Design and Implementation

Here I document some of the logic I used when designing and creating the app. I also include a list of user stories and work items I created from the design requirements I used to guide me in building and delivering this application, as well as a list of blockers and design choices that I made along the way, as well as some of the outcomes those blockers had - such as requiring alterations to designs of datastructure and refactoring the code.
At the start of this project I decided to use AWS architecture to deliver the backend and react to deliver the front for several reasons, first of which is my previous familiarity with these languages and architectures would enable me to hit the ground running. It has been some 18 months since I have been on a full-stack development team, and recent experience in other technology stacks are so narrow I was not confident I would be able to deliver a functional app in the 40 hours given.

I chose React because there are a wide choice of libraries that can be used to deliver basic functionality and GUI design that allow for much greater speed while coding. It is a more modern and developer friendly approach to creating browser facing frontends with a layout that is widely familiar to anyone who has used any electronic device, and that also allow for seamless scaling from a wide computer monitor down to a small mobile phone screen and still provide functionality.

I Chose AWS as the backend to service the needs of my webapp, and to demonstrate cloud based development skills. While the requirement of the project being accessible to the examiner left a security vulnerability (discussed in the AWS documentation) a real world web app fix to that particular vulnerability would be simple to implement, and is recorded. It also allowed me to use DynamoDB, which is my preferred database solution because of my familiarity with it, though a lack of recent experience working with noSQL databases did lead to a blocker, requiring refactoring the database to a new datastructure.

A large amount of the final product was not delivered in this project for various reasons, especially my struggle to overcome some of the blockers I encountered. While I am happy to say I can deliver a very crudely rendered vertical slice with good, clean underlying code, I am unfortunately unable to deliver all the requirements. While adding some of these would be straight forwards, such as displaying correct answers, adding buttons to select answers, some of them are not so trivial such as building a GUI for administration level users. The next steps in development, had I the time, were going to be

User Stories;
As a View level user I want to be able to see a quiz displayed with the correct answers labeled.
As a Restricted level user I want to be able to see a quiz displayed without the correct answers labeled.
As an edit level user I want to be able to add questions to an existing quiz.
As an edit level user I want to be able to edit the answers on an existing question.
As an edit level user I want to be able to remove questions from an existing quiz.
As an edit level user I want to be able to create a new quiz.
As an edit level user I want to be able to delete an existing quiz.
As a user I want to be able to log out.

Work tasks; (✔) *Completed* (-) *In progress* ( ) *Not started* ***High Importance*** *~~removed~~*
(✔) Create a webapp UX flowchart
(✔) **Create basic GUI design**
(✔) (*Write documentation for the UI/UX)*
(✔) **Create a github repo for the project**
(✔) ~~Connect the repo to a react host service~~
~~( ) *Configure the host and repo to allow ADS*~~
( ) Create a readme file
(✔) **Create the database**
(✔) *(Decide on the data structure)*
(✔) *(Understand use cases)*
(✔) *(Add one quiz to the database)*
(✔) *(Write documentation for the datastructure)*
(✔) **Create the API**
(✔) *(Create the Lambda Functionality)*
(✔) **Connect the database to an API Gateway**
(✔) *(Create the READ functionality)*
(✔) **Create a basic front end**
(✔) *(Install React)*
(✔) *(Install Bootstrap)*
(✔) Connect the API to the front end
( ) Test the API sends data to the database
( ) Test the frontend sends data back to the database
(✔) Test the database sends data to the API
(✔) Test the database sends data to the frontend
(✔) Create the navigation bar functionality
(✔) **Create the display page for the quiz**
( ) **Create the populate quiz functionality**
( ) **Create a display that *does* have correct answers**
(✔) **Create a display that *does not* have the correct answers**
( ) Create a faux log-in page

( ) Create the true log-in functionality
(-) ***(Create user functions and permissions)***
( ) *(Handle these permissions through the react app)*
( ) *(Create a log-out button that works)*
( ) ***(Encrypt the password within the code)***
(✔) **Create a quiz selection page**
(✔) Create the quiz selection functionality
( ) *Create the parameter-based API functionality to return the required data*
( ) Create a display that allows for editing and removing of questions
( ) *(Create the UPDATE functionality in the API)*
( ) Create a display that allows for the creation of new quizzes
( ) *(Create the CREATE and DELETE functionality in the API)*
(✔) Create the readme and install files(?)

