Tarek Chorfi, Project report - DIT127

Approaching this course I knew that I wanted to gain more academic and technical experience in writing react web applications. And I also knew that I had some practical knowledge of the frontend development process, which is why I wanted to do something bolder than if I was completely new to web development. I haven't read the book from cover to cover but I have used some chapters to sharpen my existing knowledge. I was semi new to TypeScript and learned quite a few things from the book which I've used in the project.

The project that I decided on was an E-commerce platform for a Sneaker brand with and CMS (Content Management System) that is intended for the administrators of the company to be able to put items on sale or add new items etc. The E-commerce platform is a web application that let's non users browse the selection but not make purchases. It allow people to register as users and then add products to cart and proceed to purchasing them.

I used ChatGPT to generate possible brand names then started developing the backend server.

This was the first time using express for me and was a good learning experience, I've had previous JavaScript experience which helped me a lot.

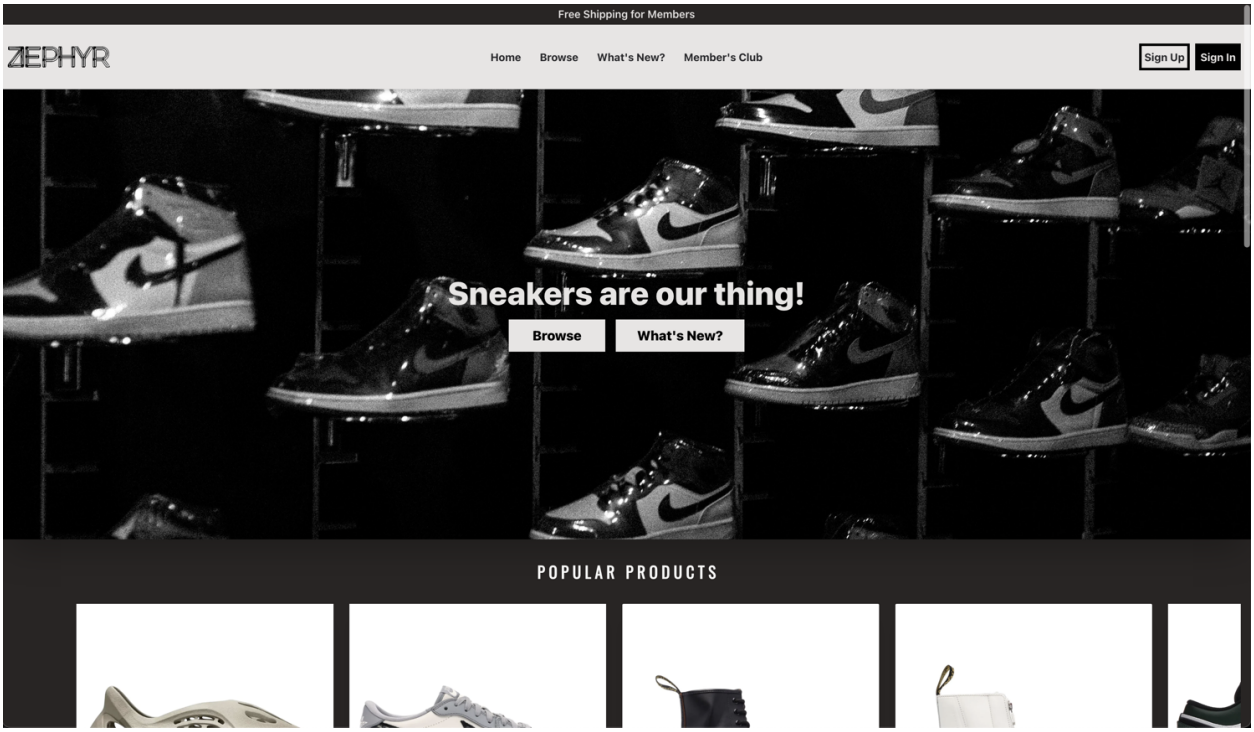
Then I moved on to developing the frontend. I'm familiar with react UI development so this was not a new experience which is why I felt comfortable in taking on the project without joining a group.

The application is divided into two types of models, a product model and a user model. The usermodel is in turn divided into Users (customers) and Admins (employees).

