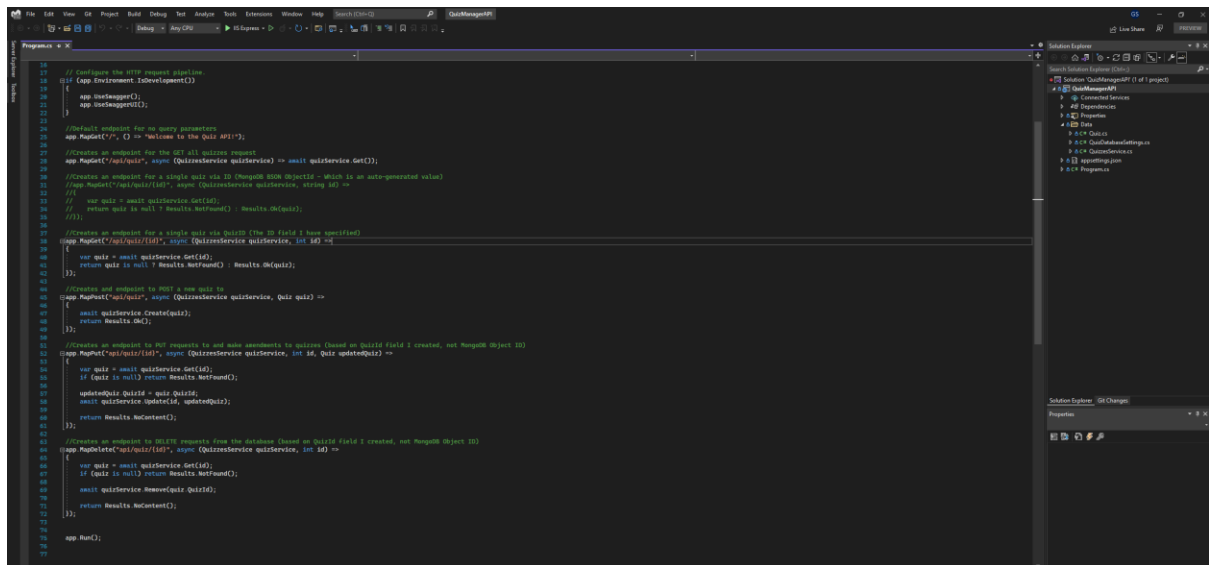
is due to having used .NET to build APIs in previous projects and wanting to use the latest iteration of the technology to prepare me for future projects.



Screenshot showing the IDE I used for the frontend development (Visual Studio Code)

As you can see above, I decided to use Visual Studio Code for my front-end development. The reason for this is because of the fact VS Code has become incredibly popular and useful over the last few years. There are many plugins and extensions which you can use to improve the efficiency of your workflow and it is very user friendly and easy to work with.

Also included in the image above, you can see a series of files and folders. These are most of the files included for the front-end aspect of my application. You can see the files are separated into folders, most of which being Vue and JavaScript related files, such as Login.vue which is the overlay that appears on the site and allows the user to log in.



Screenshot of the IDE used to create my API (Visual Studio 2022)

The above image shows my development environment for creating the .NET API. I used Visual Studio 2022 as it is the latest version and the only one which supports developing using .NET Framework 6.

You can see snippets of code in the screenshot, and this is how I created the different API calls used to work with the data. Traditionally, in C# this would take much more work and a lot more lines of code, but the Minimal API simplifies this process a lot. Each of the requests (GET, to obtain data from the database. POST, to add new data to the database. PUT to edit data in the database and DELETE to remove data from the database) now only take a few lines of code each.

