Quizzy Evaluation

Contents

Project Outline:

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.

Introduction

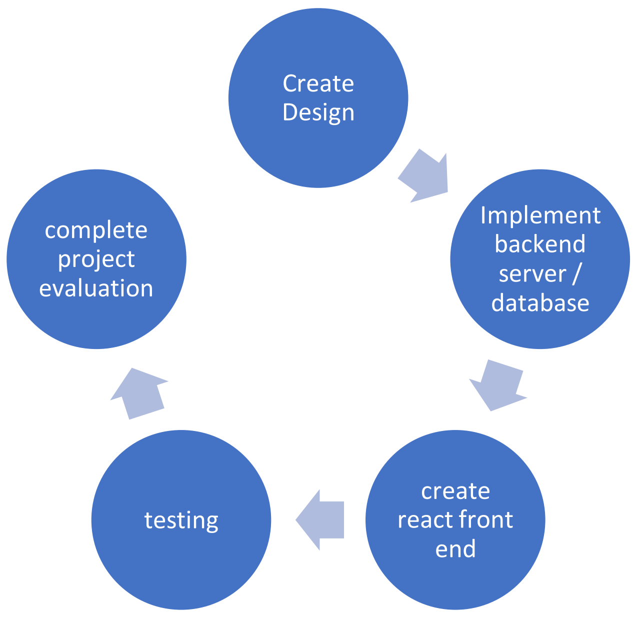
I have now implemented a code freeze in order to review work completed. Below I will outline the key deliverables and outline requirements and discuss how I implemented them along with any design limitations and future improvements.

Key Deliverables:

1. Design Documentation
2. Constructed program
3. Tests
4. Evaluation document

Software Development Lifecycle:

I attempted to strongly adhere to the sdlc I am accustomed to following at my workplace. See below for the SDLC I followed.


Create Design
Implement backend server / database
create react front end
testing
complete project evaluation

Requirements:

Key requirement changes:

I included some additional requirement in my design documentation. I disagreed with the requirement that restricted users should only be able to view quizzes with questions without any kind of interaction or feedback. I saw no value in creating the ability for pupils to log in if there is essentially nothing for them to do on the website. As someone who has come from an education background, without some kind of meaningful functionality, pupils would simply choose not to access this resource if it is not interesting. As such I added an additional requirement that users should be able to answer quizzes and receive feedback on their answers. This partially aligned with the original requirement as pupils could still see quizzes, questions and answers as they answered each question. They would then be prompted with the correct answer after submitting a response. This implementation is a much nicer user experience, ensuring that the resources on Quizzy are kept engaging and meaningful.

Lastly as a nice to have, I proposed that users with Edit access (teachers) should have the ability to create new users for Quizzy. As stated in my design document class structure changes all time, and a teacher may have more than one assistant in their class, or many pupils wanting to access Quizzy at the same time. With this in mind I created a create user endpoint for teachers to access. This will keep Quizzy accessible for all that require it.

Below I have listed out the requirements by the four main groups and an brief outline of how I implemented them.

Quiz Requirements

Requirement	Achieved?	Implementation
Quizzes to be stored in a database	Fully	Mongodb quiz collection. CRUD operations used. Quiz mongoose schema to organise data.
Each quiz has a title and multiple-choice answers	fully	Questions are linked by title. I limited each question to four possible answers and one correct answer, this was a limit of using a mongo schema as I had to outline each separate fields upfront. Each quiz, however, could have an unlimited number of questions.
All questions should be viewed on a single page with indexed answers.	Fully	As per requirements all users with Edit and Read rights could see a numbered list of questions by quiz name, with indexed answers.

User Requirements

Requirement	Achieved?	Implementation
Users should be pre-configured	Full	I created a pupil, assistant and teacher login. The details for which are included in the guide document.
Session State	Fully	Login state maintained by cookies that expire when a user either logs out or navigates away from Quizzy. Users cannot access Quizzy without logging in with a jwt token, which again will expire after a period of 6 hours. Login state checked on each page.
Only known users can login to the website	fully	If user navigates to any point of the website without logging in, they are redirected to login page. All endpoints are authenticated, with no way to create new users without logging in first even if api endpoints are exposed.
Passwords should be hashed	fully	I used bcrypt to store all passwords. Plain text passwords are not stored anywhere in my application.
Unrestricted users can edit the test of any answer	fully	Unrestricted users can enter a new value into any quiz field. They can update just one field or multiple.
Add and delete answers to any question (which may cause the answers to be re-indexed)	Partially	This was an oversight on my part which I realised after revieing my work. Questions can be created and are re-indexed upon new question creation. However, there is no implementation to delete single answers, only update the value. My model also restrict the number of possible answers to 4.

