KMITL

Developer Manual

Bangkok, Thailand

Jarne Dirken
Kobe Vandendijck
Sohaib Ibenhajene

2023 - 2024

Introduction

Welcome to the Developer's Guide for the E-borrowing Application. This manual is designed to provide a comprehensive overview of the technical structure and coding practices employed in the development of KMITL’s e-borrowing platform. Created as part of our internship project, this guide delves into the architectural design, folder structure, and challenging aspects of our application.

Throughout this manual, we will systematically explore the source code, explaining the purpose and functionality of each module and component. We aim to provide clarity on the organization of our codebase, the interaction between different parts of the application, and the specific solutions implemented to tackle complex programming challenges.

This document is intended for current and future developers who will maintain, upgrade, or scale the application. It serves as a roadmap to understanding the intricate details of our development process and offers insights into the decisions that shaped the final product.

By the end of this guide, you should have a thorough understanding of the code structure and be well-equipped to effectively work on and enhance the application. Whether you are troubleshooting, adding new features, or simply reviewing the code, this manual will be an invaluable resource in your ongoing engagement with our application.

Table of contents

Responsibilities

Introduction: Sohaib Ibenhajene

Folder structure: Sohaib Ibenhajene

Setup: Kobe Vandendijck

Database: Jarne Dirken

Backend: Jarne Dirken

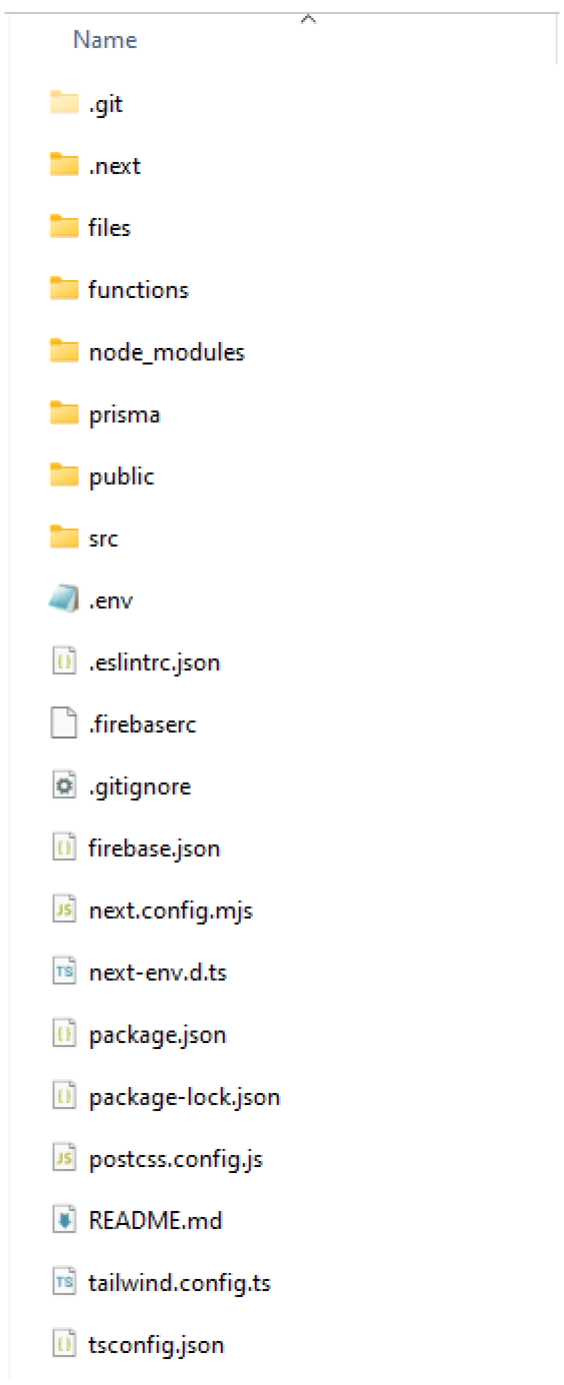
Frontend: Jarne Dirken

Firebase: Jarne Dirken

1. Folder structure

1.1. Global

First, I'm going to explain our folder structure. As you can see in the image next to the text. This is our folder structure. I'll start explaining from top to bottom.



Name

.git
.next
files
functions
node_modules
prisma
public
src
.env
.eslintrc.json
.firebaserc
.gitignore
firebase.json
next.config.mjs
next-env.d.ts
package.json
package-lock.json
postcss.config.js
README.md
tailwind.config.ts
tsconfig.json

The folders .git, .next and nodes_modules can be ignored. .Git is for GitHub or GitLab, .next is an automatically generated directory by next.js and nodes_modules is for dependencies.

The folder "files" is there just for our personal GitHub readme, this one can also be ignored.

The folder "functions" is made with the intention to add time-based notifications to the application with firebase. Because we don't have access to the paid subscription of firebase we couldn't test our functionality out. The purpose of this folder is to check every hour or so if a request has been expired or not. When the request has been expired the user should get a notification. In theory this should work but we don't have the recourses to test it out.

After we have the directory "prisma" this is where we will be making the database. We'll explain this folder further later on in the document.

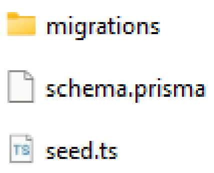
Next, we have the folder "public" this is where we store all the images, assets and svg's of the project.

And finally, we have the folder "src" which stands for source folder, this is where all of our code is based, this also is explained later on.

The files that are important are, ".env" which is for environment variables. "package and package-lock.json" this is for all dependencies and "tailwind.config.ts" which holds our tailwind config with custom variable names in for colors.

1.2. Prisma folder

Going deeper in the prisma folder you will find 3 files. Migrations, schema.prisma and seed.ts.



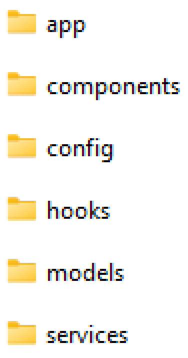
Migrations is for all the database migrations of your application. This acts as a source of truth for the history of your data model. More information can be found on: prisma's website itself.

Schema.prisma is the file that is used for creating all the tables and relations in the database. If you open this file, you will see that prisma is very easy to understand if you have ever worked with databases before if it's not clear you can take a look at their website. We added comments to the code to make your job easier and see what the relationships are.

Lastly, we have the seed.ts file. This file is used to seed the database with dummy data. This is used for testing purposes. NOTE: We have copied the first 30 ich items from the current E-borrowing system so the data should be as accurate as it could be.

1.3. Src folder

The src or source folder is the folder where you will spend almost of your time as a coder, so it is important to understand the structure we went for.



If you are familiar with next.js you will know that the "app" folder is generated by next.js itself and is needed to run your application. If you come from next.js 13 you can see this as the "pages" and "Api" folder but combined into one. We'll dive deeper into the app folder after explaining the rest.

After we have the "components" folder which is where we will store all the components of each part of our application. We have named everything so it's clear to take over. The 3 main roles are defined there, admin, supervisor and user. Then we have some general components used almost everywhere. Another folder for layout which is used to control the layout and lastly states which is where we stored some state components like buttons, loading animation and custom data pickers. This is done because the states of these components change.

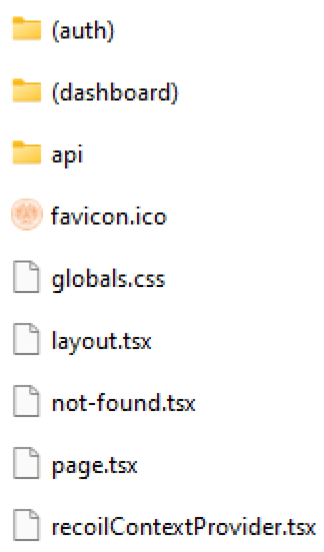
Next, we have the folder "config" this one is not available on GitHub because this folder contains the firebase admin account key. This one needs to be present inside the project but never be available to the public.

After that we have the folder "hooks" which speaks for itself, here we create our custom hooks inside of the application.

After that we have a folder “models” which represents the models inside of our database. Then a good question is, why have another model that does the same as the one in our database? This is because typescript needs to have specific types. This is when you pass data through or read data in, typescript wants to know what that data can be. This is why we have a folder models with all the models in there.

Lastly, we have the folder “services” which contains all the services our application uses. The give a quick rundown of the files in there:

- Db.ts is for the database to make sure there only runs one instance of prisma, for more detailed information check out their documentation.
- Firebase-admin-config.ts and firebase-config.ts are both firebase config files that use the env variables. So we can use the functions of firebase.
- Store.ts is used for recoil.js which is a dependency used to manage state globally in next.js



```
(auth)
(dashboard)
api
favicon.ico
globals.css
layout.tsx
not-found.tsx
page.tsx
recoilContextProvider.tsx
```

Now we’ll explain the folder “app” in some more detail. You need to view the app folder as your route folder. Meaning every folder you make is going to be a path in the browser. In here you see (auth) and (dashboard) we placed brackets around these names so it will not count as a path. The folder api on the other hand will be seen as a valid path so everything inside there you need to access via “domain/api/…”.

For the folders with the brackets we made them just for clarity, next js will not pick them up as valid paths so “domain/(auth)/” is not possible but instead it will look for the next folder inside auth to see if that one is valid. Inside auth we have Everything that has anything to do with the authentication so login folder which leads to “domain/login” and register.

Dashboard is the same there we have put everything for our dashboard in. As you can see, we have grouped everything very needly and clear. The same can be said about our api folder this is where all the api calls will be. Also, this is made into 4 separate files. General, which is globally the same for everyone, admin which are calls only for admin, the same for supervisor and user.

You can also see the favicon here, global.css file which is almost empty we only used it to get a custom scrollbar. This folder as well as (auth) and (dashboard) have a layout.tsx file. This is needed for the layout of the application. This means that the global layout is different than the layout inside dashboard or auth which is what we want.

Every page you make is a folder, but the folder needs a file called “page.tsx” to be recognized as a valid page in next.js. In our app folder we have one page.tsx folder which is the home page. If you remove this file, the home page will not show.