**quizmanager.quizzes**

STORAGE SIZE: 36KB TOTAL DOCUMENTS: 11 INDEXES TOTAL SIZE: 36KB

Find Indexes Schema Anti-Patterns ⓘ Aggregation Search Indexes ●

INSERT DOCUMENT

FILTER { field: 'value' } OPTIONS Apply Reset

QUERY RESULTS 1-11 OF 11

```

_id: 0
title: "General Knowledge"
~ questions: Array
  ~ 0: Object
    question: "What is the time?"
    ~ answers: Array
      0: "Now"
      1: "Then"
      2: "Later"
  ~ 1: Object
    question: "What is the day?"
    ~ answers: Array
      0: "Monday"
      1: "Tuesday"
      2: "Wednesday"
      3: "Thursday"

_id: 1
title: "Science"
> questions: Array

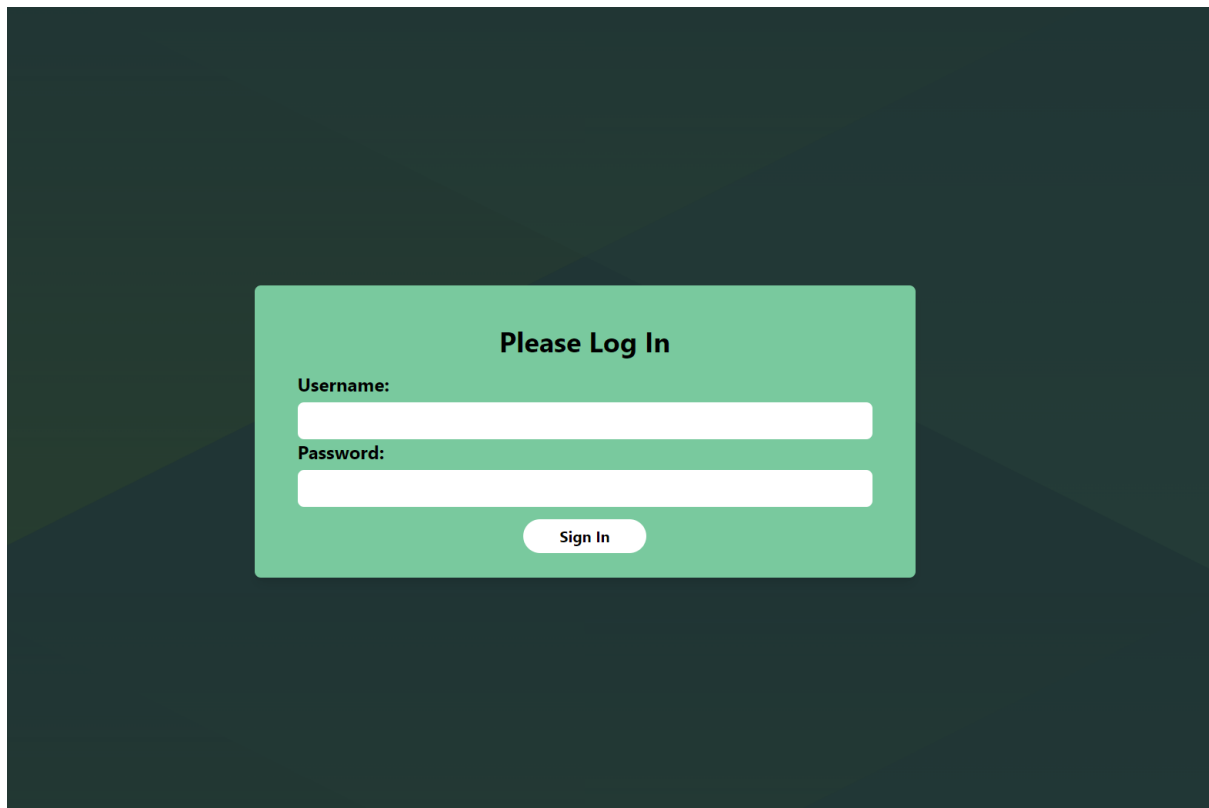
_id: 2
title: "Maths"
> questions: Array

_id: 42947
title: "Long quiz to show scrolling"
> questions: Array

```

Finally, in the above image you can see how my data is stored in MongoDB and what the format used is. The format used in MongoDB is JSON and is another one of the main reasons as to why I chose to use it. After having issues with SQL, I decided using a JSON based format would mean I wouldn't have to spend transforming the data between applications to a format which worked, as the JSON could be handled in the front and back end and my API.

The next section of the document will be used to discuss each of the pages or main features within the application, how these were achieved and why I chose to approach them the way I did.



Screenshot showing login screen of the quiz manager

The first screen the user will be presented with upon entering the site is the Login screen. This is a popup overlay that appears over any other content on the screen. You can see that the login prompt has focus, whilst the background is blacked out. Nothing behind the login prompt is clickable, meaning the only way the user can proceed is by entering the password they have been provided with and pressing the 'Sign In' button.

After consulting with the requirements, I believe this is fit for purpose and achieves what the brief asked for, without over complicating things. The product owner can provide those who need access with the user/name password. There are three different access levels as outlined in the design document: Edit, View and Restricted User, therefore there are three possible passwords to use to log in to the site. The content displayed after logging in will be determined by the password that is entered at this stage.

```

10 <!-- Top level container to darken the background and make contents behind unselectable -->
11 <div v-if="store.isLoggedIn == false" class="absolute flex justify-center items-center bg-black bg-opacity-60 w-screen h-screen z-1000"
12 >
13 </div>
14 <div class="bg-etonblue rounded-lg px-14 w-1/2 py-8 flex-none shadow-lg container absolute z-40">
15   <p class="font-bold text-4xl flex justify-center items-center p-5"> Please Log In </p>
16
17   <!-- Container for login input fields -->
18   <div class="font-bold text-2xl flex flex-col justify-center items-left">
19     <p class="pb-2"> Username: </p>
20     <input type="username" v-model="store.username" class="rounded-lg p-2">
21   </div>
22
23   <div class="font-bold text-2xl flex flex-col justify-center items-left">
24     <p class="pb-2"> Password: </p>
25     <input type="password" v-model="store.password" class="rounded-lg p-2">
26   </div>
27
28   <div class="flex justify-center items-center">
29     <button @click="setAccess()" class="bg-white hover:bg-black hover:text-white py-2 mt-4 w-40 cursor-pointer rounded-full text-center text-xl">
30       Sign In
31     </button>
32   </div>
33 </div>
34