Access Requirements:

For these requirements each user has an associated 'access level'. I created the levels are follows:

Access level:

1. Edit: User able to edit, read, update and delete all quizzes. I called this group 'teachers'.
2. Read: User able to read lists of all quizzes and answers. I called this group 'assistants'.
3. Restricted: Only access to questions and potential answers.

Requirement	Achieved?	Implementation
Edit	Fully	A user with level 3 access can complete all CRUD operations in the backend of Quizzy. Level 3 users have unrestricted navigation of the Quizzy user interface.
View	Fully	Level 2 users can navigate to the view quiz's view. They can select a quiz by name and see the associated questions and answers.

		If a level 2 user attempts a create, update or delete operation via the app a 401 is returned.
Restricted	Fully	Level 1 users can only navigate to the home and logout page in the user interface. The management options are not visible and if they navigate manually to management views the user is immediately redirected back to home. The home page enabled users to take a quiz, seeing questions and potential answers. Answers not exposed until after potential answer selected.

Best Practice Requirements:

Requirement	Achieved?	Implantation
Design should be re-brandable	Partially	My styling is saved in a CSS file for ease of future changes The navbar is a separate component with separate CSS file, can be easily swapped out. Would still be a lot of work to change the overall design, maybe I would have implemented something to make this easier.
Code should conform to industry best practice	Partially	I implemented DRY principles, creating auth for login, teachers and assistants separately. Each react component had its own responsibility with clear single responsibility functions within. For example, "handleRedirect". With more time I could have refactored further to align further with SOLID principles. My api was created RESTfully.

Test Requirements

Requirement	Achieved?	Implementation
Manual or automated testing	Fully	Prior to testing I created a test plan. I thoroughly manually tested my program, testing all api endpoints in postman and via the UI. I have included a video walkthrough of some sample testing as evidence. I began adding unit tests but ran out of time. Testing the authenticated endpoints held me up here.
Tests may be written before, after or at the same time as the program code	Fully	I manually tested my code as I created it. Before pushing each commit I tested the UI to ensure everything worked as expected.

Optional Nice to have features:

Optional Requirement	Achieved?	Implementation
Taking a quiz	Fully	Any logged in user is able to run through a quiz and answer questions.
User creation	Fully	Teachers only can create new users.
Editing users	No	Being non-MVP I prioritised other features.
Feedback for pupils	Fully	Once a question has been answered answer feedback is provided
Unit tests	No	I attempted to implement jest unit tests, but ran out of time.

Tech Stack used:

Backend

Node Js	I used node to create an event driven server for the 'backend' handling on Quizzy.
Express JS	Express made handling my http requests straightforward and highly customisable.
Validator	Validator is a npm library that sanitises and validates strings. I'll be using it on my models to ensure the data entering my database is in the correct format.
mongoose	Mongoose is an object data modelling library Mongodb,(a NoSql database manager)
mongoDb	MongoDB is a database that stores data as documents. Most commonly these documents resemble a JSON-like structure.
Bcrypt	Bcrypt is a password-hashing function.

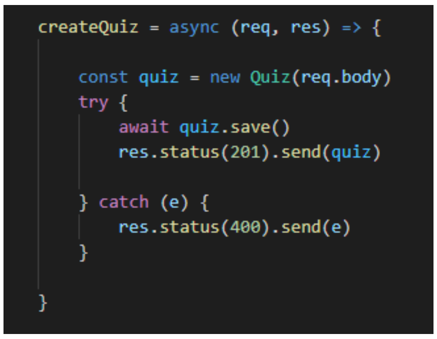
Frontend

React JS	React enabled me to create a reactive client side UI with single responsibility components that are rendered onto a single HTML page.
CSS	Styling sheet
Axios	Axios is a promise-based HTTP client for the browser and Node. js. Axios makes it easy to send asynchronous HTTP requests to REST endpoints and perform CRUD operations. I will be using it to connect my backend to my react front.