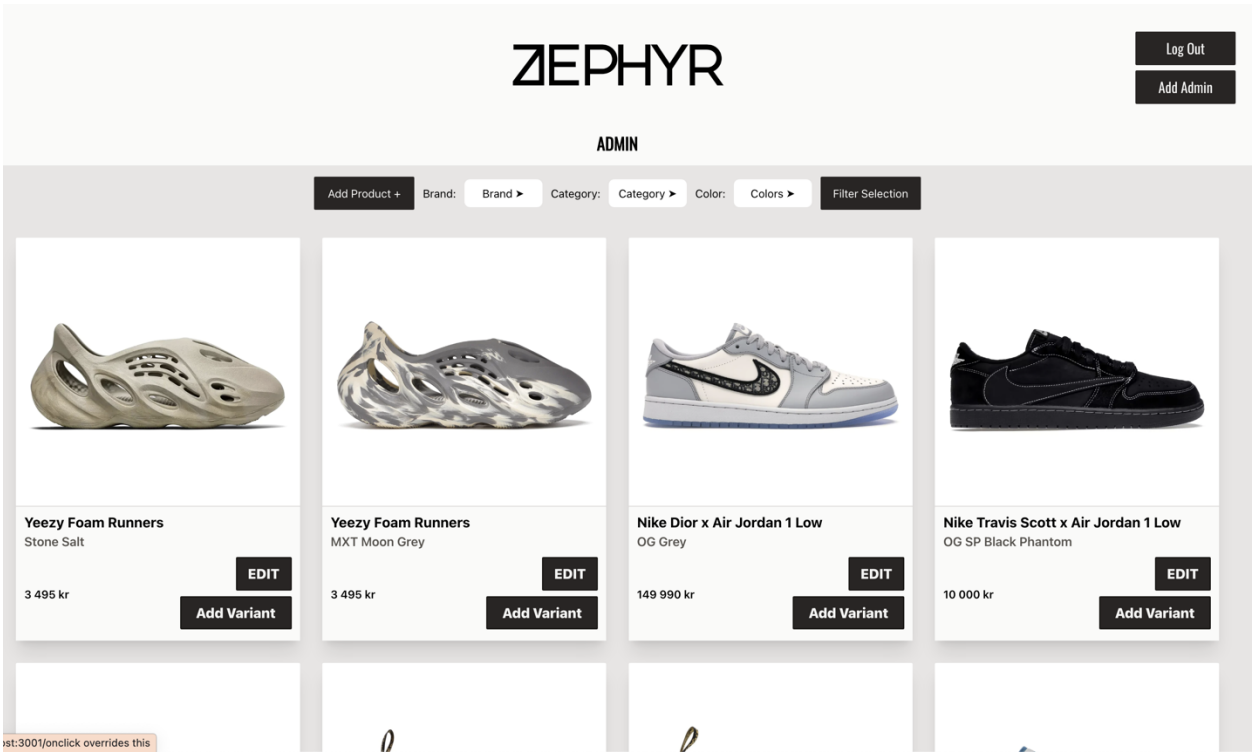
Users guide

```
1. Clone the repo
--Feel free to ignore lab1 since that is not connected to the web app.
--Lab2 is the backend of the webapp
--Lab3 is the frontend of the web app
---Lab3 is divided into two clients (also two directories),
    lab3/website for the ecommerce website and oen
    lab3/manager for the admin cms intended for employees.
2. Run npm install in the three directories (lab2, lab3/website and lab3/manager).
3. Run npm run dev in lab2 to start the server
4. Run npm run start in any of the two clients in
    lab3 depending on which one you'd like to start
```

Main page of ecommerce site



Main Page of manager - After login



Technology used

The frontend is build with React using TypeScript. This was great since I have previous experience with the library and it's a style of code I enjoy writing. Below this title I've listed additional technology used for the frontend.

The backend is build using Node and Express. To start off the storage used was in memory store which means that the data saved on the backend was reset everytime the server restarted. Towards the end of the project that was migrated to a cloud database based solution using MongoDB atlas free tier.

Notable additional technology used.

- TailwindCSS
    - This is a css library that I have previous experience with and I really enjoy the workflow that it provides for, very fast development using an almost inline styling

technique to apply css to the html/jsx/tsx. This is a love it or hate it kind of library, this is mostly due to the tight coupling of css and html, that’s the primary argument against tailwind.
Here is a good blog post from the creator that explains the case for tailwind better than I could.