```

The above image shows the code for the component, Login.vue. This component file contains mostly the structure of how the login component will look and displayed on the page. You can see that it is set up as a series of DIVs and inside there is an input and button to allow interactivity. The input uses V-Model which allows two way binding, meaning whatever the user writes in the input field, will be stored in the variable 'store.password' and 'store.username'. The store prefix indicates that these variables are stored in the global Pinia store.

```

export const useStore = defineStore('Store',
{
  state: () =>
  (
    {
      quizList: {} as {[id: number]: Quiz},

      loginModal: ref(true) as Ref,
      quizModal: ref(true) as Ref,
      questionModal: ref(false) as Ref,
      newQuizModal: ref(false) as Ref,
      newQuestionModal: ref(false) as Ref,

      quizToLoad: ref(0) as Ref,
      questionToLoad: ref(0) as Ref,
      numberOfQuizzes: ref() as Ref,

      isLoggedIn: !!localStorage.getItem("loggedin"),

      userRole: ref('') as Ref,
      userPersist: localStorage.setItem('userPersist', ''),
      username: ref('') as Ref,
      password: ref('') as Ref,

      showAnswer: ref(false) as Ref,

      //Initialising question and answer results from new quiz modal
      title: "",
      newQuizID: 0,
      newQuestions: undefined as Question | any,
      newAnswers: [""],
      newQuestionList: [] as any,

      isEditingQuestion: ref(false) as Ref,
    }
  ),

  actions:
  {
    login()
    {
      this.isLoggedIn = true;
      localStorage.setItem("loggedin", "value");
      this.loginModal = false;

      localStorage.setItem('userPersist', this.userRole.toString());
    },

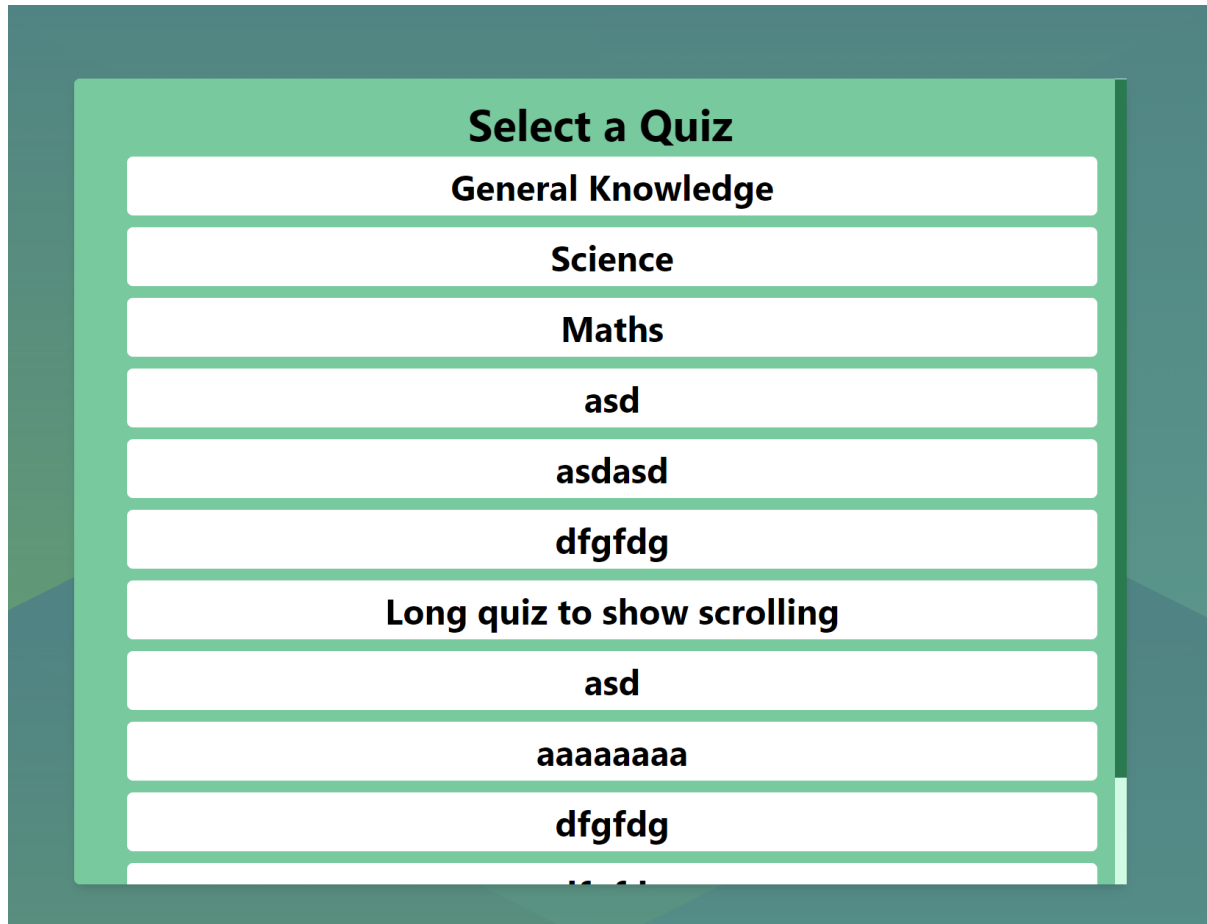
    logout()
    {
      this.isLoggedIn = false;
      this.loginModal = !this.loginModal;
    }
  }
}