Development Outcome

Backend development

I began by creating a REST api with Node Express. Using Express meant that my backend was reasonably easy to configure and customize and I was able to keep the server side of my application completely separate from client side. I found endpoint creation fairly straightforward as I was able to use the structure of the HTTP requests in both the user and quiz endpoints. For example, create user and create quiz had the same structure. Therefore, when creating the user endpoint, I could use the quiz http request to help troubleshoot.

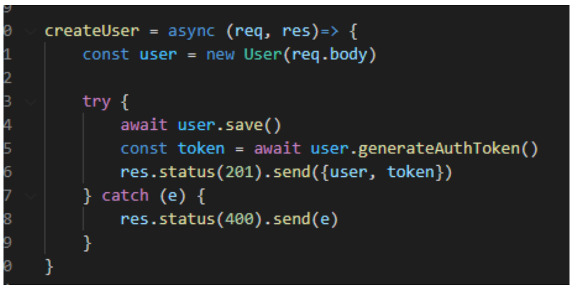


```
createQuiz = async (req, res) => {

    const quiz = new Quiz(req.body)
    try {
        await quiz.save()
        res.status(201).send(quiz)

    } catch (e) {
        res.status(400).send(e)
    }

}
```



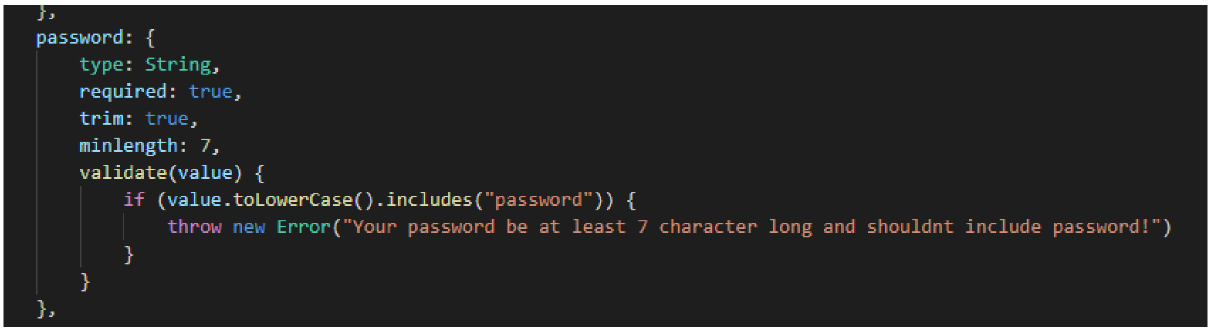
```
createUser = async (req, res)=> {
    const user = new User(req.body)

    try {
        await user.save()
        const token = await user.generateAuthToken()
        res.status(201).send({user, token})
    } catch (e) {
        res.status(400).send(e)
    }
}
```

This saved a lot of development time, which was a huge benefit in a time limited project. Looking at the above examples If I had had further time, I would have liked to have investigated refactoring my quiz and user controllers. As stated, a benefit was that a few methods were very similar, therefore could probably be refactored into one create, one read etc method which would be more aligned with DRY principles.

A key requirement was for my website to be database driven, to fit this purpose I used mongoDb, with a mongoose extension within my project. Mongoose allows you to define objects with a strongly typed schema that is mapped to a MongoDB document.

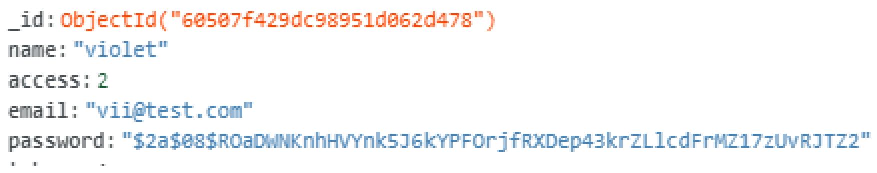
I had a separate user and quiz schema. Using schemas enabled me to tightly control what information was being passed to my database and provide some validation. For example, for quizzes I didn’t want a quiz to be created without filling out all forms so I was able to mark all quiz attributes as ‘required’, thus ensuring the request would fail if each property was not passed in. I was also able to use a handy node package called validator to ensure when creating users that the password conformed to a set of rules, thus increasing Quizzy's overall security.



```
    },
    password: {
        type: String,
        required: true,
        trim: true,
        minlength: 7,
        validate(value) {
            if (value.toLowerCase().includes("password")) {
                throw new Error("Your password be at least 7 character long and shouldnt include password!")
            }
        }
    },
```

One issue that I encountered was that I created a strict schema with separate fields for only four possible answers. Upon project review I recalled that questions should have unlimited number of potential answers which is simply not implemented in my model. In order to fix this, I would have to change my schema and create an answer endpoint that looks for the questions selected then adds the new question to that object. This would then be listed out without a problem. I could have added this If I had noticed my mistake less close to the end of the deadline!