2. Setup

2.1. Setup Ubuntu environment

For hosting this application we use an ubuntu server 22.04. On this server we installed docker and node.

```
sudo apt update
sudo apt upgrade -y
sudo apt install -y nodejs npm
sudo apt install -y apt-transport-https ca-certificates curl software-properties-common
sudo apt install -y docker-ce
sudo systemctl enable docker
```

These are the commands you need to set up all the software you need to start. First you update & upgrade your machine and after you install the software, the last command is to enable docker on startup.

2.2. Setup Docker containers and hosting

First you will need to create a fresh directory with the "mkdir" command. And after you will move into the directory.

```
mkdir webapp
cd webapp
```

Then you will need to copy you application into this directory, with the fresh directory being the root of the project. You can do this in multiple ways: with git clone (from a github project), or just use an application like Visual Studio Code that integrates your VM and host system. This will let you copy the full application into the VM by drag and drop.

After you have done this you will need to create some files for setting up docker.

```
touch .env
touch docker-compose.yml
touch Dockerfile
```

And to help with certain steps in the process we also created 2 bash files. These bash files have a function in the process and will be explained later. And because these are bash files you also need to change the permissions and give execute permissions to users.

touch execute
touch wait-for
sudo chmod u+x execute
sudo chmod u+x wait-for

Now to go over what every file does and contains.
We'll start with the '.env' file, this file contains all the variables and sensitive data we will need in this project.

```
# Environment variables declared in this file are automatically made available
to Prisma.
# See the documentation for more detail: https://pris.ly/d/prisma-
schema#accessing-environment-variables-from-the-schema

# Prisma supports the native connection string format for PostgreSQL, MySQL,
SQLite, SQL Server, MongoDB and CockroachDB.
# See the documentation for all the connection string options:
https://pris.ly/d/connection-strings

POSTGRES_PRISMA_URL="…"

NEXT_PUBLIC_FIREBASE_API_KEY ="…."
NEXT_PUBLIC_FIREBASE_AUTH_DOMAIN="…"
NEXT_PUBLIC_FIREBASE_PROJECT_ID="…"
NEXT_PUBLIC_FIREBASE_STORAGE_BUCKET="…"
NEXT_PUBLIC_FIREBASE_MESSAGING_SENDER_ID="…"
NEXT_PUBLIC_FIREBASE_APP_ID="…"

SERVICE_JSON_FILE='…'
```

Next is the docker-compose.yml file:

```
version: '3.9'

services:
  postgres:
    container_name: db
    image: postgres:latest
    restart: always
    environment:
      - './.env'
    ports:
      - 5432:5432
    volumes:
      - postgresdb_data:/var/lib/postgresql/data
    networks:
      - KMITL_inventory

  inventory_webapp:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "80:80" # Adjust the port mapping if needed
    depends_on:
      - postgres
    environment:
      - './.env'
      - PORT=80
    networks:
      - KMITL_inventory

networks:
  KMITL_inventory:
    driver: bridge

volumes:
  postgresdb_data:
```

This file will spin up both containers we will need for this project. The first is the PostgreSQL container. For this web application we don't need a custom image so the base image from docker will do just fine. We also specify ports through which we will connect to our application container. For this connection there will also need to be a bridge network between containers "KMITL_inventory" this can be created with the "***sudo docker network create -d bridge KMITL_inventory***" command. And the final thing we need for Postgres is a volume that will make our database data persistent stored here "**/var/lib/postgresql/data**".

The application container will need to be custom. And because we create the application image and run our ORM, Prisma, at the same time, the image will need to be created after the Postgres container is already running. We again connect to the bridge network and also specify which ports need to be opened. We also specified the environment file so that the program knows where to look for variables.

Next is the Dockerfile:

```
# Use Ubuntu 22.04 as base image
FROM node:22.1.0
# Install Node.js and npm

RUN apt-get update && apt-get install -y postgresql-client

# Set working directory
WORKDIR /webapp

# Copy package.json and package-lock.json
COPY package*.json ./
```

```
# Install dependencies
RUN npm install

# Copy the built application + scripts + prisma
COPY . .


# Expose ports
EXPOSE 80
# Set environment variable for the port
ENV PORT 80

CMD ["bash", "./wait-for", "postgres:5432", "bash execute"]
```

This file specifies everything we need for the application container, starting with node and a specified version. After we install some extra software and specify a work directory. Next we copy the package.json and package-lock.json files into the work directory. These files contain all the dependencies we need for the application, the following command is to install them all. After we copy the other files into the work directory, expose the port and run some commands.

This is where the bash files come into play. Before we complete the process of building the image, we need to wait and make a connection to the database. This is what the wait-for script is for.

```
#!/bin/sh

# Usage: ./wait-for host:port cmd

set -e

host="postgres"  # Setting the host explicitly to "postgres"
port="5432"      # Setting the port to 5432 as default
cmd="$2"         # Setting cmd to the second argument passed to the script

until pg_isready -h "$host" -p "$port" -q; do
  >&2 echo "PostgreSQL is unavailable - sleeping"
  sleep 1
done

>&2 echo "PostgreSQL is up - executing command"
# Execute the command passed as the third argument
exec $cmd
```

This script waits until a connection is made with the database container through the hostname and port it is supposed to forward to. Before continuing the process.
If the connection is successful the execute bash file will be run.

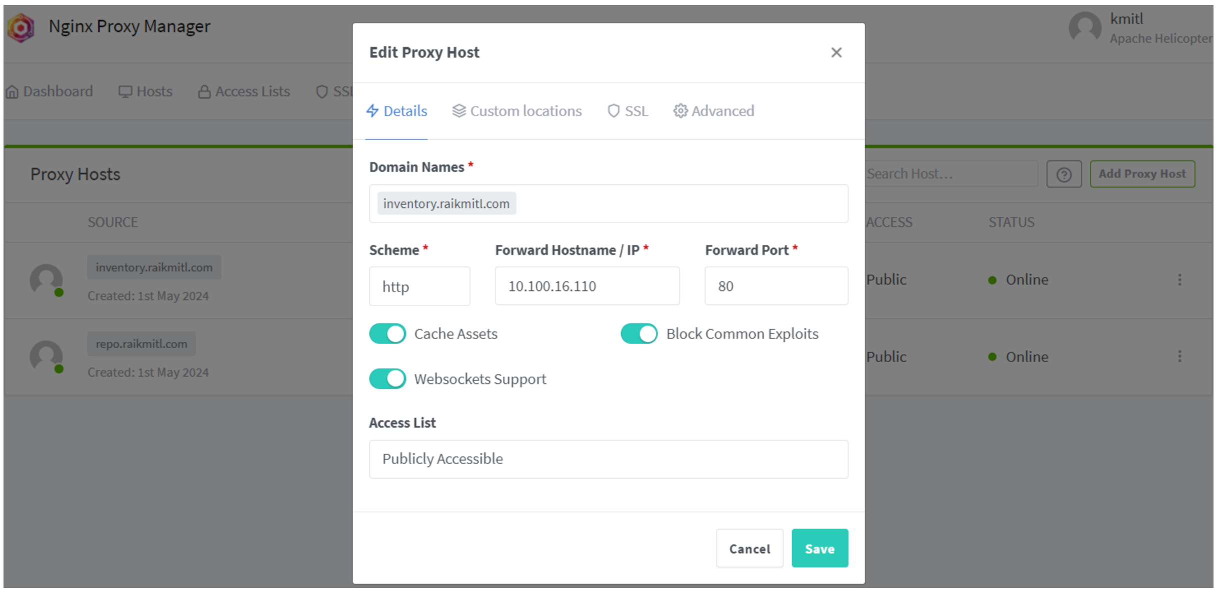
```
#!/bin/sh

npx prisma migrate deploy
npx prisma db seed
npm start -d
```

This script deploys our ORM into the database container, seeds it with some necessary data, and starts the application.

After you have all these files you need to run some commands to start the process.

sudo npm install
sudo npm run build
sudo docker network create -d bridge KMITL_inventory
sudo docker compose up -d

This is the last thing you need to do on the ubuntu server. We worked with Nginx proxy manager for the forwarding to the FQDN.



