AppNation Initial Case

openWeather API design

Content

Important Notes

I know that you are not going to read everything carefully, but the text in here can be finished in 5 to 10 minutes, and written in easy to read language, so I would be grateful if you read all them.

Since I need to prepare my final exams too, the case only meet the basic demands, such as:

- Authorization with JWT tokens, and hashed passwords,
- Admin and user roles,
- PostgreSQL and Prisma,
- OpenWeather API fetching
- Caching through database

Hence, the case is not complete in terms of proper caching, which I implemented as a basic query in the database, not with Redis. Honestly, this is the first time I've heard about it, and I learned that it is an in-memory data structure store that allows fast key access over TCP. Since I had limited time and did not know the technology, I decided not to use it and only implemented the basic query on the database before making the OpenWeather API request. But if you give me a chance, I am sure that I can learn it properly and integrate it into the case or something else.

Other than this, I did not focus on design. By design, I mean using the information of incoming data (which information should be used, where this information should be used, etc.). I believe it is more the job of a product team rather than a software engineer. Hence, I only used the basic information from the incoming weather response. Additional design choices, obviously, can easily be integrated.

I decided to use Node.js for this case. I chose it because I did not have any prior experience with Node.js, and, you know, just two days before your email, I was curious about it and started to learn. So I decided to take this opportunity to improve my knowledge of Node.js. I already had a background in mobile application development with Java and Spring, so setting up the project was not a hard task.

Since this is a case that aims to test our knowledge, capabilities, and understanding of the topic, I will explain everything I did without thinking too much whether this is too basic to tell or not.

Initial Set Up

First thing that I do was setting the Node.js project. To do that, first we create an empty folder, then using console:

```
npm init -y
```

This will create the **“package.json”**, where we can see the overview of our Node.js server such as name and dependencies.

After the initialization, we need to download our dependencies that we will use in the application. For this case, I use axios, bcryptjs, dotenv, express, jsonwebtoken and pg. Using the console:

```
npm install axios bcryptjs dotenv express jsonwebtoken pg
```

- **AXIOS:** A promise-based HTTP client that lets your code easily send GET/POST/PUT/DELETE requests to external APIs and handle their responses.
- **BCRYPTJS:** A JavaScript implementation of the bcrypt hashing algorithm, used to securely hash and compare passwords, allowing you to store passwords safely in the database and ensure confidentiality.
- **DOTENV:** Loads environment variables from a .env file into process.env. It’s typically used to store sensitive configuration such as API keys, database URLs, or static project settings.
- **EXPRESS:** A web-application framework for Node.js that simplifies routing, middleware, and request/response handling, making it easy to build REST APIs quickly.
- **JSONWEBTOKEN:** Used for stateless user authentication. You can also embed custom data like user roles inside the token, allowing role-based access control upon verification.
- **PG:** The official PostgreSQL client for Node.js, used to connect to a PostgreSQL database, run SQL queries, and manage database operations.

Lastly we need to set up Prisma ORM (Object-Relational Mapping) tool. It translates JavaScript/TypeScript function calls into SQL queries. Makes life easier to deal with database queries. To set up Prisma, using the console:

```
npm install prisma –save-dev

npm install @prisma/client

npm prisma init
```

This creates **prisma/schema.prisma**. Then we need our connection URL of the database:

```
DATABASE_URL="postgresql://user:password@localhost:5432/dbname?
schema=public"
```

Then we can create our models that we will use. I used two models in this case, one of them is “User”, the other one is “WeatherQuery”. They are defined in the prisma/schema.prisma like this:

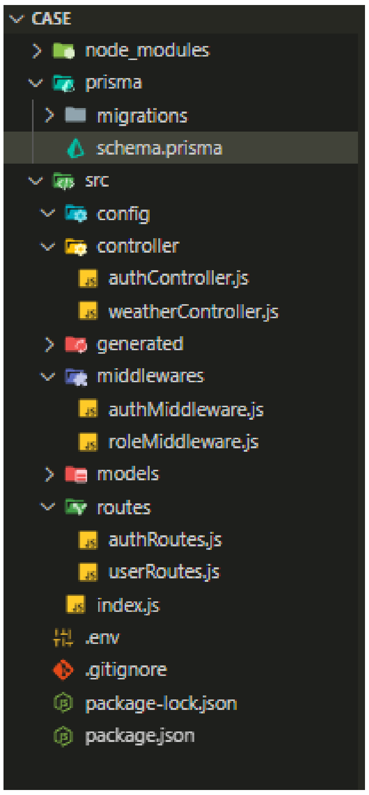
```
model User {
  id        Int       @id @default(autoincrement())
  username  String    @unique
  password  String
  role      String    @default("user")
  createdAt DateTime  @default(now())
```

```
  weatherQueries WeatherQuery[]
}

model WeatherQuery {
  id        Int         @id @default(autoincrement())
  city      String
  forecast  Json
  createdAt DateTime    @default(now())

  userId    Int
  user      User        @relation(fields: [userId], references: [id])
}
```

The last command for us the set up everything is this:

```
npx prisma generate
npx prisma db push
```

Folder Structure



```
CASE
  node_modules
  prisma
    migrations
    schema.prisma
  src
    config
    controller
      authController.js
      weatherController.js
    generated
    middlewares
      authMiddleware.js
      roleMiddleware.js
    models
    routes
      authRoutes.js
      userRoutes.js
    index.js
  .env
  .gitignore
  package-lock.json
  package.json
```

This folder structure is conventional and what it has been teached to me when I was taking the mobile application development course. The only different is that I use Controller to write implementation rather than creating service folder. This is because the case is small. But for bigger projects, it is better to split the control of the logic and the logic.