Mongoose has a handy feature that allows behaviour to be executed prior to the save function. With this I was able to encrypt any incoming passwords on the user schema so that no passwords were stored as text only. This fulfilled one of the primary requirements.

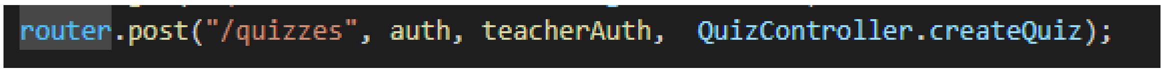


```
_id: ObjectId("60507f429dc98951d062d478")
name: "violet"
access: 2
email: "vii@test.com"
password: "$2a$08$ROaDWNKnhHVYnk5J6kYPFOrjfRXDep43krZLlcdFrMZ17zUvRJTZ2"
```

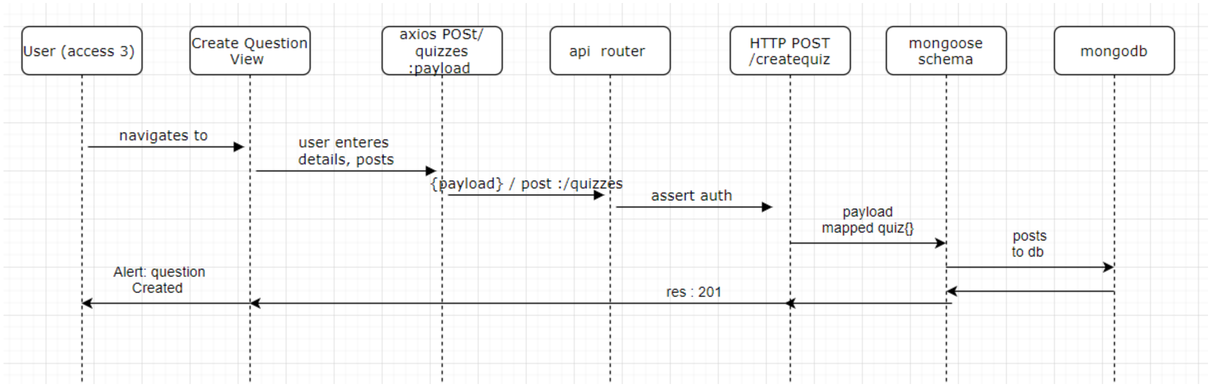
At this point I tested all http methods in postman, which I found to be an incredibly useful tool. It highlighted an error coming from the mongo database. This cost me a couple of hours of debugging to figure out. I eventually identified from console logging error messages that the error was of my creation – I had setup new questions with the property of unique and then not handled the corresponding error. Having unique questions is desirable, however having timeboxed my problem to 2 hours I decided to remove the 'unique' declaration and implement it at the end of my allotted time, with correct error handling at the end, which would have to be implemented client side. Inevitably I ran out of time and was not able to complete this, so at present users are able to create identical questions which could cause confusion in the future. If I had more time, I would create a check when creating a new question to see if that data is already stored and prompt the user to change their input if duplicate questions are found. This is a minor problem and shouldn't affect the user experience most of the time.

During planning I had decided to address the access requirement via assigning each user a number based on their access level, eg 1 for restricted, 2 for read only, and this would need to be checked before each http request was made. This number would be created when the user was initially posted and would not be changeable. I decided to create two middleware functions to validate the access number, one for the assistant level access and one for the teachers. The restricted user group would not need its own auth middleware as all endpoints except getquizzes would either check for a 2 or 3, depending on the method so pupils would automatically be excluded form accessing this. Creating this functionality arguably complied with the tiered access requirement, but had not been in my original plan. During creation of the UI I identified that if a restricted user managed to manipulate their way to a form they shouldn't have access to, they could still potentially post / retrieve data they weren't meant to. So, adding in this extra fail safe would ensure no requests by unauthorised users would be possible.