```

Screenshot showing the Pinia store file



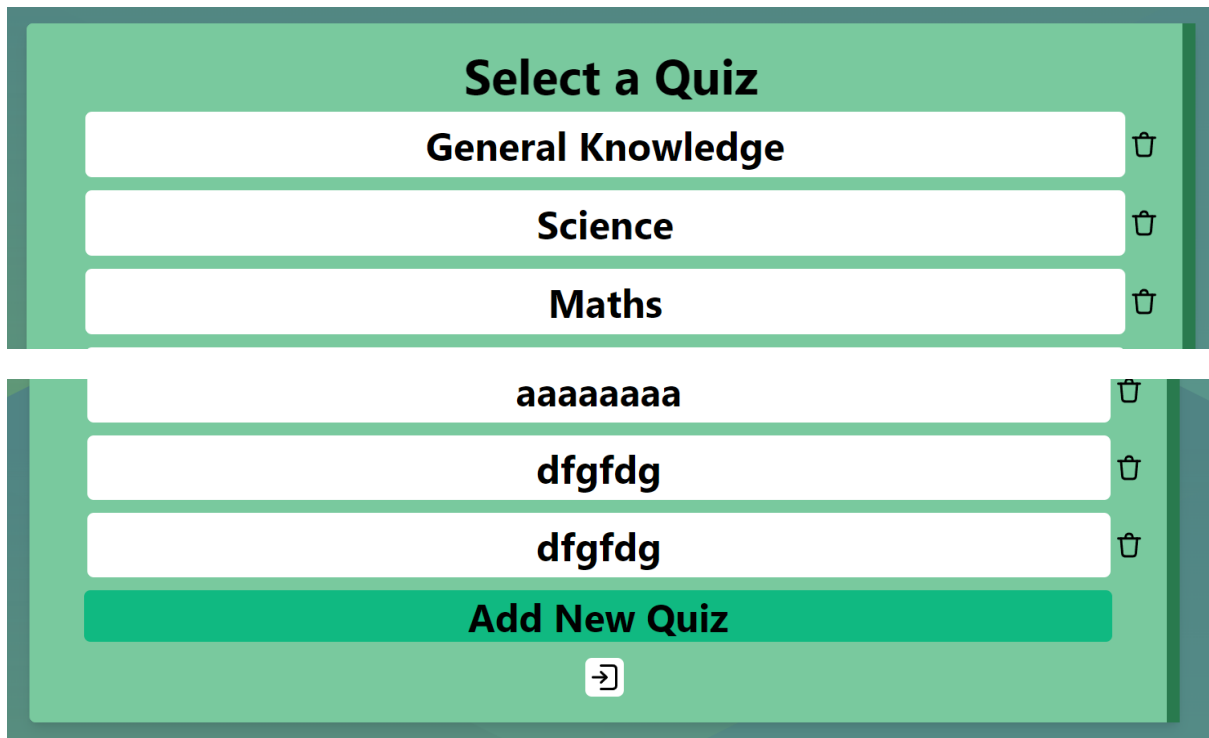
The above screenshot shows the global Pinia store. The purpose of the store is for global state management. Rather than having to pass data between each of the different components multiple times whenever an action is taken on the site, these are all accessibly at any point through any file in the application through the use of the 'store.' Prefix. The store contains both data and functions.



Upon logging into the website, the user will be presented with the above quiz selection screen. You can see that if the list of quizzes is long, a scroll bar will appear inside of the box, rather than the whole screen. This is a personal design choice but I think it looks better than the default browser scroll bar being shown on the entire page.

To proceed to the view the questions in the quiz, the user needs to click one of the options and the corresponding quiz will load immediately.

This screen remains the same for both Restricted and View users. However, as an edit user there are options to add and delete quizzes to/from the list.



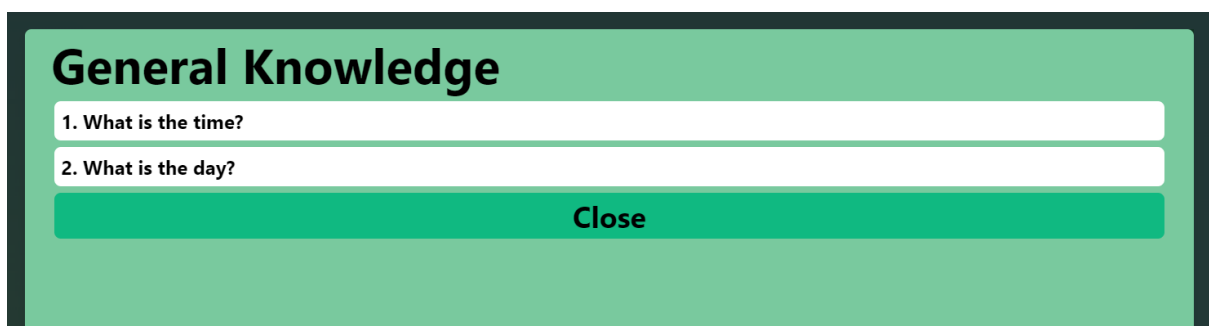
You can see each of the quizzes now has a delete button against it. Clicking this will send a delete request to the API I created, and remove the quiz from the database. From here, a function will run to refresh the quiz list, no longer showing the deleted quiz.

In addition, for Edit users, there is an Add New Quiz button, allowing them to add new quizzes as required.

For all users, there is a logout button at the bottom of the page. Clicking this will log the user out and return them to the login screen.

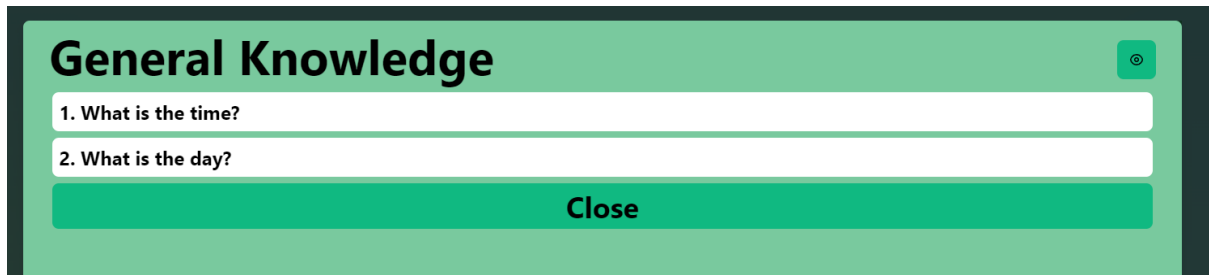
Upon selecting a quiz, all users will be taken to the questions screen.