Edit Proxy Host

Domain Names * inventory.raikmitl.com

Scheme * http

Forward Hostname / IP * 10.100.16.110

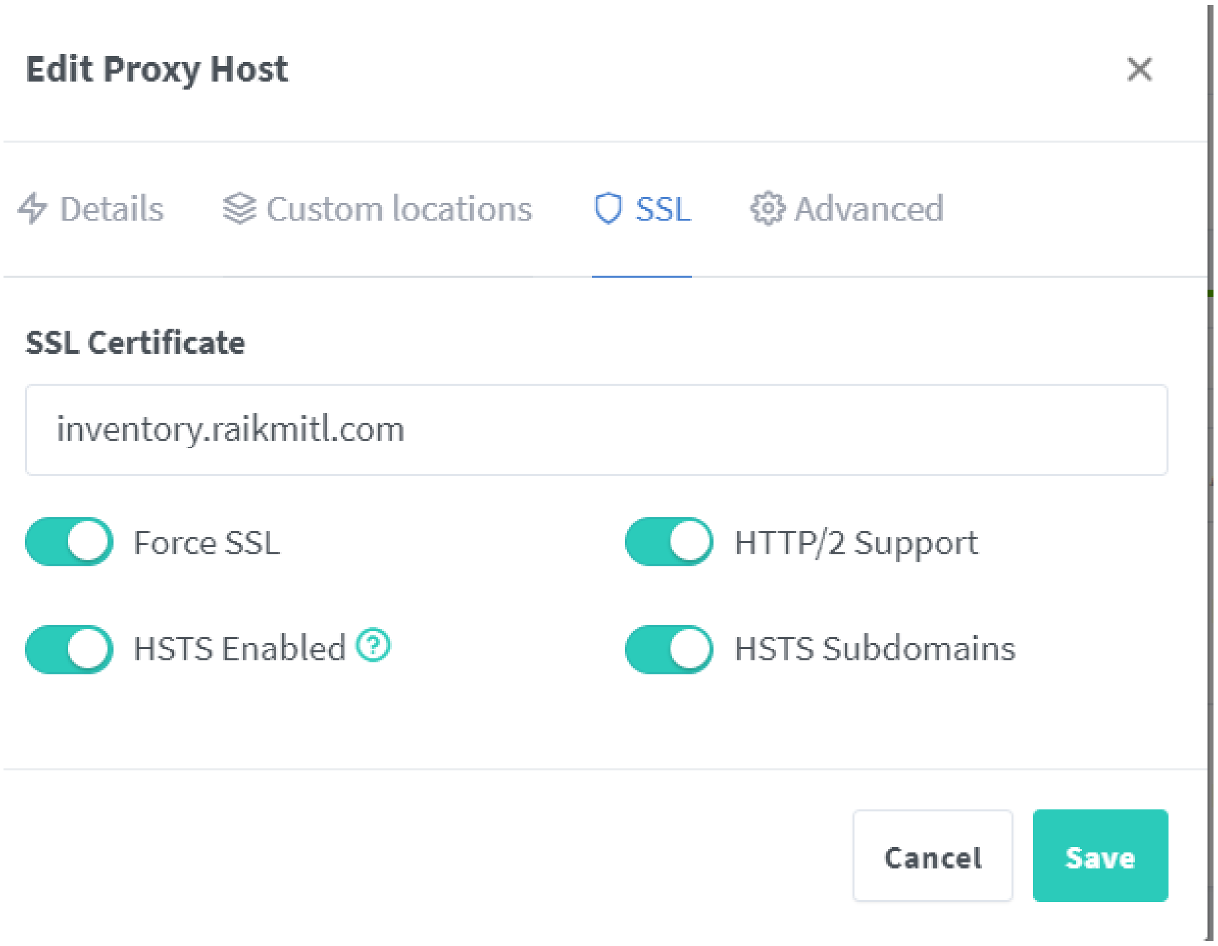
Forward Port * 80

Cache Assets: on

Block Common Exploits: on

Websockets Support: on

Access List: Publicly Accessible

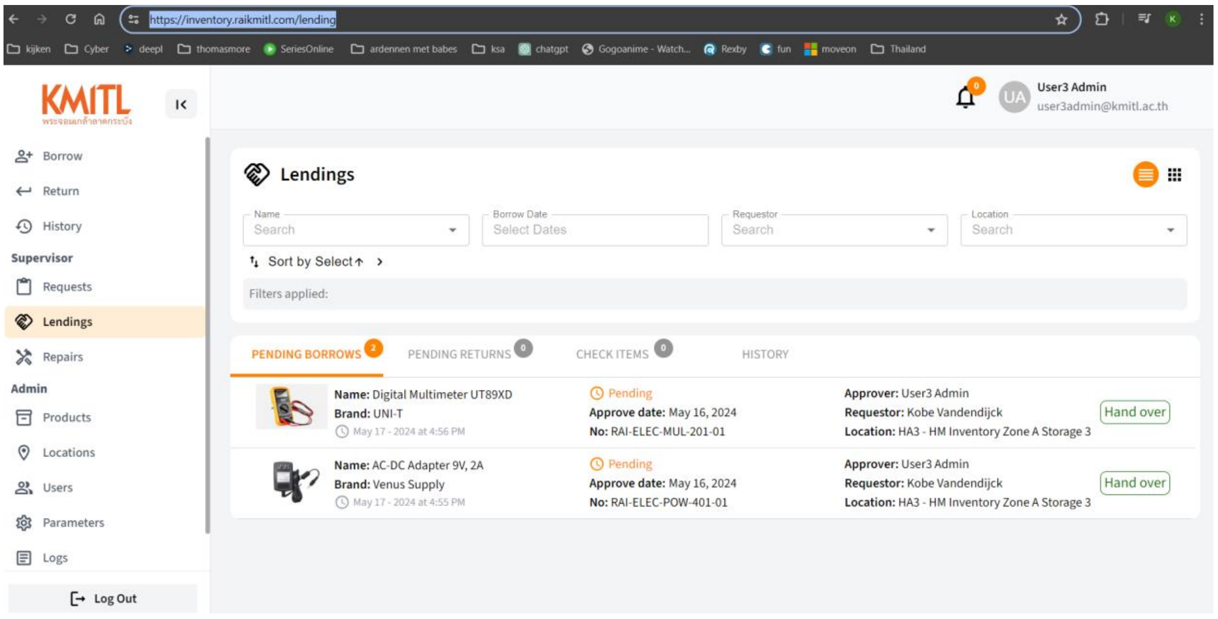


SSL Certificate

inventory.raikmitl.com

Force SSL: on
HTTP/2 Support: on
HSTS Enabled: on
HSTS Subdomains: on

Fill in the domain name and specify where the application is being run from, from which port and force SSL to make it secure.


https://inventory.raikmitl.com/lending

Lendings

Name: Search
Borrow Date: Select Dates
Requestor: Search
Location: Search

Sort by Select

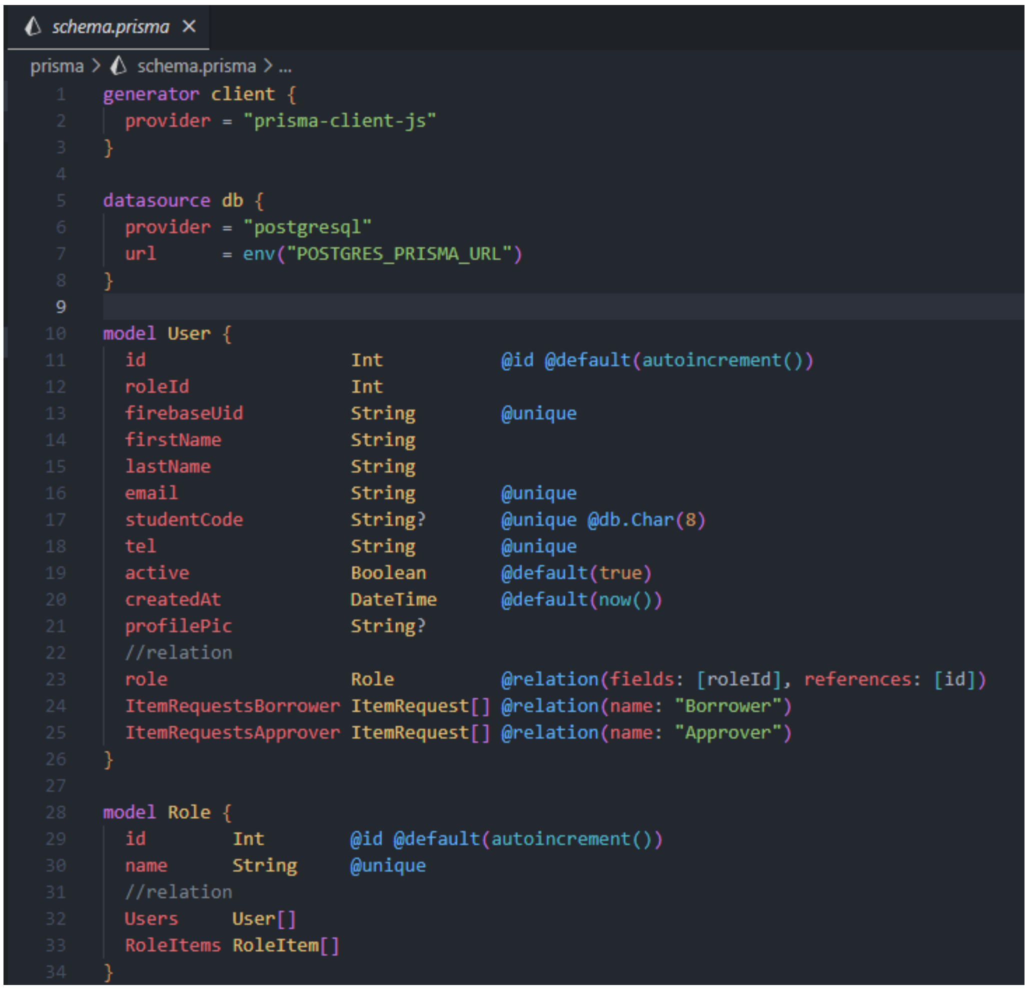
Filters applied:

PENDING BORROWS 2 | PENDING RETURNS 0 | CHECK ITEMS 0 | HISTORY

Name: Digital Multimeter UT89XD **Brand:** UNI-T May 17 - 2024 at 4:56 PM	Pending **Approve date:** May 16, 2024 **No:** RAI-ELEC-MUL-201-01	**Approver:** User3 Admin **Requestor:** Kobe Vandendijck **Location:** HA3 - HM Inventory Zone A Storage 3
Name: AC-DC Adapter 9V, 2A **Brand:** Venus Supply May 17 - 2024 at 4:55 PM	Pending **Approve date:** May 16, 2024 **No:** RAI-ELEC-POW-401-01	**Approver:** User3 Admin **Requestor:** Kobe Vandendijck **Location:** HA3 - HM Inventory Zone A Storage 3

3. Database

As previously explained, we have covered the prisma directory and what it is used for. Here I'll give the code example with some extra explanation on what it is and does.



schema.prisma ×

prisma > schema.prisma > ...

```
1   generator client {
2     provider = "prisma-client-js"
3   }
4
5   datasource db {
6     provider = "postgresql"
7     url      = env("POSTGRES_PRISMA_URL")
8   }
9
10  model User {
11    id                    Int           @id @default(autoincrement())
12    roleId                Int
13    firebaseUid           String        @unique
14    firstName             String
15    lastName              String
16    email                 String        @unique
17    studentCode           String?       @unique @db.Char(8)
18    tel                   String        @unique
19    active                Boolean       @default(true)
20    createdAt             DateTime      @default(now())
21    profilePic            String?
22    //relation
23    role                  Role          @relation(fields: [roleId], references: [id])
24    ItemRequestsBorrower ItemRequest[] @relation(name: "Borrower")
25    ItemRequestsApprover ItemRequest[] @relation(name: "Approver")
26  }
27
28  model Role {
29    id        Int       @id @default(autoincrement())
30    name      String    @unique
31    //relation
32    Users     User[]
33    RoleItems RoleItem[]
34  }
```

As you can see in the image above. This is how the prisma file looks like.

The first 3 lines of code let you use prisma.