- js-cookie for cookie handling - Link to NPM page
  - After a little googling I found a good package for easy handling of cookies with approximately 6.3 million downloads per week at the time of writing this.
- Jotai
  - Jotai is a lightweight global state management library that uses native React hook syntax to achieve seamless integration with little configuration needed.
- Framer Motion
  - Framer motion is a great animation library. The library has not been used very much but was downloaded and integrated for future extensions that will not be

The use cases and corresponding api specification for products

```
-Product
-- A product should be requested (along with all it's colors).
-- A product's specific color should be requestable
-- A product should be able to be added.
-- A Product should be able to be modified.
-- A product should be able to be deleted
-- A product's specific color should be able to be deleted.

-Products
-- A list of all products should be able to be requested
-- A Filtered List of products should be able to be requested
-- A list of multiProducts should be able to be sent and updated.
-- A list of all brands should be able to be requested
```

```
//URL = http(s)://api_endpoint.domain

//Get request for all products to  URL/product
//Responses
//200 - Ok - Returns a object of nested product objects
response body: {
```

```
                id1:{
                    color1:product,
                    color2:product
                    },
                id2:{
                    color1:product,
                    color2:product
                    }
                }
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//404 - Not Found - There are no products added yet.



//Get request for a filtered set of all products to  URL/product
//Responses
//200 - Ok - Returns a object of nested product objects
response body: {
                id1:{
                    color1:product,
                    color2:product
                    },
                id2:{
                    color1:product,
                    color2:product
                    }
                }
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//404 - Not Found - There are no products added yet.




//Get request for product and its variants to URL/product/id
//Responses
//200 - ok - Returns the specified product object
response body: {
                color1:product,
                color2:product
                }
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//404 - not found (Request follows specifion but no id found)
response body: {}



//Get request for product's specific color to URL/product/color
//Responses
//200 - ok - Returns the specified product object
response body: product
```

```
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//404 - not found (Request follows specifion but no id or color found)
response body: {}



//Get request for brands stored on the server to URL/product/brands
//Responses
//200 - ok - Returns a list of all recorded brands.
response body: string[]

//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//404 - not found (Request follows specifion but no id or color found)
response body: {}



//Post request for adding a product to URL/product
//Request
body: {
    id:string;
    name:string;
    brand: string;
    description:string;
    color:string;
    generalColor:GENERALCOLOR; //enum
    price:number;
    category:CATEGORY; //enum
    stock:stockedSize[]; //{size:number,amount:number}
    price_factor:number;
    images: string[];
}
//Responses
//200 - ok - Returns the newly added product object
response body: product
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//409 - "Product already exists, did you mean to restock?"




//Post request for updating a list of multiproducts to URL/product/updatecart
//Request
body: {
    clientlist:multiProduct[];
}
//Responses
//200 - ok - Returns an updated multiProduct[]
response body: multiProduct[]
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
```

```
//Put request for updating/editing a product to URL/product
//Request
body: {
    id:string;
    name:string;
    brand: string;
    description:string;
    color:string;
    generalColor:GENERALCOLOR; //enum
    price:number;
    category:CATEGORY; //enum
    stock:stockedSize[]; //{size:number,amount:number}
    price_factor:number;
    images: string[];
}
//Responses
//200 - ok - Returns the newly added product object
response body: product
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//404 - not found (Request follows specifion but no id found)
response body: {}
```

```
//Delete request for removing a product to URL/product
//Request
body: {
    id:string;
}
//Responses
//200 - ok - Returns the newly deleted product object
response body: product
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}
//404 - not found (Request follows specifion but no id found)
response body: {}

//Delete request for removing a product to URL/product
//Request
body: {
    id:string;
    color:string;
}
//Responses
//200 - ok - Returns the newly deleted product object
response body: product
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
```

```
response body: {}
//404 - not found (Request follows specifion but no id or color found)
response body: {}
```