For Restricted users, they are only able to see the questions as outlined by the specification requirements.



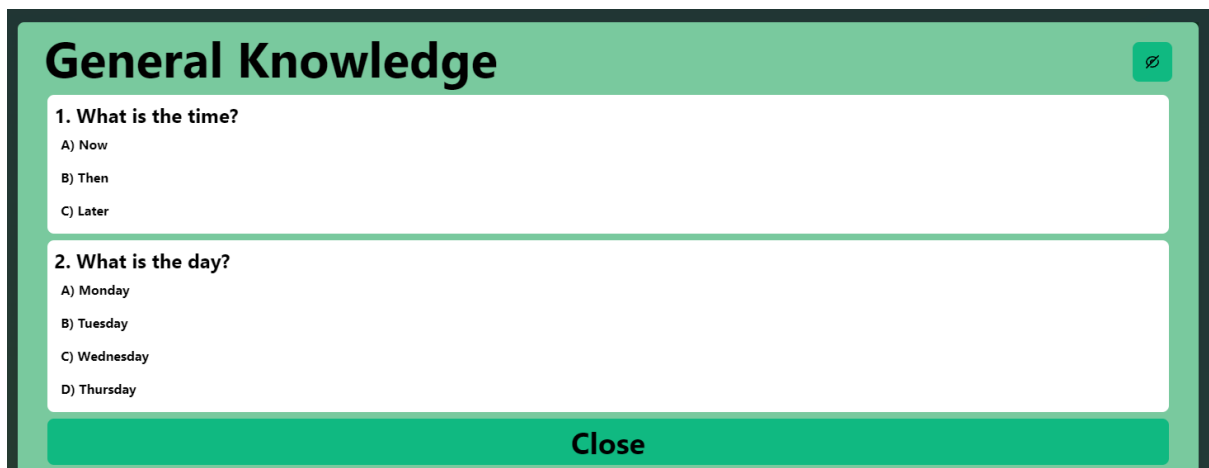
The close button will return the user to the quiz selection list.

When logged in as a user with View permissions, they will be presented with an icon at the top right of the screen. Clicking this will expand all of the questions in the quiz and display each answer to each question.

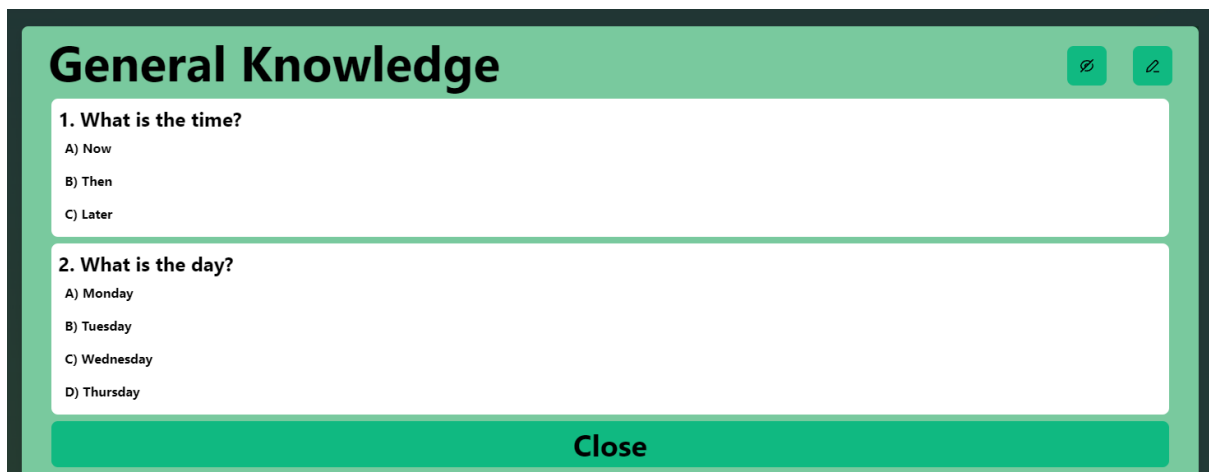


The questions are indexed with a number and each new question added or deleted will change the index of the remaining questions.

When expanding the questions to show the answers, each answer is index from letters A – E. Each quiz can have from 3-5 answers and this is handled by the code. I will discuss this further in the document.



When logged in as a user with Edit permissions, they will have access to the above functionality, but also the capability to add/delete questions and amend existing ones.



Clicking the edit icon at the top right will bring up a new modal with all of the questions in that quiz and the options to make amendments.

**Create/Update a Quiz**

**Quiz Title:**

**Question:**

**Answers:**

**Delete Question**

**Add Question**

**Question:**

**Answers:**

**Delete Question**

**Add Question**

**Submit**

**Cancel**

As you can see, the modal shows a series of input fields, which are binded to the values of the existing quiz. Making changes to these and pressing submit will send a PUT request to the API to update the data.

Clicking the Add Question button will bring up a new modal allowing the Edit user to add new questions.