"If I had more time" stuff

+ Unit testing - I am to inexperienced to allow this in the initial scope, even as a hopeful objective.
+ Hosted Webapp - Getting this to work as a demonstrable webapp accessible from anywhere would be superfluous to requirements, but a strong demonstration of webapp design that I have learned. Unfortunately sorting dependencies and build requirements by free host is very difficult and time consuming - time better spent delivering functionality.

Blockers Encountered;
React Router and React Bootstrap versions differ from tutorials, left me with irritatingly undocumented errors (router not pointing to the pages, rendering blank pages which is identical to a previous version syntax error - I was chasing the wrong error for a full day). Had to start again and work up manually testing every significant change manually until I found the error was the router, then read documentation to diagnose the basic mistake I had made.

API documentation and official tutorials for AWS are frustratingly basic and often skip entire steps, it was often easier to read old code and backwards engineer it than try to follow documentation. This was made frustratingly difficult with EMIS failing to reinstate my AWS permissions and cutting me off from much of the old code and slowed down progress with connecting the API to the front end code.

CORS permissions cannot granularly be set to wildcards if they are created by an API, you have to create the API Gateway manually to properly configure CORS and point it to the Lambda. This is a marginally less efficient means of invoking the Lambda, but without the wildcard CORS permissions it is impossible to access the database from the front end.

The React code was initially written using outdated class components which placed significant limitations on data handling and manipulation through the app, particularly without access to react hooks so the code needed refactoring into functional components, especially to allow the logic allowing the user to switch seamlessly from the quiz selection to the quiz display pages. This was the first time I have had experience writing react hooks. This also made handling the data from the database problematic, as class components and functional components have different interaction with the DOM and the scope within react code.

The original datastructure had the boolean values that stored the “correctness” of each answer stored in the same array as the answers themselves, alternating between [answer, boolean, answer, boolean]. This caused significant issues sorting the data on the front end and had to be revised.

These blockers were caused, I believe, by a lack of recent experience in working with either any backend technologies, especially AWS, and REACT, coupled with a lack of access to previous code to use for examples. This also played to my advantage when refactoring into react hooks to manipulate the data - avoiding the previous efforts that had universally avoided using react hooks, against best practice because this was what we were taught. However, the impact these blockers and the time taken to move them had resulted in only the most basic vertical slice of production being completed, and much of the end user deliverables not being delivered.

Documentation

Dependencies
React
React-router (latest)
React Bootstrap 5.1.3

Data Structure
Use cases;
Search for a list of all quizzes (Query `PK: “list” SK: “all”`)
Return all data for one quiz (Query `PK: “quiz” SK: begins with: “<quizname>_”`)
Edit data for one quiz
Data Entities + Data Logic
We will need a Partition Key (PK) and a Sort Key (SK) - The Partition Key is mandatory, and effects how the database is searched. To minimize search impact and data usage (and therefor costs) we will duplicate the PK into the Overloaded SK for the Quiz Question Data entity. The Sort Key exists so that we can drill down into the data and create artificial joins - specifically relating multiple entities with a single quiz, and return them in a manageable and predictable way.

List of quizzes;
PK: `“List”`
SK: `“All”`
Item: `“All Quizzes” : [ { “S” : “<quiz name>”, “S” : “<quiz name>”` (etc) `} ]`

This will allow us to search for a list of all quizzes and populate the page listing all available quizzes without querying the entire database on each log-in. It will also make our web app more readable and user friendly, and allow us to filter down to individual quizzes to better serve the user requirements.
This list will only need to be updated when a new quiz is created or deleted.