Lastly, I implemented the login auth middleware. This would be responsible for ensuring the login state is maintained, which was a requirement. In this I generated a jwt token and saved it to the user object as it was posted to the database. I would then use this to verify login before allowing other database interactions, such as retrieving quizzes. At present this is saved in a cookie for easy retrieval. Ideally, I would change this to make an api call to get the current user object and check the token matches to a prop variable, thus never exposed and far more secure, but the current solution fits the purpose for now.



```
router.post("/quizzes", auth, teacherAuth,  QuizController.createQuiz);
```



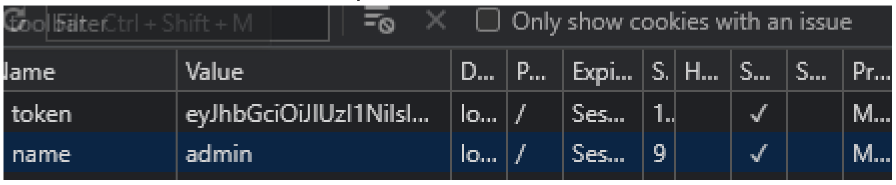
This concluded the development of my backend code and would enable me to call the http endpoints from my user interface. This stage went reasonably smoothly with only one major setback which cost me unnecessary time. If this had been a full development team with multiple people, I am confident that the problem would have been spotted far sooner. One of the problems with one person only development is that there are no 'fresh eyes' available. This is one of the reasons my company enforces a strict code review policy, that 1 – 2 people who have not been involved in the project must review all pull requests. This is something I would have strongly adhered to if possible, as having external viewpoints and code scrutiny would have increased the quality of the outcome of this section.

I am also certain that external viewpoints would have provided insight into better ways of working. I am accustomed to pair-programming in my current role and know first-hand how having someone to bounce ideas off often alters the course of action for the better.

Front End Development

I chose to use React js to develop my user interface. A requirement was that the website should be re-brandable, and therefore easy for developers to change. React uses a single index page, which components are rendered into based upon the route selected. This makes the website highly customisable and straight forward (in theory) to change. Each page had its own methods with each method having a single responsibility in most cases.

I found the original setup of react confusing and time consuming and had some difficulties passing data between the separate components. This means each page completely aligns with the single responsibility principle but was not useful when I needed access to global variables such as the user's name or log in state. I time boxed attempting to get props to work between components, so that I could pass the jwt for login authentication without exposing it to the browser. Sadly, once the timebox was complete I had still not reliably achieved this so the decision was taken to use JS Cookies to store the necessary items as session-based cookies.



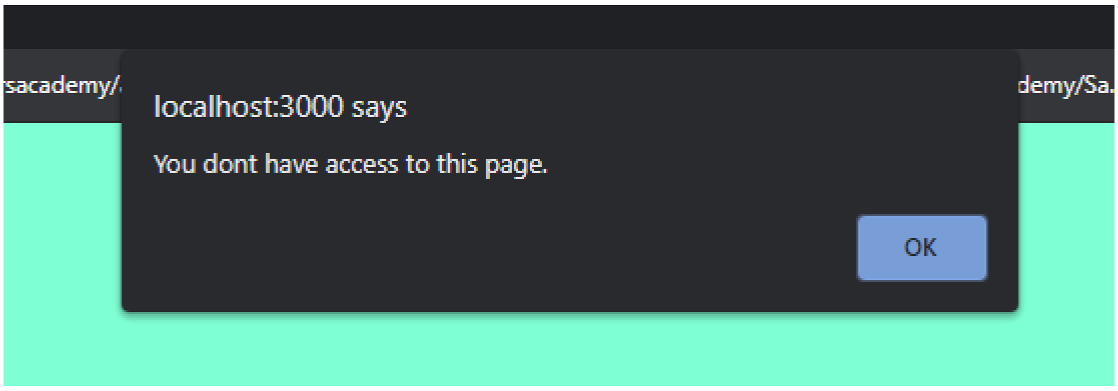
lame	Value	D...	P...	Expi...	S.	H...	S...	S...	Pr...
token	eyJhbGciOiJIUzI1NiIsI...	lo...	/	Ses...	1..		✓		M...
name	admin	lo...	/	Ses...	9		✓		M...

I implemented checks within my UI to ensure these session cookies were present, and if they weren't the user would be taken back to a point that did not require that authentication, such as the login page. Thus, fulfilling the session state requirement.