The screenshot shows a web application interface for creating or updating a quiz. In the background, there's a 'General Knowledge' quiz titled 'General Knowledge' with questions like '1. What is the time?' and '2. What is the day?'. Overlaid on this is a modal titled 'Add New Question'. The modal contains a 'Quiz Title' field with 'General Knowledge' entered, a 'Question' field with 'What is the time?', and an 'Answers' section with three input fields. Each answer field has a red 'X' icon and a green '+' icon to its right. At the bottom of the modal are 'Save' and 'Cancel' buttons. Below the modal, there are buttons for 'Delete Question' and 'Add Question'.

Each question must have a minimum of three answers, but a maximum of 5. This is handled using the icons to the right of the answer input fields.

This is a detailed view of the 'Add New Question' modal. It features a large title 'Add New Question' at the top. Below the title is a text input field for the question, preceded by the label 'Enter the Question:'. Underneath is a section for answers, labeled 'Enter the Answers:'. This section contains four horizontal input fields. To the right of each input field are two circular icons: a red one with a white 'X' and a green one with a white '+'. At the bottom of the modal are two large buttons: a green 'Save' button and a red 'Cancel' button.

# Add New Question

**Enter the Question:**

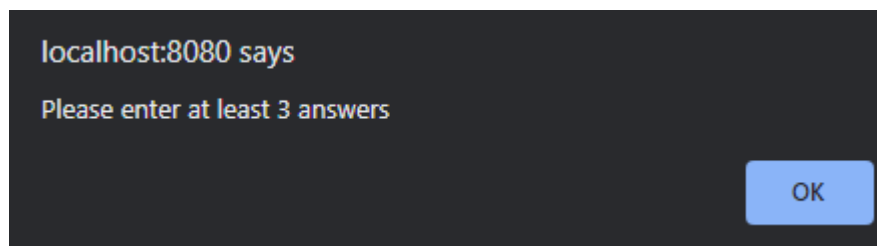
**Enter the Answers:**

**Save**

**Cancel**

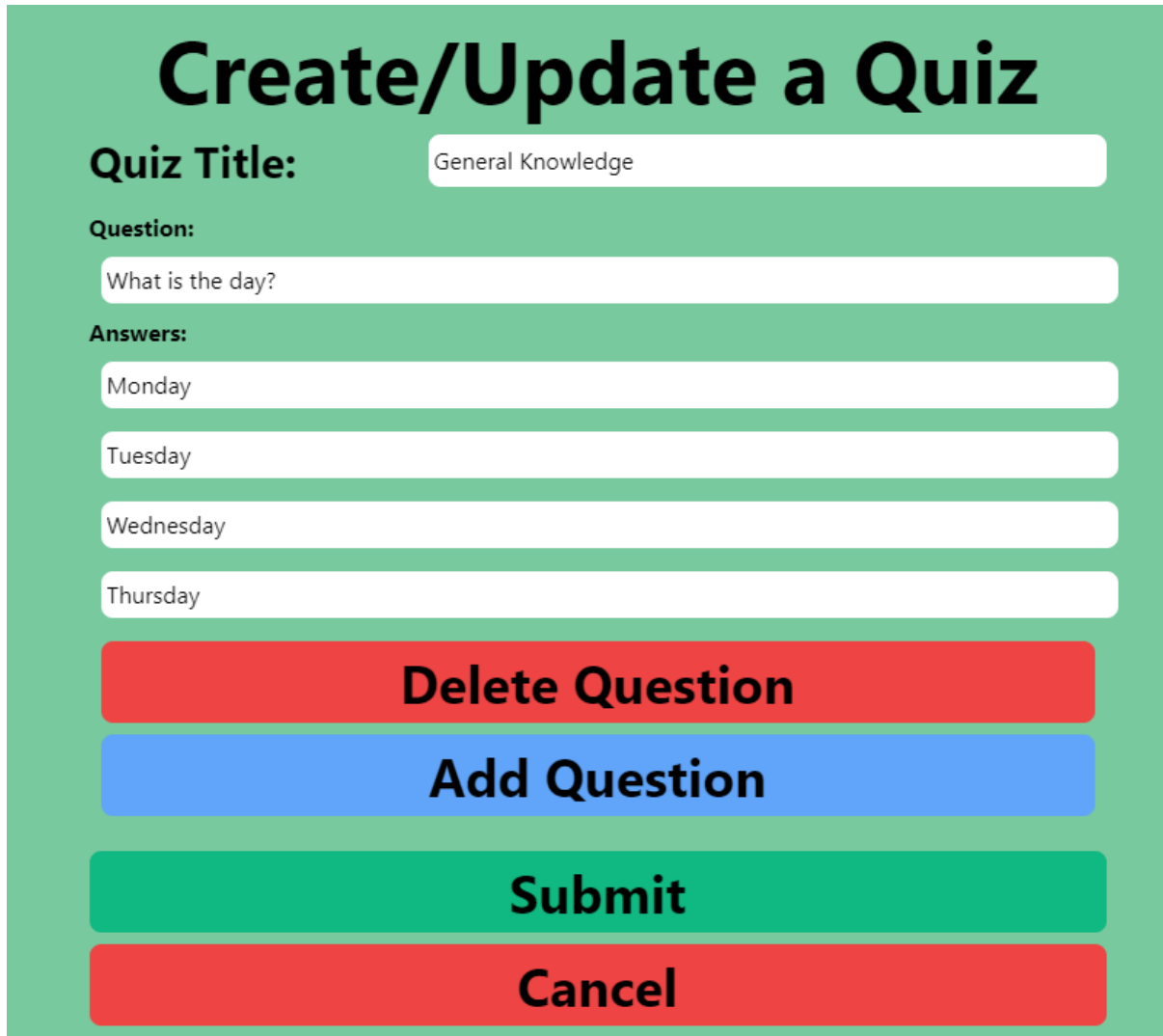
The two images above show that if there are three answer fields, a + icon will appear to allow the user to add new fields. Once the number of input fields reaches 5, the + icon will disappear and only an X icon will remain, allowing the user to remove input fields.