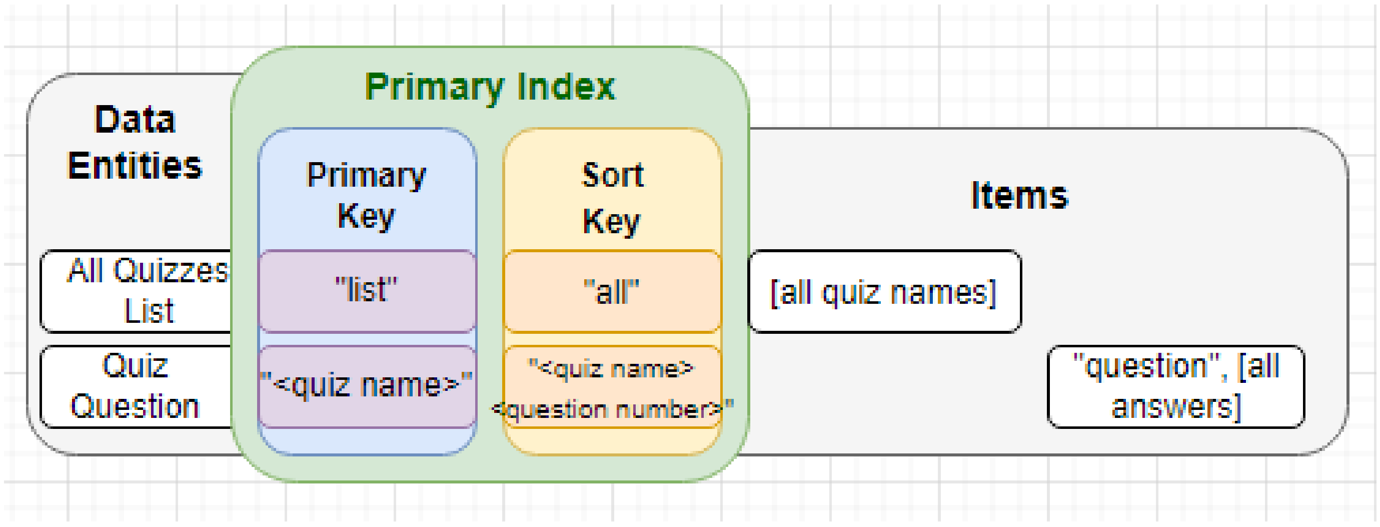
Quiz Question Data;
PK: `<quizname>`
SK: *(not required for query, required for editing a question)*
Item: `“Question” : [  { “S”: “<question>”}, [ { “L” : [ “S” : “<answer 1>”, “BOOL” : <boolean>, “S” : “<answer 2>”, “BOOL” <boolean>}` (etc) `] ]`

* *this was changed to remove the <boolean> from the array the answers are stored in during development, the new structure is;*
`“Question” : [  { “S”: “<question>”}, [ { “L” : [ “S” : “<answer 1>”,`

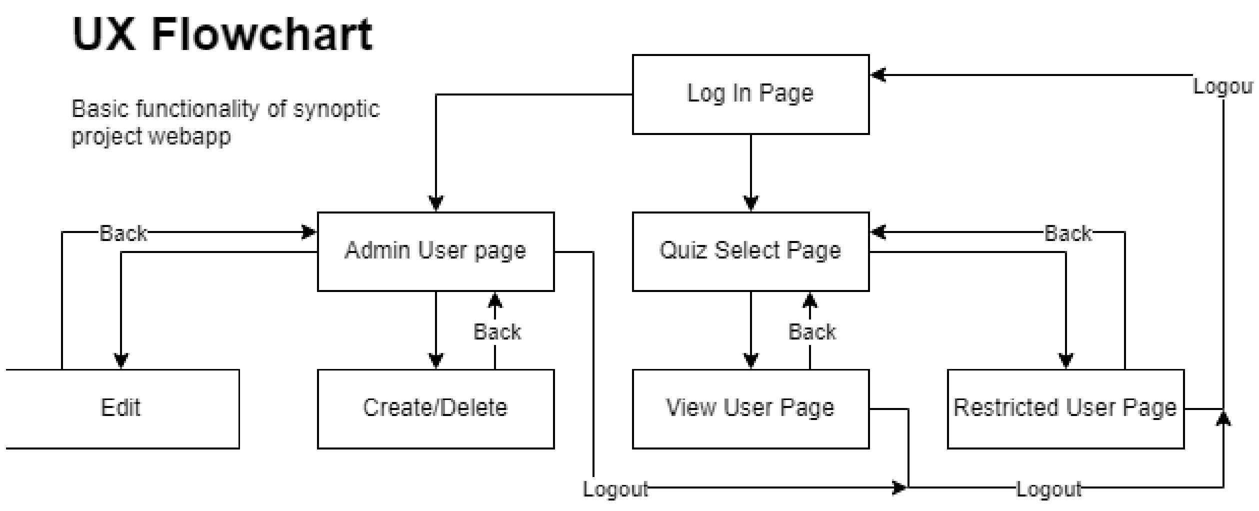
`“S” : “<answer 2>”}` (etc) `[ “BOOL” : <boolean 1>,“BOOL” : <boolean 2>]}]`