The use cases and corresponding api for Users

```
-Non User -Should be able to add email to newsletter

-User
--A user should be able to be registered
--A user should be able to log in
--A user should be able to create an order
--A user should be able to retrieve its own information
--A user should be able to be deleted
```

```
//URL = http(s)://api_endpoint.domain

-NON USER
//Post request to URL/user/newsletter
//Responses
//200 - Ok - "Sucessfully added mail"



-USER
//Get request for a User to  URL/user/:id
//Responses
//200 - Ok - Returns a user object of type Document<User>
response body:User

//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}

//404 - Not Found - No user found.



//Post request for user login to URL/user/login
//Request
body: {
  email:string;
  password:string;
}
//Responses
//200 - ok - Returns the user object
response body: User
```

```
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}

//404 - "Email or password was not found"



//Post request for registering a user to URL/user/register
//Request
body: {
  user:{
    name: string;
    email: string;
    password: string;
    phonenumber: string;
    birthdate: Date;
    orders: PastOrder[]
    adresses: {
        id: number;
        type: addressType;
        street: string;
        city: string;
        country: string;
        zip: string;
    }[];
}
}
//Responses
//200 - ok - Returns the newly created user object
response body: User

//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}

//400 - "Email already exists"



//Post request for user order to be processed to URL/user/order
//Request
body: {
  id:string;
  items:multiProduct[];
}
//Responses
//200 - ok - Returns the same multiProduct array wrapped in a PastOrder object
response body: {id:number, items:multiProduct[] }

//200 - ok - Returns the updated multiProduct array wrapped in a Error object
response body: {error:true, items:multiProduct[] }
```

```
//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}

//404 - "User doesn't exist"



//Delete request for removing a User to URL/user/:id
//Request
body: {
    user:{
      name: string;
      email: string;
      password: string;
      phonenumber: string;
      birthdate: Date;
      orders: PastOrder[]
      adresses: {
        id: number;
        type: addressType;
        street: string;
        city: string;
        country: string;
        zip: string;
    }[];
  }
}
//Responses
//200 - ok - Returns the newly deleted user object
response body: User

//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}

//404 - not found (Request follows specifion but no id found)
response body: {}
```

The use cases and corresponding api for Admins

```
-Admin
--An admin should be able to be registered
--An admin should be able to log in
--An admin should be able to be deleted
--An admin should be able to retrieve all users for admin purposes
  (not implemented client side)
```

```
//URL = http(s)://api_endpoint.domain
//admin = 7b2e9f54cdff413fcde01f330af6896c3cd7e6cd



-USER
//Get request for all users to  URL/admin/:id
//Responses
//200 - Ok - Returns a list of users
response body: User[]
//500 - internal server error if fetch fails.




//Post request for user login to URL/admin/login
//Request
body: {
  email:string;
  password:string;
}
//Responses
//200 - ok - Returns the admin object and sets 'user' cookie as admin object
response body: Admin

//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}

//404 - "Email or password was not found"




//Post request for registering a user to URL/admin/register
//Request
body: {
  user:{
    name: string;
    email: string;
    password: string;
  }
}
//Responses
//200 - ok - Returns the newly created Admin object
response body: Admin

//400 - bad request if no user (admin) cookie is set.

//400 - bad request (This is triggered if the request parameters fail to meet specifications)

//400 - "Admin already exists"
```

```
//Delete request for removing an Admin to URL/user/:id
//Responses
//200 - ok - Returns the newly deleted Admin object
response body: Admin

//400 - bad request (This is triggered if the request parameters fail to meet specifications)
response body: {}

//404 - not found (Request follows specifion but no id found)
response body: {}
```

Architecture

The project followed a server side Model-Controller and client side View architecture as instructed by the professor. Pros with this model is having less processing load on the client and handing that over to the server. This may be redundant in applications that aren’t heavy on computing resources.