If the user tries to submit a question without a title or answers they will be presented with a window alert requiring them to fill out the necessary fields.



Once the save button is clicked, it will return a user to the Create/Update Quiz modal.

If the user clicks the Delete Question button, it will remove that question and corresponding answers from the list.

A modal form titled "Create/Update a Quiz" with a green background. It contains input fields for "Quiz Title" (with "General Knowledge" entered), "Question:" (with "What is the day?"), and "Answers:" (with "Monday", "Tuesday", "Wednesday", and "Thursday" entered). At the bottom are four buttons: "Delete Question" (red), "Add Question" (blue), "Submit" (green), and "Cancel" (red).

## Create/Update a Quiz

**Quiz Title:**

**Question:**

**Answers:**

**Delete Question**

**Add Question**

**Submit**

**Cancel**

Back on the Select a Quiz screen, if the user wishes to add a new quiz, the same modal above will appear. However, rather than grabbing the data from the existing quiz, they will be presented with a blank Create/Update a Quiz modal as seen below.

# Create/Update a Quiz

Quiz Title:

**Add Question**

**Submit**

**Cancel**

The previous requirements remain, they will have to enter questions and answers before they can submit the quiz.

Once submitted, this will send a POST request to the API, which will in turn update the database.



```

getQuizzes(): void
{
    fetch(apiURL,
        {
            headers: { Accept: 'application/json' },
        })
        .then(response => {
            return response.json()
        })
        .then(data =>
        {
            let newQuiz: Quiz;
            data.forEach((quiz: any) =>
            {
                newQuiz = new Quiz(quiz.id, quiz.title, quiz.questions);
                this.quizList[newQuiz.id] = newQuiz;
            })
        })
    },

postQuiz(quiz: Quiz): void
{
    console.log(JSON.stringify(quiz).toString())

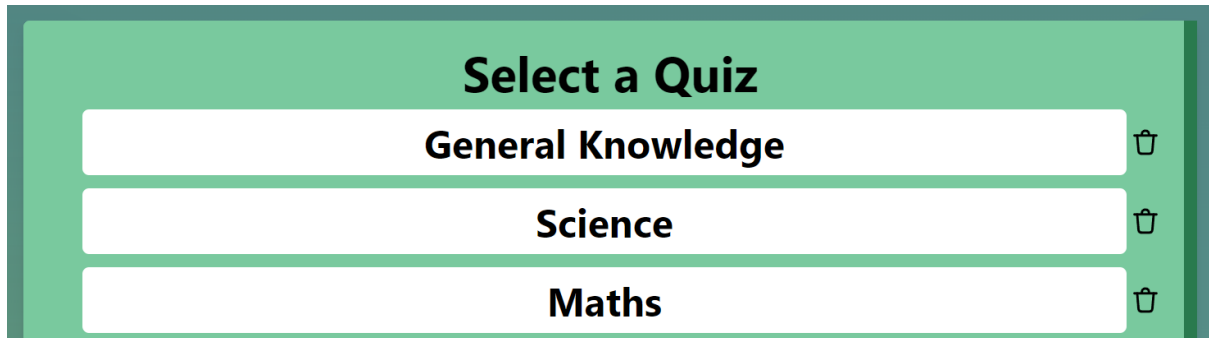
    fetch(apiURL,
        {
            method: 'POST',
            headers:
            {
                "Content-Type": "application/json",
                "Accept": "application/json",
            },
            body: JSON.stringify(quiz).toString()
        })
    this.getQuizzes();
},

putQuiz(quiz: Quiz):void
{
    fetch(apiURL + quiz.id,
        {
            method: 'PUT',
            headers:
            {
                "Content-Type": "application/json",
                "Accept": "application/json",
            },
            body: JSON.stringify(quiz).toString()
        })
    this.getQuizzes();
},

```

The above screenshot shows the API calls for GET, POST and PUT.

These API calls are done using the Fetch API. I decided to use Fetch as it is very simple and quick to use.