The Primary key is simply <quiz name> which allows us to return all the relevant data with a simple PK query - because the PK must be unique when combined with the SK, it will only ever return the relevant items. The Sort Key is overloaded by <quiz name>_<question number> to allow access to editing more easily.
In this data entity the question title is stored as a string value, and an array is returned alongside it containing the possible answers and a boolean value; true is correct and false is wrong. The front end can use this data to populate the quiz page, and we can use the returned data from the list of quizzes to populate our query to return the quiz questions when we render the individual quiz pages.
We do not need to update the question numbers if we do not render them in the front end, they are amended to the PK only to allow us to store multiple entities in our overloaded indexes. If the questions need to be listed by number then that number should be generated in the front end, not using data within the data entities.

This gives us the following data structure that serves all of our use cases;

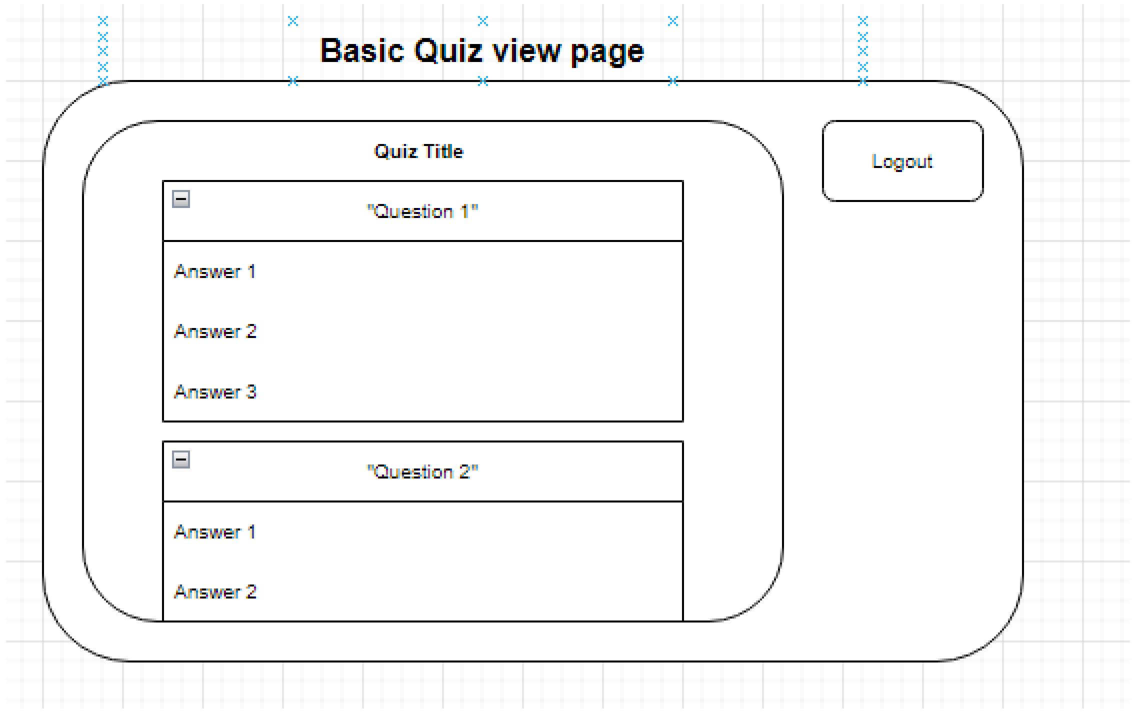


UI and UX Design



This UX is designed to allow the user to flow through the application easily and intuitively, when combined with the GUI below it should allow a user unfamiliar with the webapp to rapidly and easily find whatever function of the webapp they require easily, able to drill down to a particular quiz for ease of use and readability - displaying *every* quiz would not be helpful for the end user.

GUI Design



The GUI should be intuitive and simple following a very standard design, this can be achieved by using react.bootstrap to deliver familiar functionality while minimizing the code writing requirements during development. This will also increase usability and familiar placements. Any variation to layout requirements from different pages (such as edit/delete) will be added to this basic layout.

Permission levels;
Permissions required to service

Edit Level;
View Answers
Create Quiz
Edit Quiz
Delete Quiz

Restricted Level;
View Answers

View Level;
View Questions (default behavior)