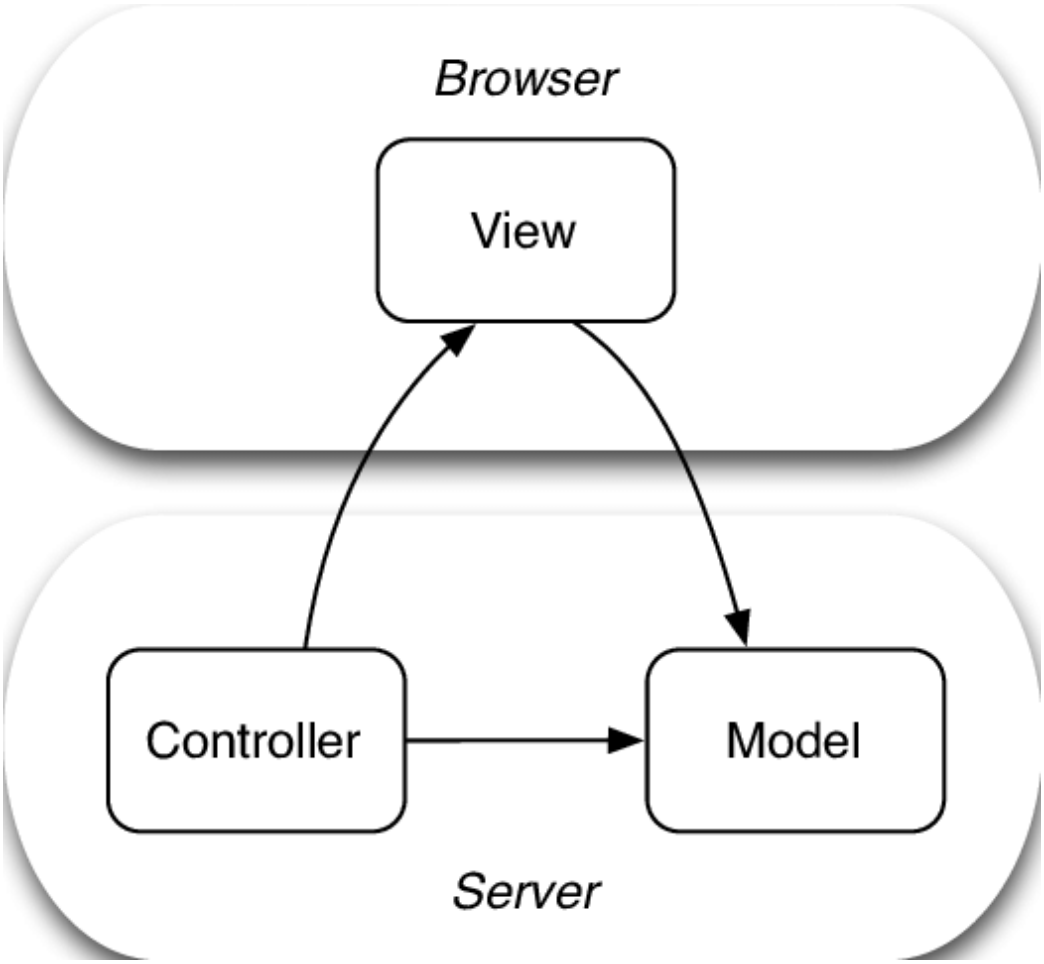


image source

Server Architecture

The server followed a Model-Service-Router architecture as instructed by the course professor.

The case for the Open Closed Principle is that the router should handle routing while treating the modification of the model as an abstraction and passing that responsibility on to the service. Thus Here we are also presenting an argument for the Single Responsibility Principle. The router handles routing and the service handles the model modification.

A case against OCP in my service-router code is that I use a class called ProjectError which returns a message and an HTTP code, I would argue that this is not a violation since the router will always handle HTTP requests.

The model was first an in-store class based solution which was then refactored into a cloud database based persistence solution.

Testing

Testing was done using supertest and jest. I only managed to come around to doing unit and integration tests on the server due to the problems faced when trying to test the frontend and did not have time to test the frontend. The tests can be found in lab2/src/tests which has two directories for router tests and service tests. I also didn't test the admin service since it's almost identical to the user and most of the admin service methods are tested in the integration.

Extension

Me being newer to backend development I have learned a lot of things, some things were met early while others not so much. Very close to the deadline I discovered that cookies are limited to 4KB, this threw a monkey wrench in my project, both the user and cart cookies had potential to exceed far beyond 4KB, which is why I refactored the user session cookie and cart cookie into using a browsers localstorage instead, now I'm faced with a dilemma, how do I send this "cookie"? Is it in the body of my requests or do I create a form of adhoc cookie that stores the minimum amount of needed information to validate a user, I do not currently know the answer to this question and I will not be able to implement it in time, so for now my application does not bode well in the security department. In the case of the admin side I suppose I could use a limiting CORS solution, by allowing a certain IP to access the admin router I suppose I could protect the application a little bit in that sense. Then I'd still need to validate queries to some of the more fragile methods in the ProductService, namely the manipulation of Products, I'm not entirely sure how I would go about it. The only thing I know is that there is a lot more for me to learn specifically about security and best practices in that field.

