Module 03

Purwadhika
Digital Technology School

Full Stack Web Development

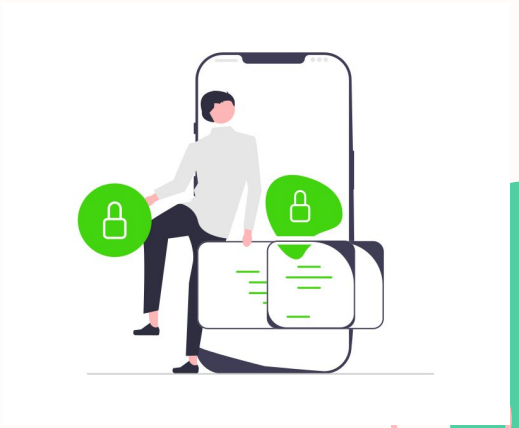
Authentication & Authorization

Job Connector Program

Introduction to Authentication & Authorization

User **Authentication & Authorization** is one of the important part of any web application.

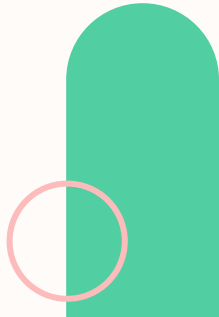
In simple words, **Authentication** is the process of verifying who a user is **(who you are)**, and Authorization is the process of verifying what they have access to **(what you are allowed to do)**.



What is Authentication?

Authentication is a process to verify that someone or something is who they say they are. Technology systems typically use some form of authentication to secure access to an application or its data. For example, when you need to access an online site or service, you usually have to enter your username and password.

Behind the scenes, it compares the username and password you entered with a record it has on its database. If the information you submitted matches, the system assumes you are a valid user and grants you an access. System authentication in this example presumes that only you would know the correct username and password.

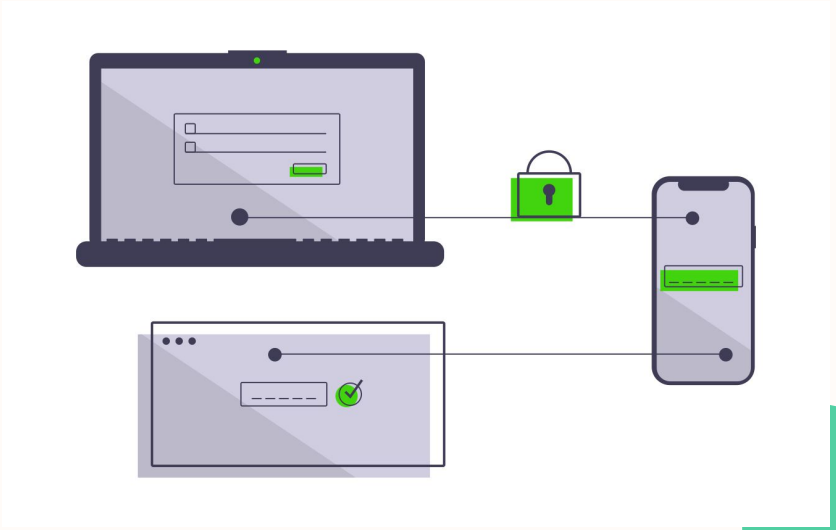


Why We need Authentication?

The purpose of authentication is to verify that someone or something is who or what they claim to be. Typically, authentication protects items of value, and in the information age, it protects systems and data.

Common Types of Authentication

Systems can use several mechanisms to authenticate a user. Typically, to verify your identity, authentication processes use something you know, something you have or something you are.



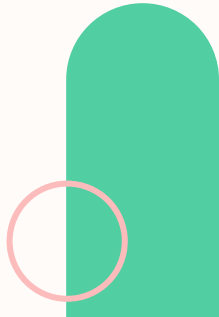
Common Types of Authentication

- **Passwords and security questions** are two authentication factors that fall under the something-you-know category. As only you would know your password or the answer to a particular set of security questions, systems use this assumption to grant you access.
- Another common type of authentication factor uses something you have. **Physical devices such as USB security tokens and mobile phones** fall under this category. For example, when you access a system, and it sends you a One Time Pin (OTP) via SMS or an app, it can verify your identity because it is your device.
- The last type of authentication factor uses something you are. **Biometric authentication mechanisms** fall under this category. Since individual physical characteristics such as fingerprints are unique, verifying individuals by using these factors is a secure authentication mechanism.

What is Authorization?

Authorization is the security process that determines a user or service's level of access. In technology, we use authorization to give users or services permission to access some data or perform a particular action.

For example, in a coffee shop business, Rahul and Lucia have different roles. As Rahul is a barista, he may only place and view orders. Lucia, on the other hand, in her role as manager, may also have access to the daily sales totals. Since Rahul and Lucia have different jobs in the coffee shop, the system would use their verified identity to provide each user with individual permissions. It is vital to note the difference here between authentication and authorization. **Authentication verifies the user** (Lucia) before allowing them access, and **authorization determines what they can do once the system has granted them access** (view sales information).

