# Tartu Narva College

Databases

NEXT

Author: Sergei Ivanov

Major: IT systems development

Subject: Data Bases

Teacher: Chen-Wan Lin

21.08.2023

Table of Contents

[Tartu Narva College 1](#_Toc153575860)

[Abstract 3](#_Toc153575861)

[Revision history 4](#_Toc153575862)

[Introduction 5](#_Toc153575863)

[Background Information 5](#_Toc153575864)

[Research Questions 5](#_Toc153575865)

[Objectives 6](#_Toc153575866)

[Methodology overview 6](#_Toc153575867)

[MySQL 6](#_Toc153575868)

[Introduction 6](#_Toc153575869)

[Keys 7](#_Toc153575870)

[Homework 7](#_Toc153575871)

[Part I 7](#_Toc153575872)

[Part II 10](#_Toc153575873)

[TODO App models 10](#_Toc153575874)

[Schema 10](#_Toc153575875)

[Implementation of the schemas 10](#_Toc153575876)

[Conclusion 15](#_Toc153575877)

[Neural networks 15](#_Toc153575878)

[Flask Full-stack application 16](#_Toc153575879)

[MongoDB 16](#_Toc153575880)

[Flask: REST API 17](#_Toc153575881)

[What was the bad decision? 20](#_Toc153575882)

[Logical Database Design 21](#_Toc153575883)

[Revise our models in terms of SQL 21](#_Toc153575884)

[API Design and Middleware 22](#_Toc153575885)

[Introduction 22](#_Toc153575886)

[Flask middleware 29](#_Toc153575887)

[Conclusion 31](#_Toc153575888)

[References 35](#_Toc153575889)

# Abstract

This paper discusses how to create NEXT application and avoid entering data manually, but using databases instead, such as Prisma which is abstraction from MySQL, and provided some automation. For example, MySQL does not automatically escape queries, while Prisma, provided by MySQL, does so for a developer. Hence, Prisma helps the developer to protect its app against notorious SQL injections which are actual even till these days.

Apart from Prisma, we also implement various applications using MongoDB, which is known as NOSQL database. Keywords: Mongoose, mongod, MySQL, SQLite, PostgreSQL, NEXT, React, TypeScript, JavaScript, CSS, HTML, Layout, Response, Request, Props, Children, and many others.

# Revision history

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Organization/Point of Contact | Description of Changes |
| 1.0 | 29.09.2023 | Sergei Ivanov | Initialization of the paper |
| 2.0 | 11.11.2023 | Sergei Ivanov | Homework 3 |
| 3.0 | 15.12.2023 | Sergei Ivanov | Final Project |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Tab. 1.: The log of changes committed to this document

# Introduction

## Background Information

The data base is not any server, but an organized collection of logically connected data gathered to carry out data analysis to support scientific or commercial research. In this paper, I aim to unravel the multifaceted nature of databases and explore their applications, functionalities, and integration with modern JavaScript frameworks.

One of the most fundamental aspects that distinguishes a database from CSV (comma separated values) is organization. It uses specific syntax to operate over data in terms of CRUD: Create, read, update, and delete. To define data structure,

Secondly, a database is not any data structure. The difference is that data structure is abstraction while a database implements it. For example, we can organize our data as a red-black tree, and then implement it using SQL.

Thirdly, there are server and serverless databases, which means that syntax might vary from database to another, as well as the interaction between a developer or a user and a database. To illustrate that, we can mention MySQL and SQLite: the server and serverless ones, respectively. The first one does require a server to host a data, commit changes, and maintain it, while the other one does not.

## Research Questions

My questions:

1. What is a database?
2. What is the purpose of a database?
3. What are applications of a database?
4. What databases exist?
5. What are examples of using databases?

## Objectives

My primary goal is to demonstrate how to use databases in combination with modern JS framework known as NEXT and built on top of React. In doing so, I explore various concepts related to databases and implement a few applications leveraging the functionality of Prisma, Mongoose, Oracle, mySQL, and SQLite.

Hence, my objectives are

1. Answer the research questions.
2. Introduce and demonstrate daemons (mongod).
3. Create servers and connect to databases.
4. Operate over data and introduce data structures that are implemented by database.

## Methodology overview

To achieve my goals, I have to conduct research on various data bases referring to academic papers and official documentations provided by Mongoose, NEXT, VERCEL, Prisma, and Oracle. Therefore, I can write my methods:

1. Research
2. Study
3. Analyze
4. Demonstrate
5. Build applications.

# MySQL

## Introduction

MySQL is lower-level RDBMS (Relational Database Management System) when compared to high-level ORM (Object Relational Modeling) such as Prisma, or ODM (Object Data Modeling) such as MongoDB. It is owned by Oracle company.

