Euan Mendoza Year 12 Major Project Logbook

# 15/02/2019

Tested different web frameworks for the potential use

# 22/02/2019

Created a material ui react project and testing different configurations

# 01/03/2019

Created a rust rocket rest api for potential use in project

# 08/03/2019

Created a go project and moved repository to it, discovered a web technology called graphql which removes a redux state management dependency but is not available in rust, so moved to a go web server

# 25/03/2019

Moved repositories again to conform to the go development standards of using go modules, keeping the same packages

# 01/04/2019

Started developing web server. Setup Git Submodules, created branch feature/setup\_webserver in repo. Created a static webserver route for deploying the react website with Go.

# 05/04/2019

Rewrote algorithms and logic within requirements report and planning to better suit the project and improve simplicity. Planned crud features for database managed through an admin panel similar to the Django framework.

# 09/04/2019

Started building the actual graphql api using go-graphql. Some issues I have found is that go-graphql is not documented well, while other graphql libraries such as graphql-java and python graphene are documented well, made me reconsider using go. The other issue is code examples all have one to three queries and responses, my plan has a lot more than three graphql queries. Three quries in a graphql application is small enough to be built in one file, but my project is a lot larger, and personally I prefer to have a lot of code split up into multiple files than a large file with lots of functions. I considered moving to java to make the project object oriented so I could easily organise the project, but java involves a lot of setup with the build systems and dependency management. I also considered switching to a different go graphql implementation that had better support for separating graphql responders into separate files. The solution I ended up using was to create a file called models where I declare the ‘models’ for how the data is requested and stored based off the Django framework. It contains all the structs also known as objects or non primitive data types depending on which language your using, which are used in the application to serialize and deserialize into json which can be used by the client. The structs are than serialised in the same file into a graphql object, which can be used for commands like query users id which returns a list of all users id’s. The graphql object is than mapped to a graphql query request within the schema file, this file handles create, read, update, and delete requests. If a read request is called, a callback function is called which exists within separate files that end in handler, these functions return the requests from the graphql query. This way there will still be a large amount of functions per file, but it is neater than any of the examples provided.

# 10/04/2019

In order to validate the json web tokens for authorization, I needed to pass the json web token in the http request header, the only problem is that I was using a pre written http request and response handler. I decided that because go has inbuilt support for a lot of the server features, and the original intention of the go graphql library is that you roll your own http handler, I decided on creating a http handler function. This function takes the json web token and passes it into the context of the graphql request.

# 11/04/2019

I refactored the code and added comments and extra information

# 12/04/2019

I created the functions and modules that are used to store the constants, for example the secret key used for tagging the json web tokens. The functions are stored locally in a json file and parsed using the native libraries included with golang.

# 14/04/2019

Go lang http handler did not work, so I decided to copy with full credit to the original http handler, but add a line where I pass the variable within the context.

# 15/04/2019

I started working on the frontend, using react router and react to create prototypes of the login screen and the home screen using material ui.

# 16/04/2019

Another set of refactoring because I realised that some of the functions of the server side, namely the resolver to get a login token could be shrunk. I had a set of three functions to handle each user type, but it was three copies of the same function, and in an effort to make the application DRY, I created one function that handles the signing in of all users, instead to login in different user types the application now has an enum where you can select user types with a select within the frontend html select statement. This gets passed as an argument to the resolver function which uses a case statement to login a user of all different types. The problem is when the form submits, it passes all the variables through the html submit functionality. So I started working on react state management as opposed to the traditional html submit to handle the data.

# 17/04/2019

Ditched the idea of using a switch to handle the user selection for logging in because I realised it wasn’t the best way of doing things. Instead I use the url, and have an argument usertype which passes what type of login the user wants. The argument is saved as a state.

# 19/04/2019

Finished the form completely for login, with routing and handling graphql errors. The only problem is the application is returning a overflow error and I cant find the source of the bug. I see to many state calls from the login form, but that could either be the router, the login form or the Apollo library complaining. Will try to fix it but in the meantime improving the backend with more refactoring of functions.

# 24/04/2019

Finished all the authentication and authorization on the backend, and I have started work on fixing the bug. I could not find what was causing the bug, and I decided rather than trying to fix the bug, I would rewrite the entire frontend, using typescript for type checking and making it purely functional. That includes avoiding using loops and instead using iterator methods and callback functions, using constants for everything. I use react hooks for state management so I can have functional forms instead of complicated classes with a single method, and have removed saving the usertype as a state because the usertype never changes on a particular url. On the server side, I have done the complete opposite and have refactored the code to better suit an object oriented paradigm.

# 25/04/2019

I refactored the frontend and improved my typescript, before I used to pass the props as a type of any, but this made the entire type checking redundant. So I learn’t how to create interfaces which are used to add type checking to certain functions, I have switched from Apollo to Apollo-hooks because it brings down the amount of code I need to write and simplifies a lot of things.

# 26/04/2019

Today was testing the frontend and backend together and completely finishing authentication and authorization.

# 27/04/2019

Today was fixing bugs in the backend and developing the actual core functionality. I completely removed the env.json idea and now I hard code the variables into a go file, I added a commented line in the .gitignore file that ignores a go file that has these hard coded variables. I included this in the build so that the project could compile online. All other variables apart from the secret key will be stored in the database. I added more object oriented style programming to the backend through the use of getter methods of the environment variables and also separated the database from the schema through a private pointer and a getter method to the database connection. A lot of the code for the secret key was removed, now it is auto generated everytime the application restarts and is stored in a temporary variable. This means that every time the server restarts, everybodies tokens will be invaled and they will need to sign in again. Through this there is less code and the application is more secure, if I have time I might even make the tokens secret key regenerate each week increasing the security even more.

# 30/04/2019

Remade all userType variables lower case.

# 02/05/2019

Started working on the timetable system, created a model for the school year type, including terms, public holidays, events and periods of a day.

# 04/05/2019

Created a copy of the graphql school year types as an input type in order to make a create user timetable.

# 08/05/2019

Renamed all usertype to user role, merged the admin, teacher, and student table into one user table. Refactored, removed all case and switch statements, mapped the role enum to strings instead of integers and worked on the resolver function for the timetable system. Replaced all userType variables with role, but for some reason the variables are not updating on the production build.

# 09/05/2019

Deleted all cache, restarted computer, and rebuilt the project.

# 10/05/2019

Project works, started working on the resolver function for setting up the school year config. Rewrote many sql tables to utilise the secondary key functionality. The year config has a list of terms, public holidays and events which are used to structure the table dynamically. Rather than a static table, this makes it so that less components need to be re written in the case of a special event.

# 11/05/2019

Finished resolver function but on testing it does not work. The problem seems to be that a list of items is not a supported input type.

# 12/05/2019

Scrapping year config function, the year configuration will have to be inputted by an admin directly into the database. Maybe I might have a create term, or create public holiday input that adds to the current year configuration. Started working on a CRUD user management system. Finished the creation, update and read functions because it was simple, leaving the delete for the next day.