Files and Codes

- **Index.js**

```
const express = require("express");
const dotenv = require("dotenv").config();
const authRoutes = require("./routes/authRoutes");
const userRoutes = require("./routes/userRoutes");
const app = express();

//Middleware
app.use(express.json());

//Routes
app.use("/weatherApp/auth", authRoutes);
app.use("/weatherApp", userRoutes);
```

```

//Start the server
const PORT = process.env.PORT || 7002
app.listen(PORT, () => {
    console.log(`Server is running at port ${PORT}`);
})
```

This is our applications start point. The server is listening on PORT 7001, defined in .env file. For middleware we use json to handle json files easily. And we use express module to handle routes.

- **userRoutes**

```
const express = require("express");
const verifyToken = require("../middlewares/authMiddleware");
const authorizeRoles = require("../middlewares/roleMiddleware");
const {getLocationFromCity, getWeatherReportFromCity, getQueryHistory} = require("../controller/weatherController")
const router = express.Router();

// Only "role:admin" can access this, actually this is only for test.
router.get("/admin", verifyToken, authorizeRoles("admin"), (req, res) => {
    res.json({message: "This is admin page."});
});

router.get("/cityToLocation", verifyToken, getLocationFromCity);
router.get("/getWeatherReportFromCity", verifyToken, getWeatherReportFromCity);
router.get("/getQueryHistory", verifyToken, getQueryHistory);

// Everyone can access this, actually this is only for test.
```

```
router.get("/user", verifyToken, authorizeRoles("admin", "user"), (req, res) => {
    res.json({message: "This is user page."});
});

module.exports = router;
```

In this file, we define our endpoints, or API’s we can say. These are the services that are available to user from our server. Each of the endpoints call functions to do necessary logic such as authentication, getting location from the given city name, getting weather report, or getting the query history. These functions are defined their corresponding layer as we will see.

- **authRoutes.js**

```
const express = require("express");
const {register, login} = require("../controller/authController");
const router = express.Router();

router.post("/register", register);
router.post("/login", login);

module.exports = router;
```

This is the endpoints for user resgister and login. User use these API’s to register and to login. They both call their corresponding function “register” and login”.

- **roleMiddleware.js**

```
const authorizeRoles = (...allowedRoles) => {
    return (req, res, next) => {
        if (!allowedRoles.includes(req.user.role)){
            return res.status(403).json({message: "Access denied."})
        }
        next();
    }
};
```

```

module.exports = authorizeRoles;
```

In this file, we check wheter the user has the allowed roles or not, given in the parameter of the function "…allowedRoles". If it is alolwed, then with next() function it lets the caller function continue.

- **authMiddleware.js**

```
const jwt = require("jsonwebtoken");

const verifyToken = (req, res, next) => {
    let token;
    let authHeader = req.headers.authorization;
    if (!authHeader || !authHeader.startsWith("Bearer")) {
        return res.status(401).json({message: "No header,
authorization denied."});
    }
    token = authHeader.split(" ")[1];

    if (!token)
        return res.status(401).json({message: "No token,
authorization denied."});

    try {
        const decode = jwt.verify(token,
process.env.JWT_SECRET);
        req.user = decode;
        console.log("The decoded user is: ", req.user);
        next();

    } catch (error) {
        return res.status(401).json({message: "Token is not
valid, authorization denied."});
    }
}
```

```
•
• module.exports = verifyToken;
```

authMiddleware.js is used to verify token. It uses the module “jsonwebtoken” that I explained before. In case of error, it gives approtiate messages and status codes.

- authController.js

It has two functions, register and login. They are used in authRoutes. Register function expects username and password fields in json. The role is either “user” or “admin”. I did not put any check whether the spelling is correct or not. So the function behaviour is undefined for that. The login function expect two field username and password in json body.

```
const register = async(req, res) => {
    const {username, password, role} = req.body;

    try {
        const newUser = await createUser(username, password, role);
        return res.status(201).json({ message: "User registered.",
userId: newUser.id, username: newUser.username});

    } catch (error) {
        if (error.code === "P2002") {
            return res.status(400).json({message: "Username is
already taken."});
        }

        return res.status(500).json({message: "Something went wrong
while registering.", error: error.message});
    }
};
const login = async(req, res) => {
    try {
        const {username, password} = req.body;
        const user = await prisma.user.findUnique({ where: {
username }, });
```

```
        if (!user)
            return res.status(404).json({message: "User not found.",
user: username});

        const isPasswordCorrect = await bcrypt.compare(password,
user.password);
        if (!isPasswordCorrect)
            return res.status(400).json({message: "Invalid
credentails.", user: username});

        const token = jwt.sign({id: user.id, role: user.role},
process.env.JWT_SECRET, {expiresIn: "1h"});
        return res.status(200).json({ token });

    } catch (error){
        return res.status(500).json({message: "Something went wrong
while login process.", error: error.message});
    }

};
```

- **weatherController.js**

I can say this is the main logic. Since it has 240 lines of code, I only explain what it does and what it has inside. Since source code also be shared, you can check it later if you want more detail.

The code is divided to two main part. The first part is consist of helper functions. What they do is straight forward (For example isCityExist(city) check whether a city is exists with that name or not).

The second part is the actual enpoint functions. Our userRoutes.js file uses these functions. These are getLocationFromCity, getWeatherReportFromCity and getQueryHistory.

Note for getWeatherReportFromCity, since I do not implement the Redis, this function does the cache machanism as looking at the database first before doingthe request to the openWeather API.

Note for getQueryHistory, only admins can access other users query history.