Remark. Even though MySQL is owned by Oracle company, Oracle DB is different database management system.

## Keys

In MySQL system, there the following commonly used keys:

1. Primary: for unique data; it is unique and does not contain NULL value.
2. Alternate: also, unique key used for complementing primary key.
3. Candidate: it is a unique identifier without unnecessary redundancy.
4. Foreign: It establish a link among tables; an attribute in a table that refers to the primary key.
5. Super: A composite candidate key,

# Homework

## Part I

My first homework assignment is that I must determine primary, alternate, super, candidate and foreign keys for the following tables:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| propertyNo | street | city | postcode | type | rooms | rent | ownerNo | staffNo | branchNo |
| PA14 | 16 Holhead | Aberdeen | AB7 5SU | House | 6 | 650 | CO46 | SA9 | B007 |
| PL94 | 6 Argyll St | London | NW2 | Flat | 4 | 400 | CO87 | SL41 | B005 |
| PG4 | 6 Lawrence St | Glasgow | G11 9QX | Flat | 3 | 350 | CO40 | - | B003 |
| PG36 | 2 Manor Rd | Glasgow | G32 4QX | Flat | 3 | 375 | CO93 | SG37 | B003 |
| PG21 | 18 Dale Rd | Glasgow | G12 | House | 5 | 600 | CO87 | SG37 | B003 |
| PG16 | 5 Novar Dr | Glasgow | G12 9AX | Flat | 4 | 450 | CO93 | SG14 | B003 |

Tab. 2: PropertyForRent

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| clientNo | fName | lName | telNo | prefType | maxRent | eMail |
| CR76 | John | Kay | 0207-774-5632 | Flat | 425 | [john.kay@gmail.com](mailto:john.kay@gmail.com) |
| CR56 | Aline | Stewart | 0141-848-1825 | Flat | 350 | [astewart@hotmail.com](mailto:astewart@hotmail.com) |
| CR74 | Mike | Ritchie | 01475-392178 | House | 750 | [mritchie@yahoo.co.uk](mailto:mritchie@yahoo.co.ukm) |
| CR62 | Mary | Tregear | 01224-196720 | Flat | 600 | [maryt@hotmail.co.uk](mailto:maryt@hotmail.co.uk) |

Tab. 3: Client

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ownerNo | fName | lName | address | telNo | eMail | password |
| CO46 | Joe | Keogh | 2 Fergus Dr, Aberdeen AB2 7SX | 01224-861212 | [jkeogh@lhh.com](mailto:jkeogh@lhh.com) | \*\*\*\*\*\*\*\* |
| CO87 | Carol | Farrel | 6 Achray St, Glasgow G32 9DX | 0141-357-7419 | [cfarrel@gmail.com](mailto:cfarrel@gmail.com) | \*\*\*\*\*\*\*\* |
| CO40 | Tina | Murphy | 63 Well St, Glasgow G42 | 0141-943-1728 | [tinam@hotmail.com](mailto:tinam@hotmail.com) | \*\*\*\*\*\*\*\* |
| CO93 | Tony | Shaw | 12 Park Pl, Glasgow G4 0QR | 0141-225-7025 | [tony.shaw@ark.com](mailto:tony.shaw@ark.com) | \*\*\*\*\*\*\*\* |

Tab. 3: PrivateOwner

As it can be seen, some keys are reused in other tables, and other ones are unique. Let us determine those keys:

1. Primary key: propertyNo, clientNo, ownerNo;
2. Candidate key: propertyNo, clientNo, ownerNo;
3. Alternate key: postcode, telNo, eMail;
4. Foreign key: ownerNo, staffNo, branchNo;
5. Super key: (propertyNo, street), (clientNo, fName, lName, rent, maxRent);
6. Regular keys: fName, lName, rent, maxRent, rooms, city, type, prefType, passwordHash.