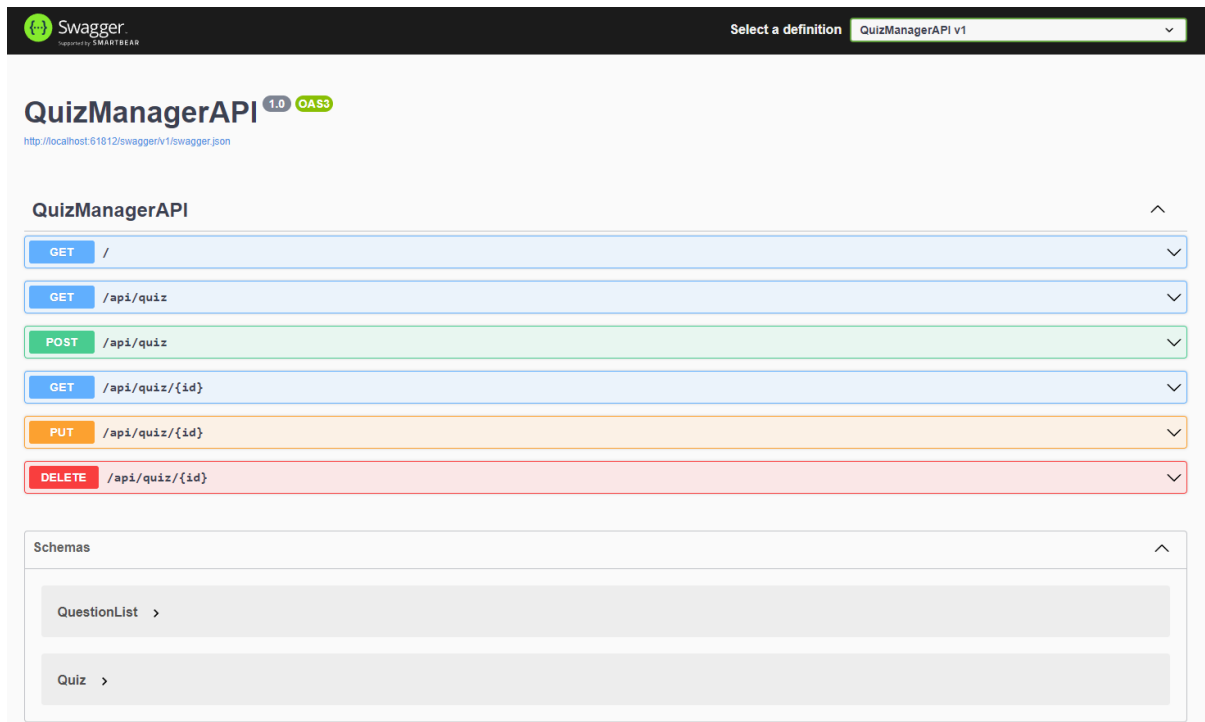


Deleting quizzes from the list is done on the Select a Quiz screen and again, can only be done by a user with Edit permissions. The user must click the Bin/Trashcan icon to delete the desired quiz.

Once this clicked, it calls a function, which in turn sends a DELETE request to the API via Fetch.

```
deleteQuiz(id: number):void
{
  fetch(apiURL + id,
    {
      method: 'DELETE'
    })
},
```

The request is passed the ID of the selected quiz and then appends this to the API URL and sends as a DELETE request.



The screenshot above shows the API open in Swagger. The API is built using the OpenAPI standard and it shows all of the endpoints that have been made available to my API, as well as their functionality.

**Limitations, Future Improvements and Final Reflection**

Overall, I am very pleased with how this project turned out and the results I was able to deliver. Although the requirements were brief and vague, I have taken these requirements and developed them into an appropriate solution.

The brief in its entirety can be seen as follows:

*You work for WebbiSkools Ltd, a software company that provides on-line educational solutions for education establishments and training providers. Your manager would like you to design, build, and test a database-driven website to manage quizzes, each consisting of a set of multiple-choice questions and their associated answers. The website's capabilities will only be accessible to known users. Users with full permissions will be able to view and edit the questions and answers; users with lesser permissions will be able to view them but not edit them; users with minimal permissions will only be able to see the questions.*

As illustrated earlier in the document I have successfully created a quiz management application. This website is database driven and contains several quizzes, each consisting of a set of multiple-choice questions and their associated answers. In addition to this, I have implemented the ability for users with full permissions to edit the questions and answers. I have also created a user access level that allows a user to view the answers but not edit the questions or answers. Finally, I have added a level that allows only the questions and answers to be viewed and the quiz to be completed with no amendments.

One main limitation to the application is the login system. I know that as a login system, it is not particularly secure, sophisticated or scalable in terms of the number of users on the system. In the future I would like to implement a proper login system allowing individual user accounts, managing the passwords better by storing them as hashes in the database, rather than as hardcoded text values in the code etc.

There are other several improvements I would like to add in the future, which are not necessarily limitations of the system. For example, I would like to add a dark mode or a theme selector option which would allow the user to choose from a greater variety of colour schemes for the website. This could easily be done in Tailwind and is something I would look into adding at a later date.

In addition, I would like to add leader board and the capability for users to play/take the quizzes. I think it would make for a more engaging user experience and a more complete solution.

Finally, I would like to add a timer limit to how long the user has to be able to answer each question. Currently they have unlimited time to answer questions, but I think it would be more enjoyable as a quiz experience if there was a timer visible on-screen putting users under pressure to choose an answer quickly.

There are several bugs within the system which I am aware of and could not resolve due to time constraints. Currently there is an issue with refreshing the quiz list when a quiz is deleted. When submitting/amending a quiz, a function is called which updates the quiz list

accordingly. For some reason, this function would not work with the delete operation and as a workaround I have to force the page to re-load which is not ideal.

In addition, there are issues with the indexing of newly added quizzes, which again I did not have the time to resolve. When adding new answers to a question, these can be added/amended at any position and the index updates accordingly. However, this functionality would not quite work correctly with the quizzes.

To conclude, I am happy with how this project went. As with any project, there are improvements that can be made when looking back in hindsight but I feel confident I have met the majority of the user requirements and delivered a solid product with the time I was given.