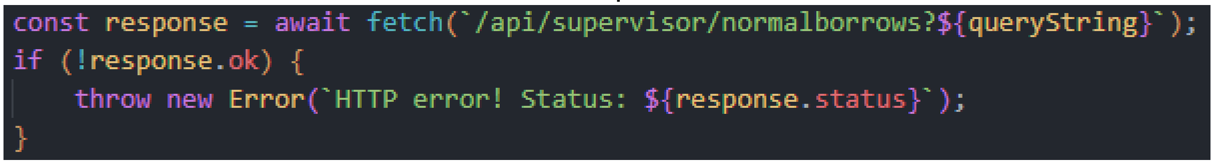
From line 5-8 you declare what database provider you want and what the url to the database is. The url comes from the .env file.

All the rest of the lines you will see that we make our models and connect them to each other using relations. As you can see it is easy to understand.

If we break down line 10-26 quickly everyone will understand. We declare a model that is going to be called "User" we give it an id which is the primary key (@id @default(autoincrement()). Because user and role have a Many to One relationship, we need a roleId which is the Foreign Key. Then we provide all the details of a User, name, email, etc… And lastly, we add the relations. Because we have a many to one with role, we need to add a @relation comment and reference the roleId we made to connect it to the Role Model created at line 28.

4. Backend

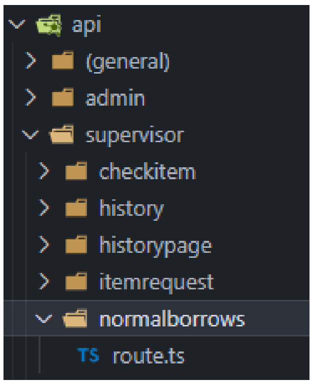
As for our backend, we will cover the api and some api calls made to help you understand. In our frontend we call the api like this:



```
const response = await fetch(`/api/supervisor/normalborrows?${queryString}`);
if (!response.ok) {
    throw new Error(`HTTP error! Status: ${response.status}`);
}
```

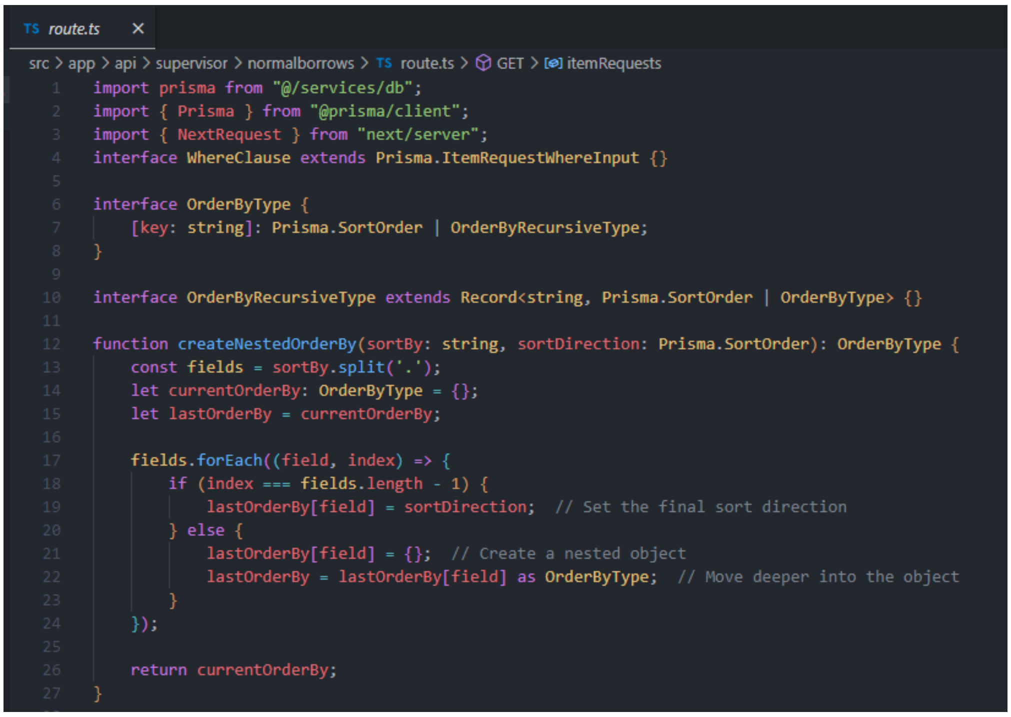
Not every api call has a queryString but this is used for the filter which is used on almost every page of our application so it will be quite normal to see this in our code.

Now when we go to the folder api/supervisor/normalborrows you will see this:



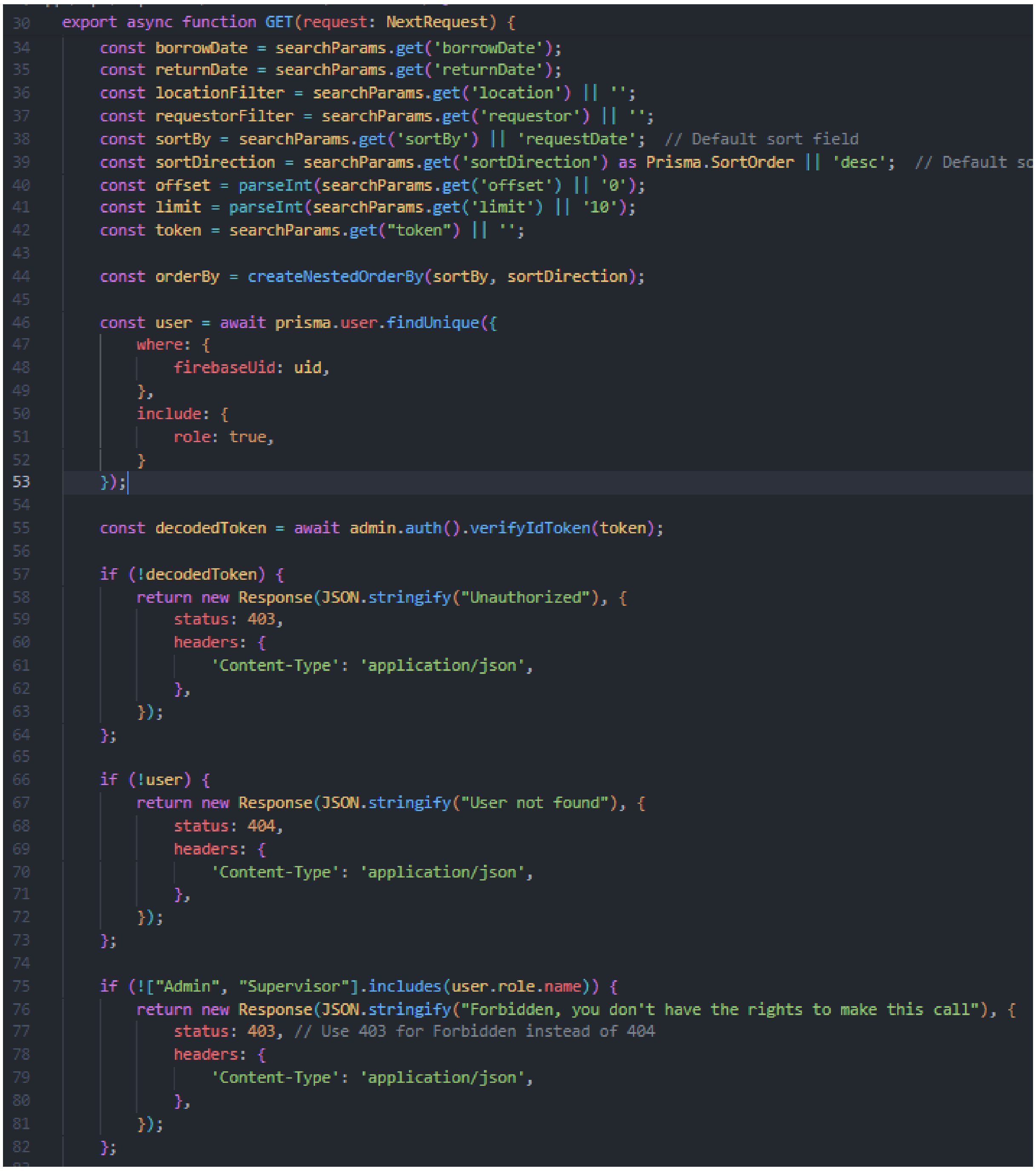
As you can see the route is a folder and inside that folder we have a route.ts. This is how next.js knows it is an api call. Each folder you see is a route when it has a route.ts file inside.

Opening the file, it will be quite overwhelming. The first part is imports. After that we have code that is used to check for the type of sort we want to do. This meaning filter on name asc or desc is all handled here:


route.ts
src > app > api > supervisor > normalborrows > route.ts > GET > itemRequests

```
import prisma from "@/services/db";
import { Prisma } from "@prisma/client";
import { NextRequest } from "next/server";
interface WhereClause extends Prisma.ItemRequestWhereInput {}

interface OrderByType {
    [key: string]: Prisma.SortOrder | OrderByRecursiveType;
}

interface OrderByRecursiveType extends Record<string, Prisma.SortOrder | OrderByType> {}

function createNestedOrderBy(sortBy: string, sortDirection: Prisma.SortOrder): OrderByType {
    const fields = sortBy.split('.');
    let currentOrderBy: OrderByType = {};
    let lastOrderBy = currentOrderBy;

    fields.forEach((field, index) => {
        if (index === fields.length - 1) {
            lastOrderBy[field] = sortDirection;  // Set the final sort direction
        } else {
            lastOrderBy[field] = {};  // Create a nested object
            lastOrderBy = lastOrderBy[field] as OrderByType;  // Move deeper into the object
        }
    });

    return currentOrderBy;
}
```