Let us define the models for these tables, using SQL syntax:

CREATE TABLE PropertyForRent

(  
 propertyNo VARCHAR(255),

city VARCHAR(255),

postcode VARCHAR(255) UNIQUE,

type ENUM(‘house’, ‘flat’),

rooms INT,

rent INT,

ownerNo VARCHAR(255) UNIQUE,

staffNo VARCHAR(255),

branchNo VARCHAR(255),

PRIMARY KEY (propertyNo)

);

CREATE TABLE Client (

clientNo VARCHAR(255) UNIQUE,

fName VARCHAR(255),

lName VARCHAR(255),

telNo VARCHAR(255) UNIQUE,

prefType ENUM(‘house’, ‘flat’),

maxRent INT,

eMail VARCHAR(255) UNIQUE,

PRIMARY KEY (clientNo)

);

CREATE TABLE PrivateOwner (

ownerNo VARCHAR(255) UNIQUE,

fName VARCHAR(255),

lName VARCHAR(255),

address VARCHAR(255),

telNo VARCHAR(255) UNIQUE,

eMail VARCHAR(255) UNIQUE,

passwordHash VARCHAR(255),

PRIMARY KEY (ownerNo)

);

## Part II

### TODO App models

In this section, we consider TODO’s app schemas for databases and try to plot them in order to gain better understanding about database models.

The first critical aspect to keep in mind is the fact that database model refers to the architecture of the database, not to any schema that is used by the database. For example, Prisma is object-relational mapping, while MongoDB is object data modeling.

### Schema

For our application, we have the following schemas:

* User (in case we implement authentication).
* Todo – the task to do.
* Project – this is a part of normalization because Todo can be a project, but we split it into another model, called, project.
* Label – categorizing and filtering.
* Comment
* Attachment
* History
* Notification

### Implementation of the schemas

The first two models are User and Post, then Todo, Project, Comment, Notification, Label, Attachment and History.

A computer screen shot of a program code

Description automatically generated

Fig. 1.: User and Post Models

A screen shot of a computer program

Description automatically generated

Fig. 2.: Todo and Project models

A screen shot of a computer program

Description automatically generated

Fig. 3.: Comment and Notification models

A screen shot of a computer program

Description automatically generated

Fig. 4.: Label and Attachment models

A computer screen with text

Description automatically generated

Fig. 5.: History model

### Conclusion

As you can see, we can break down our logic into models used by other models.

### Neural networks

The neural network is implemented using `pytorch`, `tensorflow`, `keras` Python libraries. In C++, developers usually write the code that is utilized in these fundamental libraries. Therefore, in the context of TODO app, neural network is used as an API hosted by Flask:

* Flask server receives a request.
* Input data is sent to neural network.
* Neural network sends output back to Flask.
* NEXT.js receives data from API, Flask.

To illustrate this, we can create models that has the attribute `price` (also model) recognized by the neural network, while the network itself does not require its own model.

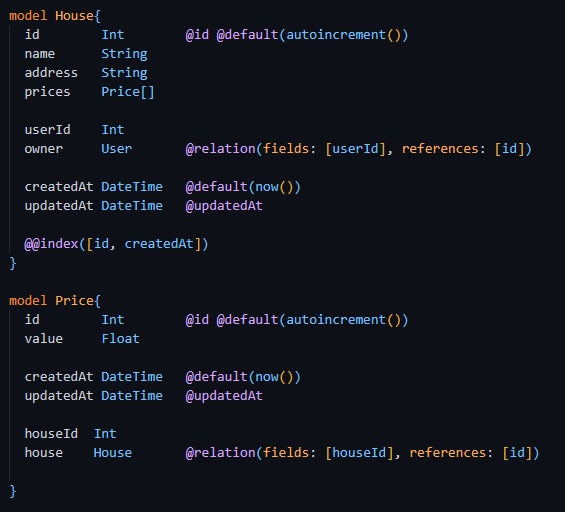


Fig. 6.: Neural Network processes Prices data

# Flask Full-stack application

## MongoDB

To submit my project for Python course, I was assigned to create a Python project. I chose Flask because I wanted to use it. The website can be found on my GitHub repository: [Python project](https://github.com/derweisskrag/Python-CV-New). The purpose of mentioning the project within the scope of databases course is that I used MongoDB for that project and implemented simple login and signup pages.

Model of User is as follows:

from mongoengine import Document, StringField, EmailField

class User(Document):

name = StringField(required=True)

email = EmailField(required=True, unique=True)

password = StringField(required=True)

As can be seen, we are using `mongoengine` library, and not `pymongo` which is official MongoDB Python library. The syntax reminds us of Python data classes. This is a basic schema for user unlike Prisma’s one. However, we can make it more complex by adding other models to it such as houses. For that, we would use ListField and pass to it a ReferenceField as follows:

class House(Document):

address = StringField(required=True)

name = StringField(required=True)

class User(Document):

name = StringField(required=True)

email = EmailField(required=True, unique=True)

password = StringField(required=True)

houses = ListField(ReferenceField(House))

## Flask: REST API

This API designs include:

* REST API
* gRCP
* GraphQL

In my project, I used REST API because it is straightforward approach of handling `server actions`. In fact, any specified route `/api/users/create-user` is a server route which is not visited by humans unless developers test their functionality. Once the action is completed, server redirects a user to the target page or alerts it about the success or failure.