There are more usecases that I've simply not implemented yet, due to lack of interest or time constraint.

- A user should be able to change their email or password.

- A user should be able to update their address information
- An admin should be able to delete their account.
- An admin should be able to send a put or post request to manipulate existing users.
- An admin should have different roles, maybe separate a super admin from a regular admin and determine which use cases the different roles should be able to use.

Testing should be able to be done on the front end. I was met with problems during my initial frontend testing phase which I eventually managed to debug, although I couldn't focus my efforts there since there was still much to do. The manager part of the client needs commenting. I've cleaned out and commented the website since that was the main focus of the project. The manager mostly to manipulate what is shown on the website which of course is also a large part and I'd love to have had more time to comment that part, I did clean it a bit and reduce code clutter and increase code reuse.

There are some commented parts related to performance enhancements in the service. Off the top of my head I know that when I process the order I call the getProductColor method every loop iteration which is really unnecessary, I could simply call the getProducts to get all the products and store it outside the loop to reference it every iteration, searching through a Map is very fast and would increase the performance of processing orders.

Security Concerns

I met an issue related to cookies, their limit being 4kb and my user object are over 3.5kb and my cart cookie is the real problem, a user shouldn't be limited to **x** cart items. Thus I switched the website cart and session storage into the localStorage object.

The server doesn't check that a user cookie is set, since I met the problem with cookies being limited to 4kb I've been unsure on what to do. Should the client send the user as payload in the body instead, to then be parsed and checked by the server.

The admin router doesn't check cookies yet, but this is something that will be fixed if I get around to it. There is no issue in storing admins as cookies since they will most likely not

push 4kb due to the static nature of their storage (no collections of previous orders or addresses like the user object).

Conclusion

Overall I’m content with the job that I’ve done, I definitely see the shortcomings of the projects and some of the interesting solutions to the problems I’ve faced. The biggest issue for me was to come up with the ideas of how an ecommerce store should be modeled and I’ve certainly learned a fair bit.

Some notes reflecting on the process of the project.

Practical lessons learnt:

- Ponder where to handle errors and user events. Handle them as early as possible.
    - Example if a shoe size isn’t in stock. Make the text gray and unclickable. At first I had it clickable and then handled the event inside the addToCart function, later I moved this into the onClick on the add to cart button {size.size != 0 && addtocart()} but then I realized that it’s better to disable any triggering of a mouse click in the sizelist component. Simply don’t give an onclick to the list item if the stock is 0.
    - This seems like an obvious thing when you realize it, but I can imagine that in the heat of the moment one doesn’t necessarily think about it. In my case I created the sizelist component 3 days before encountering this issue. Thus it took me working wiht the problem for a little while before I realized that I should move the handling of this event (user choosing a size that isn’t in stock) as close to the initial event (list item onclick) as possible.

Reflection:

- Utilize the power of code reusability, in the component sense. I did this in many places but as I was working alone my workflow was more “code fast, clean up later” since I’d rather have a working project that is slightly messier, which I can just clean up than not having a fully functioning project.
- Plan more with a pen and paper when presented with a project with a deadline. I feel that I’ve done fairly alright considering that I’m alone in my group. I had the project in my head for the whole duration. Getting inspired on walks and creating a class or a component for the idea I’ve had but the ideas just kept coming. I realized more and more that I had a lot of functionality to implement, the closer I got the larger the project got until I just had loose ends to tie up. I think that the reason for this is mostly is due to me not having anyone to bounce ideas off off, I’ve been the sole thinker on the project which is bound to lead to some pieces missing.

  If I would’ve planned the project a little more I would have a concrete roadmap to follow, but I’m still satisfied with my workflow, I prefer this workflow when doing hobby projects.
- Problems arising are truly just an emergent phenomenon of many things and life and software development is certainly not exempt from this phenomenon, I would say, welcome these problems and handle them with grace. Sometimes for me it’s easy to get caught up in the anxiety of fires turning up all around me, and many times I accept the fact and keep working in a calm manner, not to say that I neglect the fires if I feel more anxious, it’s just that the process is executed with more irritation. Welcome these problems and accept them as a part of the development journey.