Then we get to the api call itself:



```
export async function GET(request: NextRequest) {
    const borrowDate = searchParams.get('borrowDate');
    const returnDate = searchParams.get('returnDate');
    const locationFilter = searchParams.get('location') || '';
    const requestorFilter = searchParams.get('requestor') || '';
    const sortBy = searchParams.get('sortBy') || 'requestDate';  // Default sort field
    const sortDirection = searchParams.get('sortDirection') as Prisma.SortOrder || 'desc';  // Default so
    const offset = parseInt(searchParams.get('offset') || '0');
    const limit = parseInt(searchParams.get('limit') || '10');
    const token = searchParams.get("token") || '';

    const orderBy = createNestedOrderBy(sortBy, sortDirection);

    const user = await prisma.user.findUnique({
        where: {
            firebaseUid: uid,
        },
        include: {
            role: true,
        }
    });

    const decodedToken = await admin.auth().verifyIdToken(token);

    if (!decodedToken) {
        return new Response(JSON.stringify("Unauthorized"), {
            status: 403,
            headers: {
                'Content-Type': 'application/json',
            },
        });
    };

    if (!user) {
        return new Response(JSON.stringify("User not found"), {
            status: 404,
            headers: {
                'Content-Type': 'application/json',
            },
        });
    };

    if (!["Admin", "Supervisor"].includes(user.role.name)) {
        return new Response(JSON.stringify("Forbidden, you don't have the rights to make this call"), {
            status: 403, // Use 403 for Forbidden instead of 404
            headers: {
                'Content-Type': 'application/json',
            },
        });
    };
```

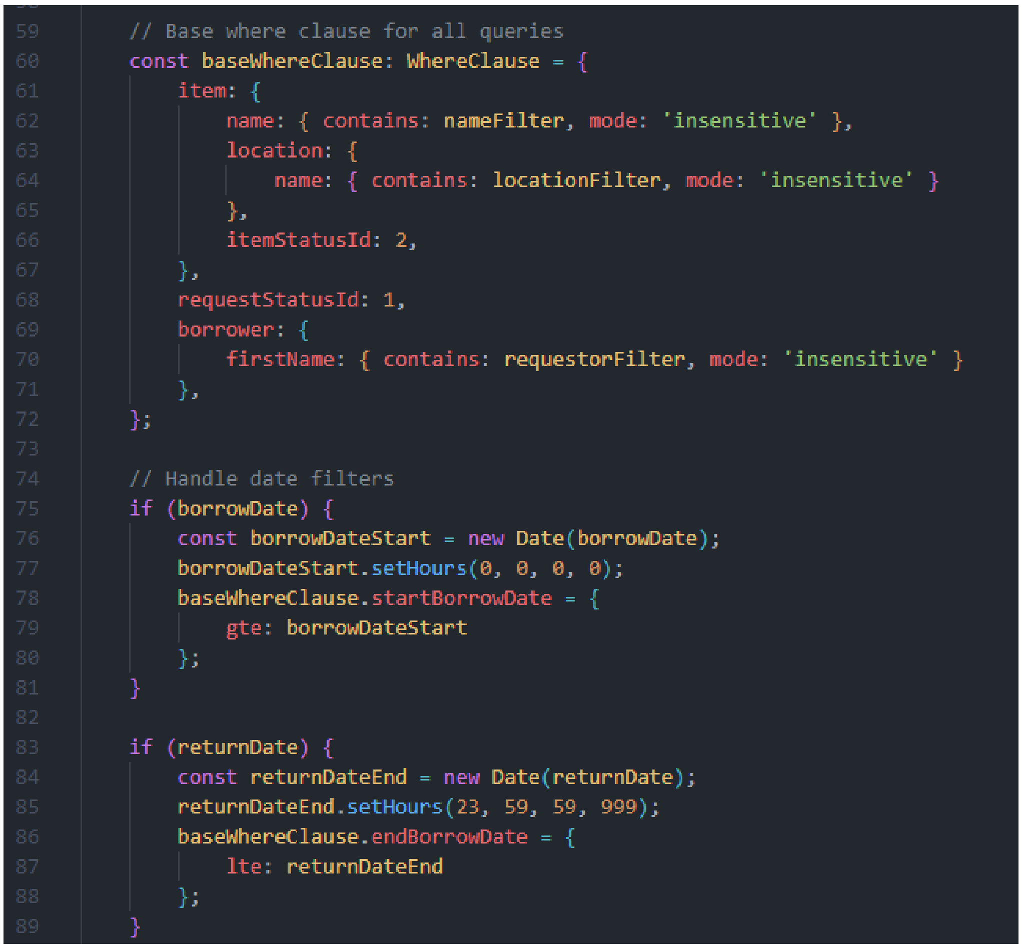
First we export the asynchronous function GET because it is a get request. We gave is searchParameters (queryString) so we will get them all as variables first. The last ones “Offset” and “Limit” are for the infinite scroll effect.

First we check if the user has a valid api token.

After that, the first thing we do is, we check if the user exists meaning we check if the user that makes this request is a real user of the system and has an account. If not, we return an error.

Then we check if the user has the right role to make this api call. When not we return a forbidden error.

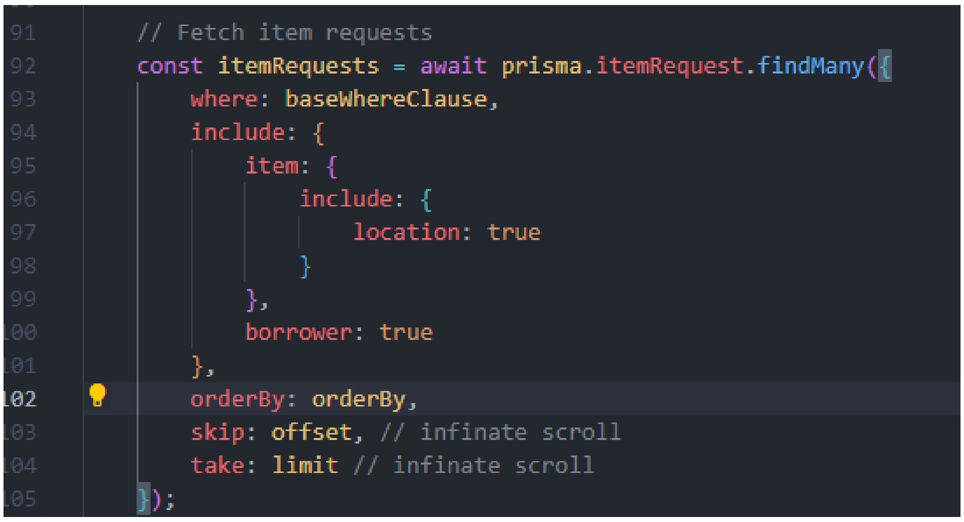
Then we make a WhereClaus you will see this a lot in our application. What this does is it makes us filter based on the filters we entered:



```
    // Base where clause for all queries
    const baseWhereClause: WhereClause = {
        item: {
            name: { contains: nameFilter, mode: 'insensitive' },
            location: {
                name: { contains: locationFilter, mode: 'insensitive' }
            },
            itemStatusId: 2,
        },
        requestStatusId: 1,
        borrower: {
            firstName: { contains: requestorFilter, mode: 'insensitive' }
        },
    };

    // Handle date filters
    if (borrowDate) {
        const borrowDateStart = new Date(borrowDate);
        borrowDateStart.setHours(0, 0, 0, 0);
        baseWhereClause.startBorrowDate = {
            gte: borrowDateStart
        };
    }

    if (returnDate) {
        const returnDateEnd = new Date(returnDate);
        returnDateEnd.setHours(23, 59, 59, 999);
        baseWhereClause.endBorrowDate = {
            lte: returnDateEnd
        };
    }
```

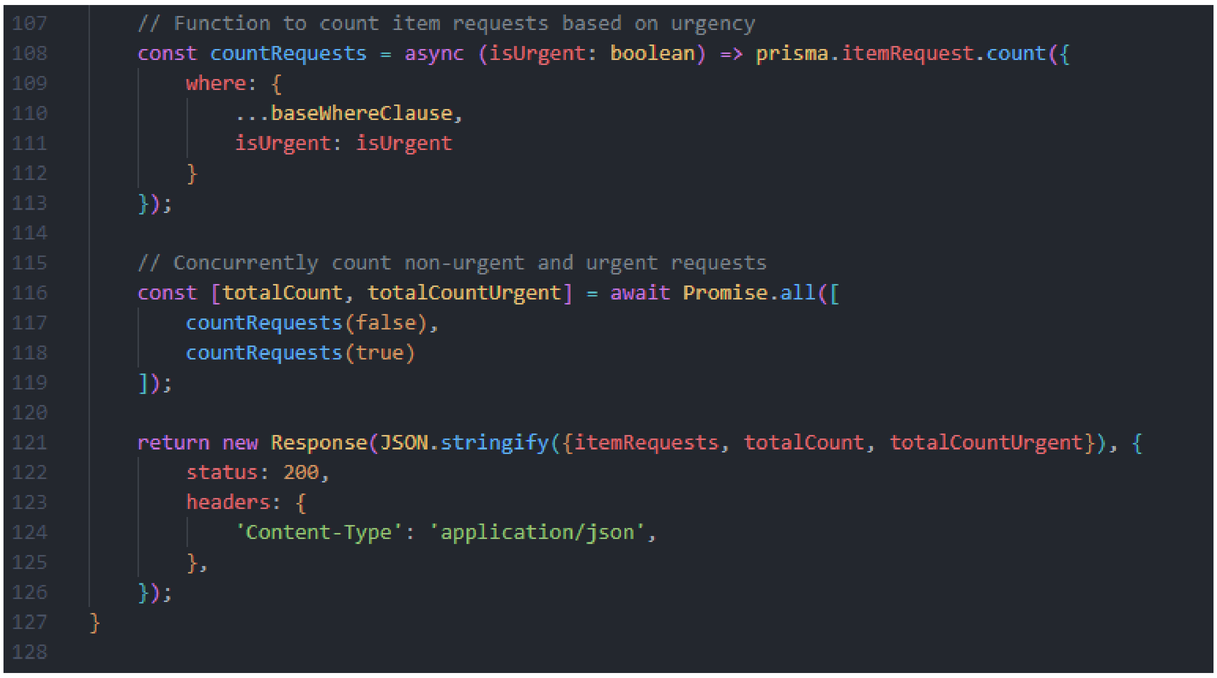
Then because inside the filter you can select dates we check if they are given, if they are we check if they started or ended correctly.