In Flask, we can use a decorator provided by Flask library: `app.get(route, methods)`. As you can see, Flask works with REST API which also uses HTTP protocol to handle all of server actions defined within the protocol. For example, the `POST` method can be used to create a user, and the `GET` method can be used for retrieving the information from databases.

To ensure the security against SQL injections, Flask uses `escaped` function to secure the route. In NEXT.js, however, it is handled automatically by Prisma due to higher level of abstraction.

The first route is to create a user:

@app.route('/api/users/create-user', methods=['POST'])

def create\_user():

print(request.form)

username = request.form.get('username')

email = request.form.get('email')

password = request.form.get('password')

repeat\_password = request.form.get('repeat-password')

if password != repeat\_password:

return jsonify({"error": 'Passwords do not match!'}), 400

# Create a new User instance

new\_user = User(name=username, email=email, password=password)

try:

new\_user.save() # Save the new user to the database

return 'User created successfully!'

except Exception as e:

return f'Error creating user: {str(e)}', 500 # Return an error message if something goes wrong

The module `request` is essential. In NEXT, it is just `request` object of NextApiRequest type in `next/server`. From this object, we can get everything about the request: method, type, body and so forth. The check for the password is actually not correct because for that, you would use `Middleware` to process data before sending it to the server. However, for the simple project, it was sufficient.

User has been created by calling the model constructor and passing all the data. In Python, we could do it by `User(\*\*props)` where `props` is a dictionary that we called `props` and it has all the information about the user that we are going to create. In Prisma, for this purpose, we mark the function with `use server` as of NEXT 14 and call the asynchronous Prisma function to create a user with properties we fetch from a form. For this purpose, you can already make sure that the form which is UI/UX component has all the valid data by using React hooks, so Login page which is a server component in `/app/login/page.tsx` can get data as ServerSideProps.

The second important aspect is how Flask handles home page:

app.get("/")

def home():

user\_name = session.get('name') if 'logged\_in' in session else 'Guest'

return render\_template('home.html', user\_name=user\_name)

As you can see, home is just `root` and by utilizing `session` we can retrieve all the data about the current user and then make use of it in `Jinja2` templates. In NEXT, you would also use `session` to implement the same functionality.

The login functionality is also an API route with the method `GET`:

@app.route('/login', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

email = request.form.get('email')

password = request.form.get('password')

user = User.objects(email=email).first()

if user and user.password == password:

session['logged\_in'] = True

session['email'] = email

session['name'] = user.name

return redirect(url\_for('dashboard'))

else:

'Incorrect email/password'

return render\_template('login.html')

Here, we retrieve the data from MongoDB (all clusters can be viewed inside database) and check if they coincide with the data retrieved from the form using `request` module.

Lastly, we can implement “/dashboard” and “logout” functionalities:

@app.route('/logout', methods=['GET', 'POST'])

def logout():

session.clear()

return redirect(url\_for('home'))

@app.route('/dashboard')

def dashboard():

if 'logged\_in' in session:

user\_name = session.get('name')

return render\_template('dashboard.html', user\_name=user\_name)

else:

return redirect('login')

## What was the bad decision?

Let us list all the necessary modules that were used in the project:

from flask import Flask, render\_template, request, session, redirect, url\_for

from mongoengine import connect

from models.user import User

from flask import jsonify

import json

from resume.create import create\_cv

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'my$key'

# Connect to your MongoDB database

connect(

'CV-APP',

host='mongodb+srv://<username>:<password>@cv-app.cn4g9ri.mongodb.net/?retryWrites=true&w=majority')

In NEXT.js, you would use `.env` where all your environment variables (e.g., `NEXT\_PUBLIC\_MYVAR=1`) are accessed by `process.env.NEXT\_PUBLIC\_MYVAR`. In Python, you would use `environ` from `sys`.