I acknowledge that this in live would not be a sufficient solution as it potentially exposes the token to attacks. However usually the client secret would also be stored in an environment variable, which would again help prevent security vulnerability. I would like to implement a system of using REDUX, as I believe that would have solved this problem. Redux with hooks give function components the ability to use local component state, which would have solved the difficulties with passing the token between components. I believe it would have taken me 2 -3 days to understand how to implement this into my Quizzy project, which I simply didn't have time for. This is certainly an improvement I would like to implement, for my own development and to improve the functionality of Quizzy.

Continuing on about access, I spoke previously how the api endpoints are all authenticated as a failsafe I found that alone was a poor user experience as users in the assistants' group were able to navigate all the way to the create a quiz page, only for the quiz to not be created with no indication to the users. As such I created a function to check the access level of the user before navigating them to the edit / create views.

Assistants can see the management view -but cannot get to the form to make changes.


localhost:3000 says

You dont have access to this page.

This is a much more transparent solution to access. Ideally, I would have liked to make the create and edit buttons only visible to teachers.

The other management views were created flexibly and will respond to any change in the website without needing too much intervention, which will make rebranding the website more straightforward.

To conclude the front-end development this was by far the most challenging aspect in this sdlc so far. In my current role I very rarely touch front end development, so I learnt react specifically for this challenge when I realised using pure JS/HTML was going to be complicated. In hindsight using a newly learnt tech stack under a time frame was always going to be a challenge but it's been one that I've enjoyed, and I look forward to further cementing my own knowledge on with personal projects. As stated in particular I would like to get to grips with passing props, which really is an essential part of react. In my pre-quiz-manager practice I was able to get this working on a task-app and it was frustrating that I could not implement this for Quizzy. I am sure I will look back and see the problem clearly once the time pressure is relieved and Quizzy submitted! To the average user this shortfall will neither be noticeable or impact their interactions as Quizzy is fully functional and performs well despite this.

Testing

I was able to fully manual test the backend and front end of quizzy via postman and the user interface. I have uploaded a recording of some manual testing which I have included in a separate folder. The test cases I have included in the next section. To run the tests I did create enter 'npm run test' into the terminal.

I was not able to complete unit testing with jest as planned as I simply ran out of time. With another day I would probably have been able to implement this.

Test Plan

Here I will outline my proposed manual testing plan. See below for my outline test cases.

Pupil Test Plan:

- Test that Pre-existing pupil can log in
- Pupil can take a quiz
    - Pupil can receive feedback on incorrect answers
- Pupil does not have access to the answers until after a question is answered
- Pupil can log out and session data is cleared

Assistant Test Plan:

- Test that Pre-existing assistant can log in
- Can read quizzes and answers
- Assistants should not be able to create quizzes
- Assistant should not be able to edit or delete questions
- Assistant can log out

Teacher Test Plan

- Teachers should be able to log in with a pre-existing account
- Teachers can read existing quizzes
- Teachers can create new questions
- Teachers can update and delete questions
- Teacher can log out

Developer Test Plan

- Passwords should be hashed for privacy
- Restricted endpoints should have authentication
- User interface should interact with mongo database
    - New questions should be added to the quiz collection
    - New users should be added to the users collection
- Login is not allowed with incorrect credentials

Development conclusion:

It was a tight timeframe to design, implement, test and review a full stack application. Time available implemented the overall quality of the program deliverable, however as stated the MVP was achieved along with some of the additional requirements.

There are many outstanding features that I would have liked to implement, such as:

- completion of the testing
- refactoring the program to not rely on cookies for login
- refactoring to further align with SOLID principles, a lot of my components could use this
- fixing missed requirement that answers can be deleted one by one,
- allowing unlimited answers to questions
- I would like to move certain variables, such as the port number, client secret and mongoDB string to an environment variable file for security
- allowing entire quizzes to be deleted at once, not just questions by question

The overall outcome of this section of work was a fully functional application that met the MVP outlined in the project outline. Pupils can log in and take a quiz, only seeing the answers once answering. Assistants can login and view a list of numbered questions and answers and teachers can complete all CRUD operations in relation to quizzes. All logins are authenticated, and the login token is cleared upon user logout. I was able to follow DRY, SOLID and rest principles when designing Quizzy. I am proud of how far I managed to take the development of Quizzy during the hours allocated.

User guide

I have a created a user guide which can be found in the project readme:

https://github.com/makersacademy/Sarah-Wade-SP/blob/main/README.md