Now the real fetch happens here:



```
    // Fetch item requests
    const itemRequests = await prisma.itemRequest.findMany({
        where: baseWhereClause,
        include: {
            item: {
                include: {
                    location: true
                }
            },
            borrower: true
        },
        orderBy: orderBy,
        skip: offset, // infinate scroll
        take: limit // infinate scroll
    });
```

Here we use prisma and check for many itemRequests. As you can see, we give a lot of variables that we defined earlier.



```
107     // Function to count item requests based on urgency
108     const countRequests = async (isUrgent: boolean) => prisma.itemRequest.count({
109         where: {
110             ...baseWhereClause,
111             isUrgent: isUrgent
112         }
113     });
114
115     // Concurrently count non-urgent and urgent requests
116     const [totalCount, totalCountUrgent] = await Promise.all([
117         countRequests(false),
118         countRequests(true)
119     ]);
120
121     return new Response(JSON.stringify({itemRequests, totalCount, totalCountUrgent}), {
122         status: 200,
123         headers: {
124             'Content-Type': 'application/json',
125         },
126     });
127 }
128
```

In this example we also count the item requests so we can have that number next to "requests" and "urgent requests" that is what happens on line 107-119.

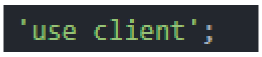
And at the end we return our requests that are filtered and the two counts we made for the normal and urgent requests.

5. Frontend

Next.js what we use as programming language is react based. Meaning if you want to understand how things work make sure you have a solid understanding of React before jumping into the code. We have included a lot of links to documentation to make sure everything is clear.

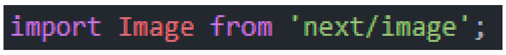
After jumping into the code, you will see that everything is basically React. There are a few things that are unique to next.js and we'll cover them here:

First next.js '[use client'](#) tags you see at some pages. Normally you would try not to use this because this means the client must load the page. If we don't use this the server can pre-load the page, sort of like caching.

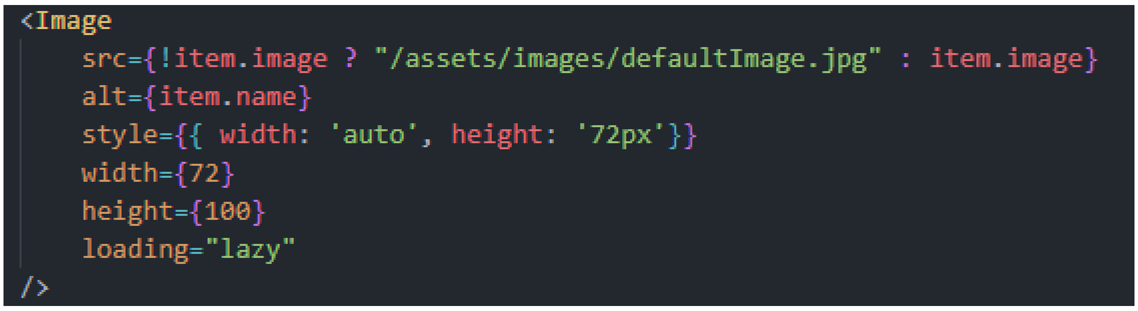


```
'use client';
```

Next, the image tag of next.js itself, there is a lot of documentation available on [their website](#). Basically, this makes it so we can lazy load the image and it's better for performance.

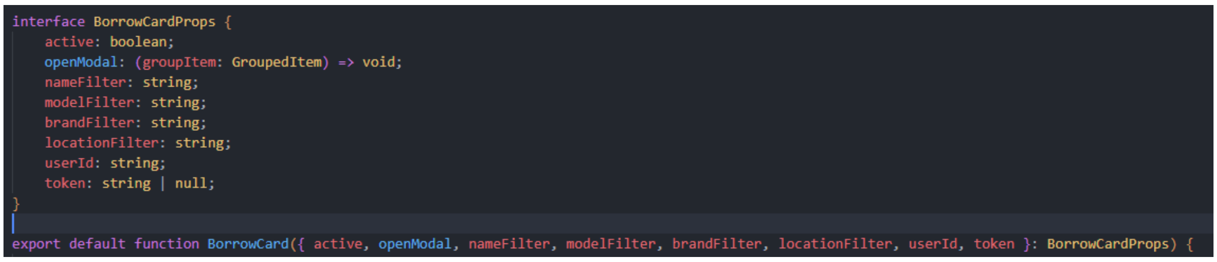


```
import Image from 'next/image';
```



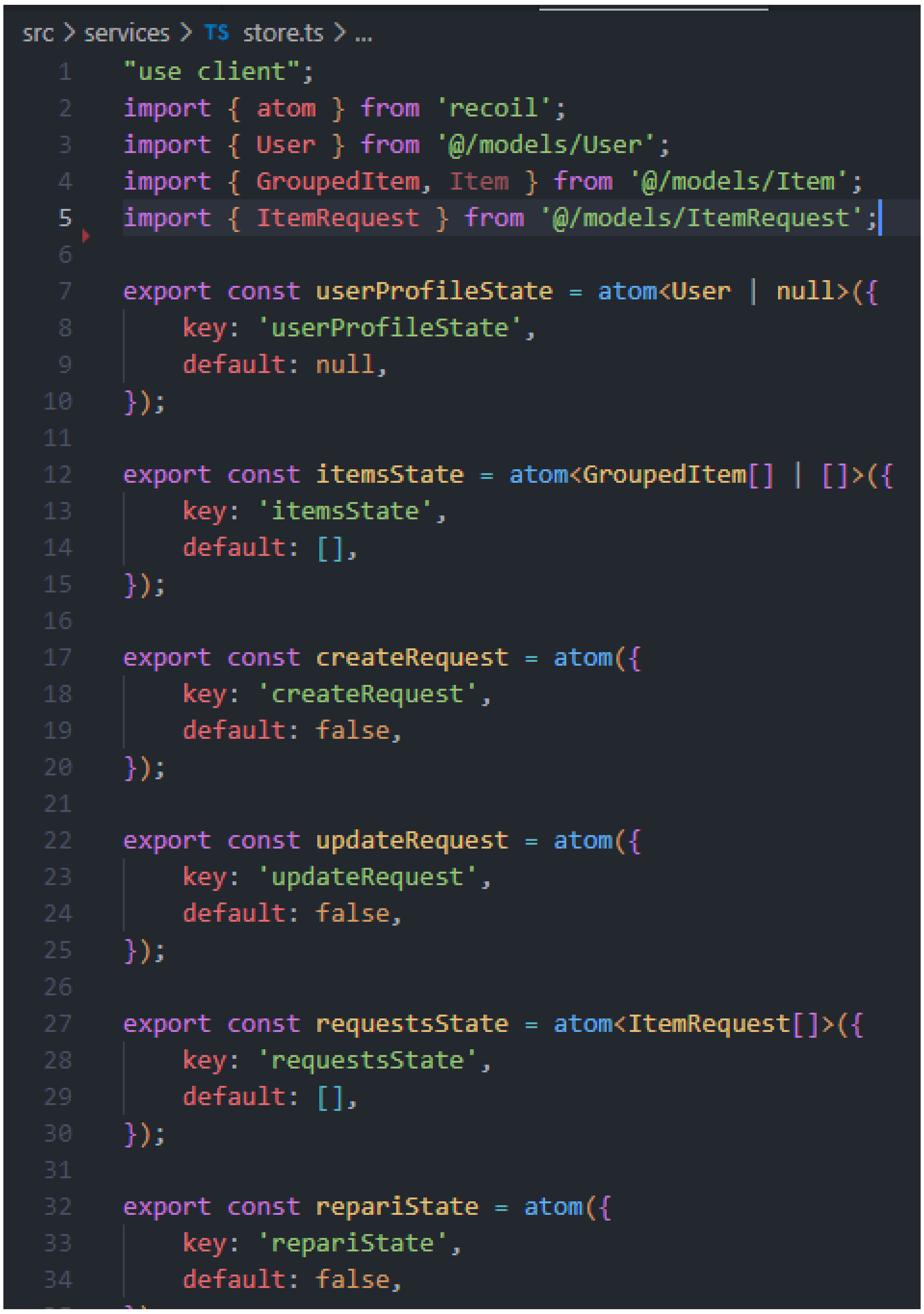
```
<Image
    src={!item.image ? "/assets/images/defaultImage.jpg" : item.image}
    alt={item.name}
    style={{ width: 'auto', height: '72px'}}
    width={72}
    height={100}
    loading="lazy"
/>
```

Next up is a bit of a [typescript](#) thing, you must declare everything. So that's why at the start of each file we have an interface where we define what every variable is.



```
interface BorrowCardProps {
    active: boolean;
    openModal: (groupItem: GroupedItem) => void;
    nameFilter: string;
    modelFilter: string;
    brandFilter: string;
    locationFilter: string;
    userId: string;
    token: string | null;
}

export default function BorrowCard({ active, openModal, nameFilter, modelFilter, brandFilter, locationFilter, userId, token }: BorrowCardProps) {
```

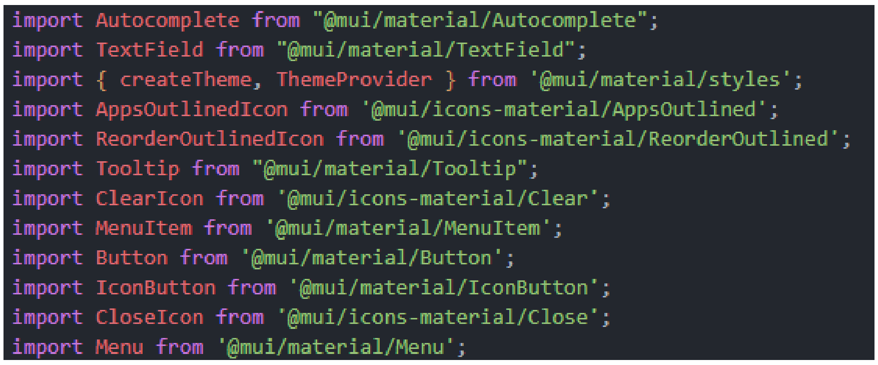
We also used a file called Store.ts it’s in the services directory. This is to handle state changes globally. Normally you should pass props to a component but with recoil state it’s not needed. You store all the states globally and can access and modify them from anywhere. You can find more details on the website of recoil.



src > services > TS store.ts > ...