# Logical Database Design

## Revise our models in terms of SQL

Let us rewrite our Full-stack application models in terms of SQL:

CREATE TABLE USERS (

USER\_ID INT PRIMARY KEY,

NAME VARCHAR(50) NOT NULL,

EMAIL VARCHAR(100) UNIQUE NOT NULL,

PASSWORD\_HASH VARCHAR(128) UNIQUE NOT NULL,

HOUSE\_ID INT, - - foreign key

CONSTRAINT USER\_HOUSE FOREIGN KEY (HOUSE\_ID) REFERENCES HOUSE(HOUSE\_ID)

)

CREATE TABLE HOUSE(

HOUSE\_ID INT PRIMARY KEY,

NAME VARCHAR(50) NOT NULL,

ADDRESS VARCHAR(75) NOT NULL,

)

## API Design and Middleware

### 

### Introduction

In Flask, we can use `decorator` to validate data before submitting to the server. This is because decorators usually run before function execution. For example,

def add(a: int, b: int) -> int:

return a + b

def main():

result = add(2, 2)

print(result)

# Run only if not module but entry point of a program

if \_\_name\_\_ == “\_\_main\_\_”:

main()

To add a decorator:

from functools import wraps

# implement a decorator

def decorator(func: callable):

@wraps(func)

def wrapper(\*args, \*\*kws):

result = func(\*args, \*\*kws)

return result \* 2

return wrapper

Now we can use this decorator to decorate `add` function:

@decorator

def add(a: int, b: int) -> int: …

Please, notice that this decorator does not accept any parameters, but we can make it do so by nesting our functions as follows:

def power(num: int):

def decorator(func: callable):

@wraps(func)

def wrapper(\*args, \*\*kws):

result = func(\*args, \*\*kws)

return result \*\* num

return wrapper

return decorator

After we modified the decorator, we can pass a parameter to the decorator, which is an integer:

@power(2) # raise the result to the power of 2: (2 + 2) \*\* 2 = 16

def add\_and\_pow(a: int, b: int) -> int:

return a + b

The entire code:

from functools import wraps

def decorator(func: callable):

@wraps(func)

def wrapper(\*args, \*\*kws):

result = func(\*args, \*\*kws)

return result \* 2

return wrapper

def power(num: int):

def decorator(func: callable):

@wraps(func)

def wrapper(\*args, \*\*kws):

result = func(\*args, \*\*kws)

return result \*\* num

return wrapper

return decorator

@decorator

def add(a: int, b: int) -> int:

return a + b

@power(2) # raise the result to the power of 2: (2 + 2) \*\* 2 = 16

def add\_and\_pow(a: int, b: int) -> int:

return a + b

def main():

result = add(2, 2)

print(result)

# print the result of the second function

print(add\_and\_pow(2, 2))

if \_\_name\_\_ == "\_\_main\_\_":

main()

There is an OOP approach to implement decorators by using `\_\_call\_\_` which makes our object callable. That is, we can make `our class` callable for a specific reason other than instantiating the object (normally, `User(name=”Pavel”, age=30)` would create an instance of the class `User` with attributes `name` and `age` accessible via the instance object: `pavel.name`, `pavel`). I include some examples:

# Let us implement our first decorator

class Power:

def \_\_init\_\_(self, arg):

self.\_arg = arg

def \_\_call\_\_(self, params):

result = self.\_arg(params)

return result \*\* 2

@Power

def multiply(args):

result = 1

for item in args:

result \*= item

return result

print(multiply([2, 2])) #16

class decorator:

def \_\_init\_\_(self, exp):

self.exp = exp

def \_\_call\_\_(self, func):

def wrapper(\*args, \*\*kwargs):

result = func(\*args, \*\*kwargs)

return self.power(result)

return wrapper

def power(self, result):

return result \*\* self.exp

@decorator(2)

def sum\_two(a, b):

return a + b

print(sum\_two(2, 2)) # Output: 16

# Utility class

class MathUtils:

@staticmethod

def calculate\_sum(numbers):

return sum(numbers)

@staticmethod

def calculate\_average(numbers):

return sum(numbers) / len(numbers)

@staticmethod

def find\_maximum(numbers):

return max(numbers)

@staticmethod

def find\_minimum(numbers):

return min(numbers)

@staticmethod

def sort\_numbers(numbers):

return sorted(numbers)

# Refactored class

class NumberOperations:

def \_\_init\_\_(self, numbers):

self.numbers = numbers

@property

def get\_data(self):

return {

'sum': MathUtils.calculate\_sum(self.numbers),

'average': MathUtils.calculate\_average(self.numbers),

'maximum': MathUtils.find\_maximum(self.numbers),

'minimum': MathUtils.find\_minimum(self.numbers),

'sorted\_numbers': MathUtils.sort\_numbers(self.numbers)