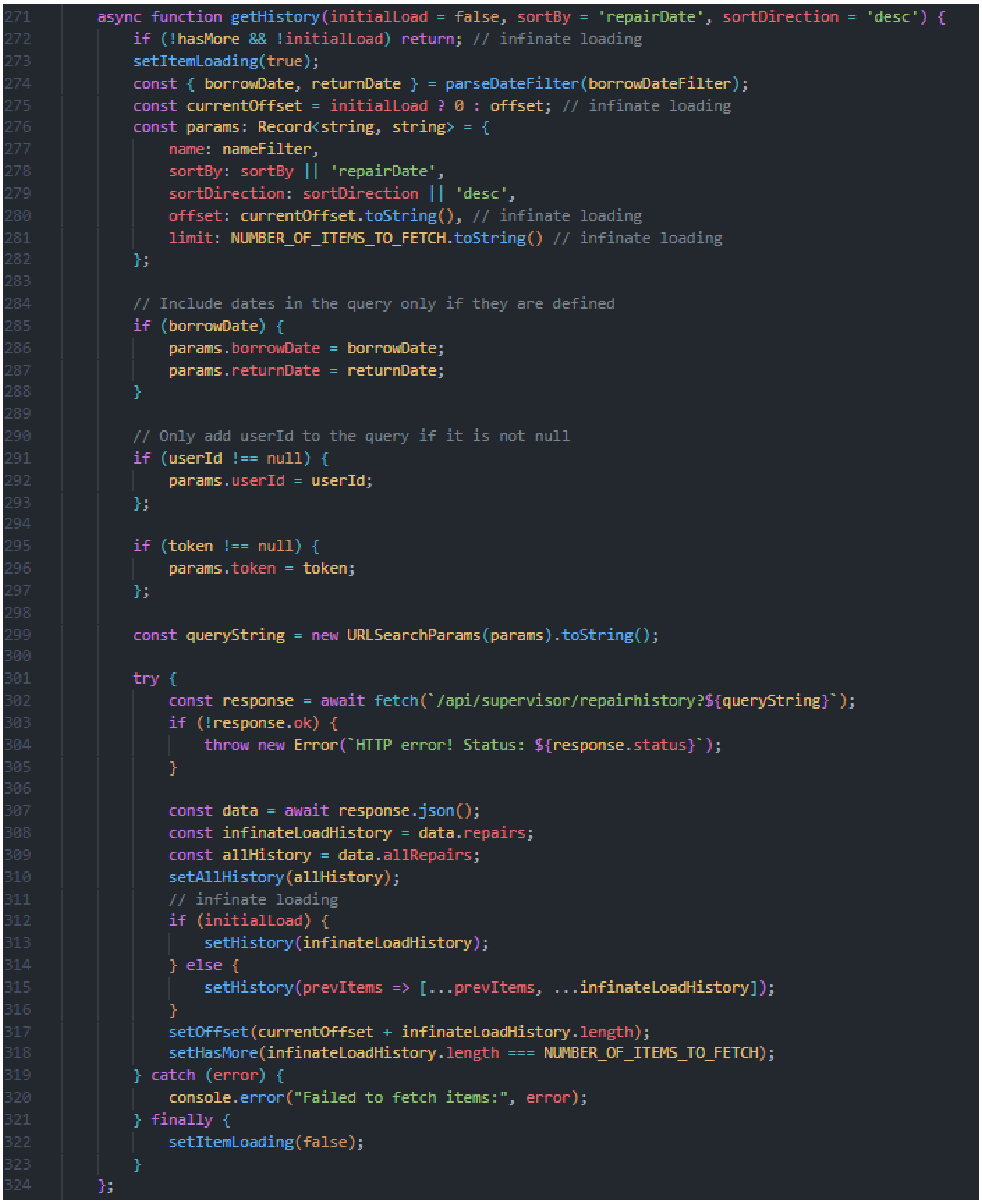
```
"use client";
import { atom } from 'recoil';
import { User } from '@/models/User';
import { GroupedItem, Item } from '@/models/Item';
import { ItemRequest } from '@/models/ItemRequest';

export const userProfileState = atom<User | null>({
    key: 'userProfileState',
    default: null,
});

export const itemsState = atom<GroupedItem[] | []>({
    key: 'itemsState',
    default: [],
});

export const createRequest = atom({
    key: 'createRequest',
    default: false,
});

export const updateRequest = atom({
    key: 'updateRequest',
    default: false,
});

export const requestsState = atom<ItemRequest[]>({
    key: 'requestsState',
    default: [],
});

export const repariState = atom({
    key: 'repariState',
    default: false,
```

You can also see that we used MUI components and icons a lot in our project. Mostly in our filter for the autocomplete fields, input fields, etc. Their documentation is good so if something is unclear you can find an answer on their website.



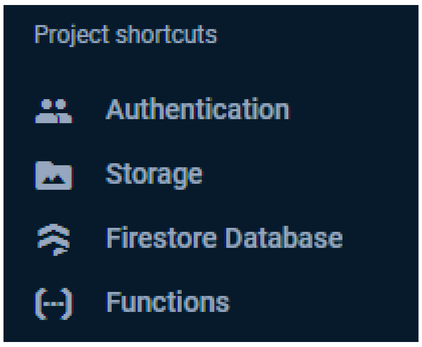
```
import Autocomplete from "@mui/material/Autocomplete";
import TextField from "@mui/material/TextField";
import { createTheme, ThemeProvider } from '@mui/material/styles';
import AppsOutlinedIcon from '@mui/icons-material/AppsOutlined';
import ReorderOutlinedIcon from '@mui/icons-material/ReorderOutlined';
import Tooltip from "@mui/material/Tooltip";
import ClearIcon from '@mui/icons-material/Clear';
import MenuItem from '@mui/material/MenuItem';
import Button from '@mui/material/Button';
import IconButton from '@mui/material/IconButton';
import CloseIcon from '@mui/icons-material/Close';
import Menu from '@mui/material/Menu';
```

Lastly, we'll explain one API call that you make from the front-end. As you can see in the image next to this page this is one of the most complex API calls, we make, that's why we explain this one. The initial load is used for infinite loading (scroll) which you can see on the pages. We also specify the sort direction and on what item we want to sort. Then we pass the dates and make sure our params work that we will use for our filter (276-282). After that we will see if we have a valid borrow date, userId and token set. If it is, we pass all of them to our API call. Next, we check if our response is valid, if it is valid we get all the data from API.



```
    async function getHistory(initialLoad = false, sortBy = 'repairDate', sortDirection = 'desc') {
        if (!hasMore && !initialLoad) return; // infinate loading
        setItemLoading(true);
        const { borrowDate, returnDate } = parseDateFilter(borrowDateFilter);
        const currentOffset = initialLoad ? 0 : offset; // infinate loading
        const params: Record<string, string> = {
            name: nameFilter,
            sortBy: sortBy || 'repairDate',
            sortDirection: sortDirection || 'desc',
            offset: currentOffset.toString(), // infinate loading
            limit: NUMBER_OF_ITEMS_TO_FETCH.toString() // infinate loading
        };

        // Include dates in the query only if they are defined
        if (borrowDate) {
            params.borrowDate = borrowDate;
            params.returnDate = returnDate;
        }

        // Only add userId to the query if it is not null
        if (userId !== null) {
            params.userId = userId;
        };

        if (token !== null) {
            params.token = token;
        };

        const queryString = new URLSearchParams(params).toString();

        try {
            const response = await fetch(`/api/supervisor/repairhistory?${queryString}`);
            if (!response.ok) {
                throw new Error(`HTTP error! Status: ${response.status}`);
            }

            const data = await response.json();
            const infinateLoadHistory = data.repairs;
            const allHistory = data.allRepairs;
            setAllHistory(allHistory);
            // infinate loading
            if (initialLoad) {
                setHistory(infinateLoadHistory);
            } else {
                setHistory(prevItems => [...prevItems, ...infinateLoadHistory]);
            }
            setOffset(currentOffset + infinateLoadHistory.length);
            setHasMore(infinateLoadHistory.length === NUMBER_OF_ITEMS_TO_FETCH);
        } catch (error) {
            console.error("Failed to fetch items:", error);
        } finally {
            setItemLoading(false);
        }
    };
```

6. Firebase

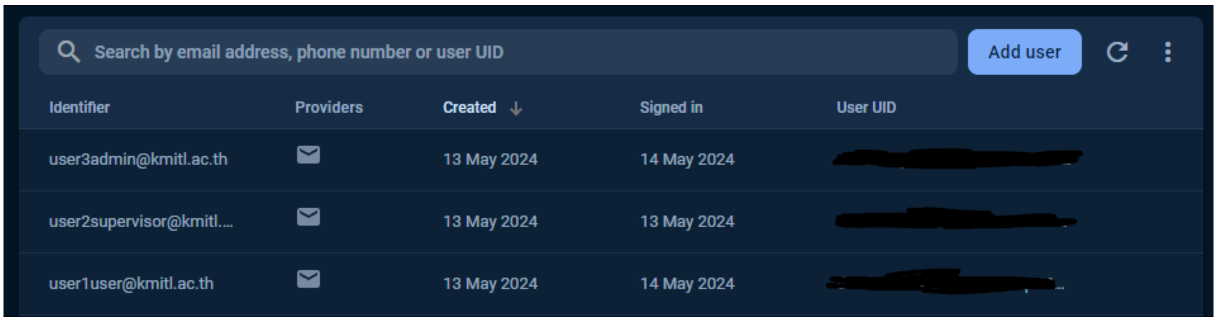


Because we also have quite a few things on firebase I shall briefly explain them here. The documentation of firebase is also very clear so if something is unclear you can check that out.

First of all, these are the four tools we used in our firebase project.