}

# Usage:

numbers = [10, 5, 20, 15, 25]

operations = NumberOperations(numbers)

data = operations.get\_data

print(data['sum']) # Output: 75

print(data['average']) # Output: 15.0

print(data['maximum']) # Output: 25

print(data['minimum']) # Output: 5

print(data['sorted\_numbers']) # Output: [5, 10, 15, 20, 25]

def dec(func):

def wrapper():

print("@@@@@@@@@@@@@@@")

func()

print("@@@@@@@@@@@@@@@")

return wrapper

@dec

def test():

print("Hello, World!")

decor = dec(test)

decor()

def example():

class pow:

def \_\_init\_\_(self, arg):

self.arg = arg

def \_\_call\_\_(self):

def wrapper(\*args, \*\*kwargs):

result = self.arg(\*args, \*\*kwargs)

return result

return wrapper

return pow

### Flask middleware

We assume the basic set up in `app.py` (make sure that you have the `Flask` installed! If not, install it using `python3 -m pip install flask` or `python -m pip install flask` or simply `pip install <package name>`):

from flask import Flask

app = Flask(\_\_name\_\_)

@app.get(“/”)

def home():

return “Welcome to Home! Flask is working”

app.run()

When running this script, as we already know, we have to use `flask` as environmental variable on our machine (run from terminal): `flask –app app run`. If it is not on `PATH`, then you can access it from Python: `python -m flask –app app run`.

Now we can decorate this route:

from functools import wraps

def middleware(func):

wraps(func)

def wrapper(\*args, \*\*kws):

result = func(\*args, \*\*kws) # get the string

if isinstance(result, str):

print(“The data has been validated: Data is valid”)

else:

print(“The data is invalid! Please, try again!”)

return result

return wrapper

@app.get(“/”)

@middleware # make sure this is used first before server gets it

def home():

return “Welcome to the Home! Flask is working”

This is how Flask middleware works.

# Conclusion

There are different layers of an application: server and client-sides; database, controllers, API design, and security shell, as well as Redux to keep track of all states in one place as the application keeps growing.

For API, there are several options, but REST API is the most straightforward approach, because it allows the developers to interact with SQL, Prisma, or MongoDB database efficiently by running Server Actions on specific routes. For example, in NEXT.js, you can have `middleware.ts` and `/api/users/route.ts` where you define all functions in one place and then import them when you need to use them in your application.

gRCP is another API design and it provides you with public and private procedures for `server functions` to create and manage databases. For example,

import { z } from "zod";

import {

createTRPCRouter,

publicProcedure,

protectedProcedure,

} from "~/server/api/trpc";

export const tweetRouter = createTRPCRouter({

infiniteFeed: publicProcedure

.input(z.object({

limit: z.number().optional(),

cursor: z.object({

id: z.string(),

createdAt: z.date().optional()

})

}))

.query(async ({input: {limit = 10, cursor}, ctx}) => {

const currentUserId = ctx.session?.user.id;

const data = await ctx.prisma.tweet.findMany({

take: limit + 1,

cursor: cursor

? {cursor: id} as const

: undefined,

orderBy: [

{

createdAt: "desc"

},

{

id: "desc"

}

],

select: {

id: true,

content: true,

createdAt: true,

\_count: { select: { likes: true } },

likes:

currentUserId == null

? false

: { where: {userId: currentUserId}},

user: {

select: {

name: true,

id: true,

image: true

}

}

}

});

let nextCursor: typeof cursor | undefined;

if(data.length > limit){

const nextItem = data.pop();

if(nextItem){

nextCursor = {

id: nextItem.id,

createdAt: nextItem.createdAt

};

}

}

return {

tweets: data.map((tweet) => ({

id: tweet.id,

content: tweet.content,

createdAt: tweet.createdAt,

likeCount: tweet.\_count.likes,

user: tweet.user,

likedByMe: tweet.likes?.length > 0,

})),

nextCursor

};

}),

create: protectedProcedure

.input(z.object({ content: z.string() }))

.mutation(async ({input : {content}, ctx}) => {

const tweet =

await ctx.prisma.tweet.create({ data: { content, userId: ctx.session.user.id}});

return tweet;

}),

});

As you can see, this API design introduces a layer of security, not just middleware.

# References

1. Sergei Ivanov, Computer Hardware, 2023, Narva Tartu College
2. Vercel Inc., NEXT.js, 2023
3. MongoDB Inc., MongoDB, 2023
4. Oracle company, MySQL, Oracle, 2023