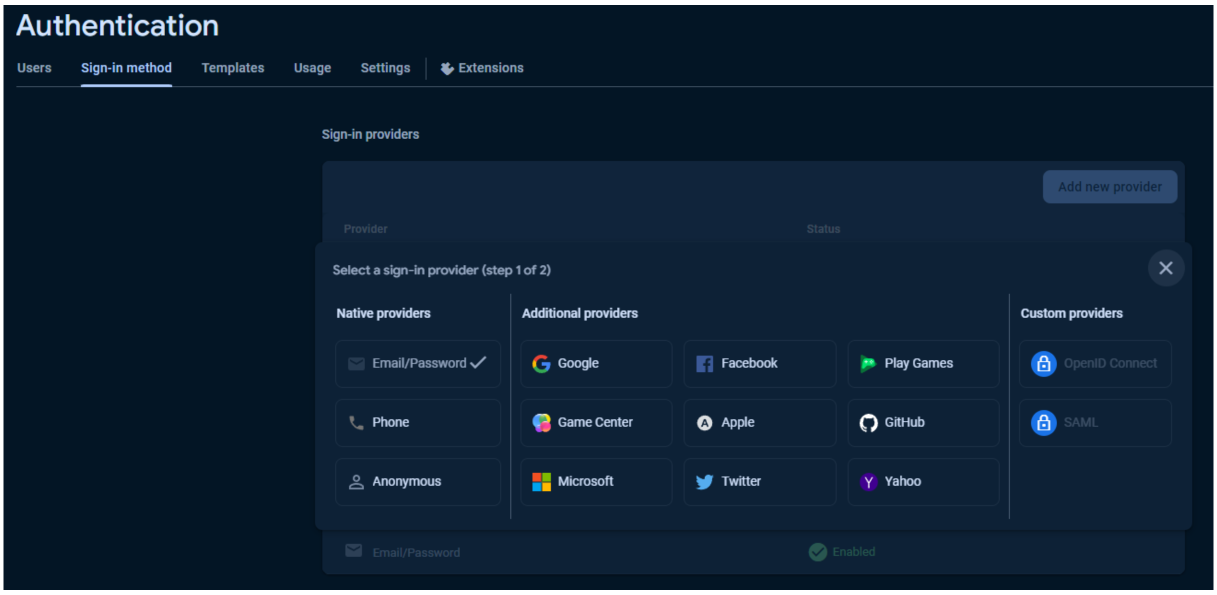
- Authentication for user authentication
- Storage for file and image storage
- Firestore database for notification storage
- Functions as a setup for maybe in the feature have time-based functions.

Clicking on Authentication you will see all the users that have made an account on our program.



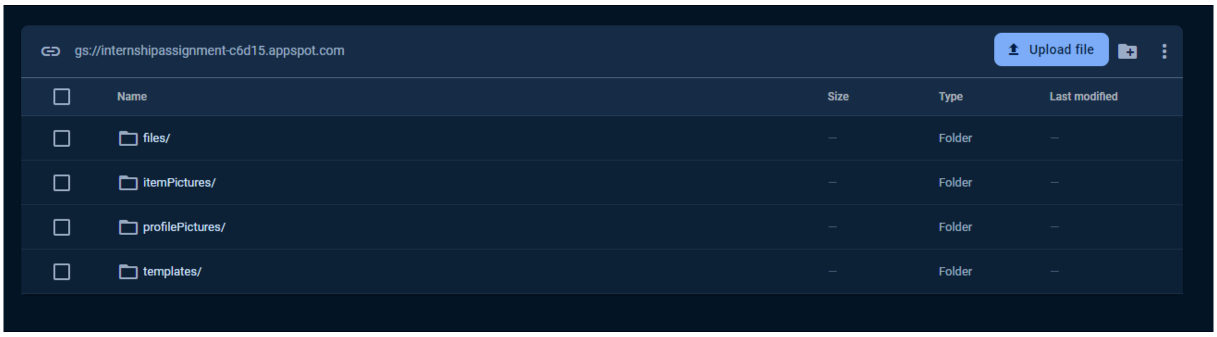
Identifier	Providers	Created	Signed in	User UID
user3admin@kmitl.ac.th	✉	13 May 2024	14 May 2024	[redacted]
user2supervisor@kmitl....	✉	13 May 2024	13 May 2024	[redacted]
user1user@kmitl.ac.th	✉	13 May 2024	14 May 2024	[redacted]

For now the only sign in method we have selected is email / password but it is perfectly possible to add more in the future. You can do this by going to the tab "sign-in method". And clicking on new provider.



Next up is the storage. You can view this as an s3 bucket from AWS because it is similar. I'll explain all the folders:

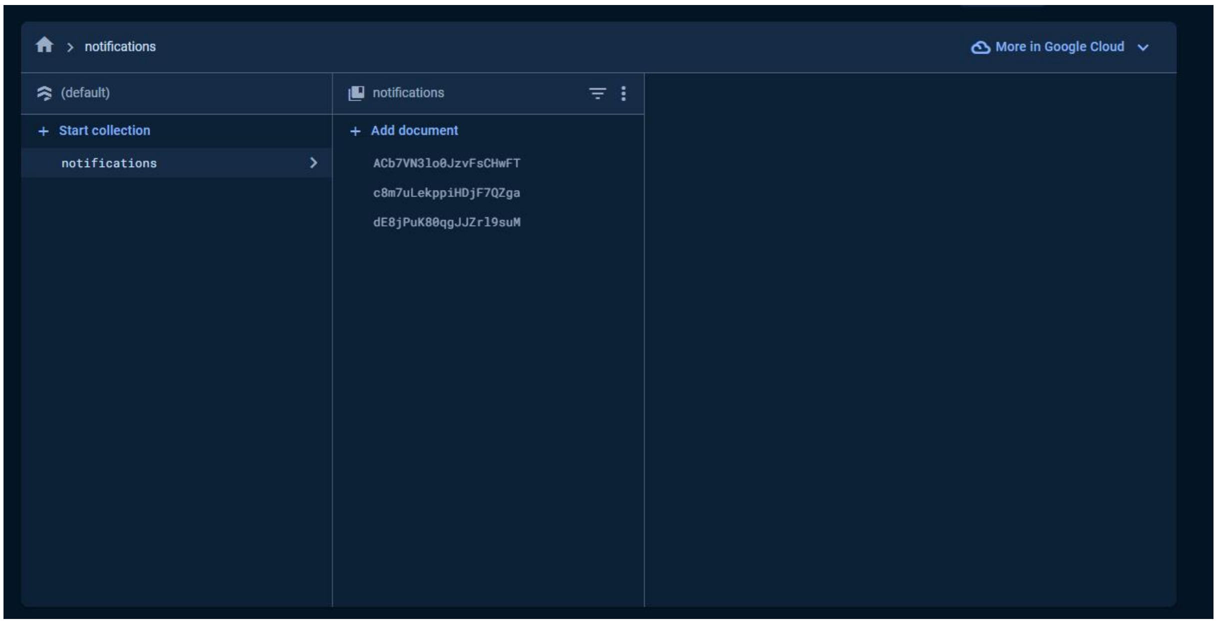
- Files: This is used for the urgent borrow requests made by the students. Here all the files that they uploaded will be stored.
- ItemPicutres: Here all the photos of the items will be stored.
- ProfilePicutres: Here all the photos of the user profiles will be stored.
- Templates: Here the template for the urgent borrow request will be stored.



gs://internshipassignment-c6d15.appspot.com

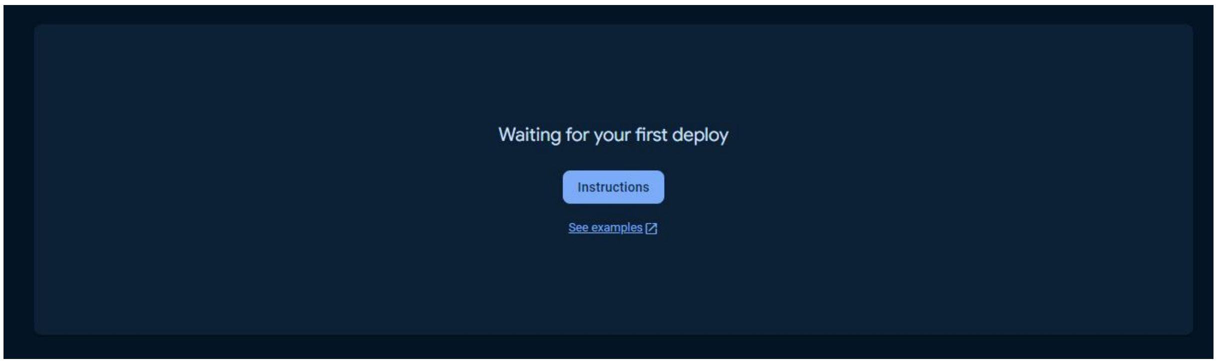
Name	Size	Type	Last modified
files/	–	Folder	–
itemPictures/	–	Folder	–
profilePictures/	–	Folder	–
templates/	–	Folder	–

Next up is the Firestore database. This is where all the notifications will be stored. We could have done this locally as well by adding a standalone table to our database (like parameters) and making a call to that table anytime somethings get done on the system. But we chose not to do this because we didn't know how many resources the server was going to have. If you prefer to make it locally in the future, you can but there will be quite a lot of extra requests being made. Thus, for safety reasons we made it in the cloud, you have 1GB of free storage. This gives you by estimate between 3-4 million records you can store in this database.

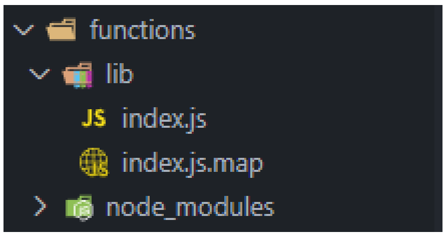


```
ACb7VN3lo0JzvFsCHwFT
c8m7uLekppiHDjF7QZga
dE8jPuK80qgJJZrl9suM
```

Functions is something we haven't used but we made a start either way. This will be used (if needed) to write functions that run time-based functions. Meaning if you want for example to give a notification 24hours before a request has to be brought back. You will need this. Because that function has to constantly check if the current time is less than 24 hours before the return date.


Waiting for your first deploy

To give a little bit of extra information. You will see that there are 2 folders in the functions folder. One for nodes modules and one with lib.



Lib is used to setup the files that will run as functions. You can find more information about that in the documentation.