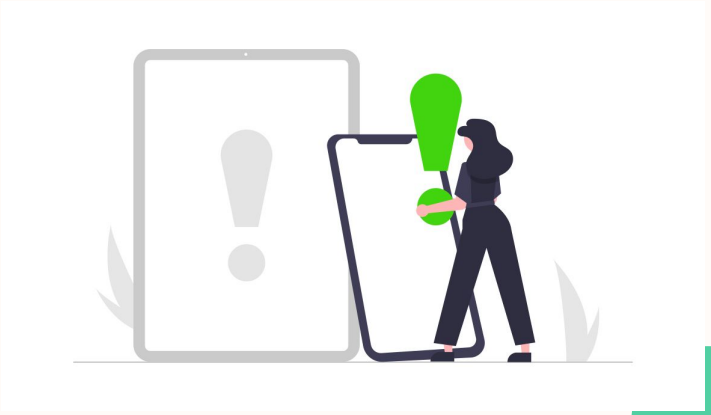


Common Types of Authorization

- Authorization systems exist in many forms in a typical technology environment. For example, Access Control Lists (ACLs) determine which users or services can access a particular digital environment. They accomplish this access control by enforcing allow or deny rules based on the user's authorization level. For instance, on any system, there are usually general users and super users or administrators. If a standard user wants to make changes that affect its security, an ACL may deny access. On the other hand, administrators have the authorization to make security changes, so the ACL will allow them to do so.
- In any enterprise environment, you typically have data with different levels of sensitivity. For example, you may have public data that you find on the company's website, internal data that is only accessible to employees, and confidential data that only a handful of individuals can access. In this example, authorization determines which users can access the various information types.

The Difference Between Authentication and Authorization

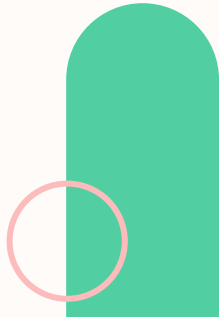
authentication and authorization may sound alike, but each plays a different role in securing systems and data. Unfortunately, people often use both terms interchangeably as they both refer to system access. **However, they are distinct processes. Simply put, one verifies the identity of a user or service before granting them access, while the other determines what they can do once they have access.**



The Difference Between Authentication and Authorization

The best way to illustrate the differences between the two terms is with a simple example:

Let's say you decide to go and visit a friend's home. On arrival, you knock on the door, and your friend opens it. She recognizes you (**authentication**) and greets you. As your friend has authenticated you, she is now comfortable letting you into her home. However, based on your relationship, there are certain things you can do and others you cannot (**authorization**). For example, you may enter the kitchen area, but you cannot go into her private office. In other words, you have the authorization to enter the kitchen, but access to her private office is prohibited.

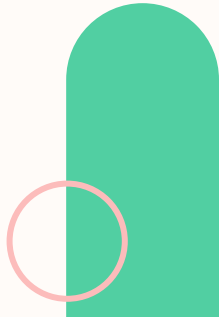


Authentication & Authorization

After understanding the concept of authentication and authorization, lets implement that into your projects.

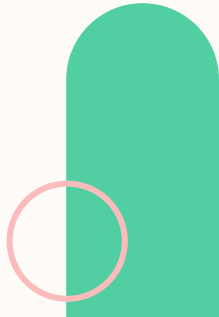
In this part, we assume you already know about:

- NodeJS and ExpressJS
- SQL with MySQL as database
- ORM with Prisma
- Postman to perform api testing



Authentication & Authorization

- First initialize your projects and setup your db connection on config/config.json adjust your username, password, database, and host based on your db connection
- After initialize your projects and setup your db connection, create two files :
  - auth.controller.ts
  - auth.router.ts



Create Auth Controller

```
import { PrismaClient } from "@prisma/client";

type User = {
  email: string;
  name: string;
  password: string;
};
const prisma = new PrismaClient();

export const register = async (user: User) => {
  try {
    // register
  } catch (error) {
    throw error;
  }
};

export const login = async (email: string, password: string) => {
  try {
    // login
  } catch (error) {
    throw error;
  }
};
```

Create Auth Router

```
import { Router, Request, Response } from "express";
import { register, login } from "./auth.controller";

const authRouter = Router();

authRouter.post("/register", async (req: Request, res: Response) => {
  await register();
});

authRouter.post("/login", async (req: Request, res: Response) => {
  await login();
});

export { authRouter };
```

Register

```
export const register = async (user: User) => {
  try {
    const existingUser = await prisma.user.findUnique({
      where: { email: user.email },
    });

    if (existingUser) {
      throw new Error("email has been used");
    }

    const newUser = await prisma.user.create({ data: user });

    return newUser;
  } catch (error) {
    throw error;
  }
};
```

In this code you will create register action. Where you would check email availability. If it's available then create register process would be successful.

Try to create user with post method:

http://localhost:3000/auth/register

Put username, email, and password through body.

Login

```
export const login = async (email: string, password: string) => {
  try {
    const user = await prisma.user.findUnique({
      where: { email, password },
    });

    if (!user) {
      throw new Error("invalid email or password");
    }

    return user;
  } catch (error) {
    throw error;
  }
};
```

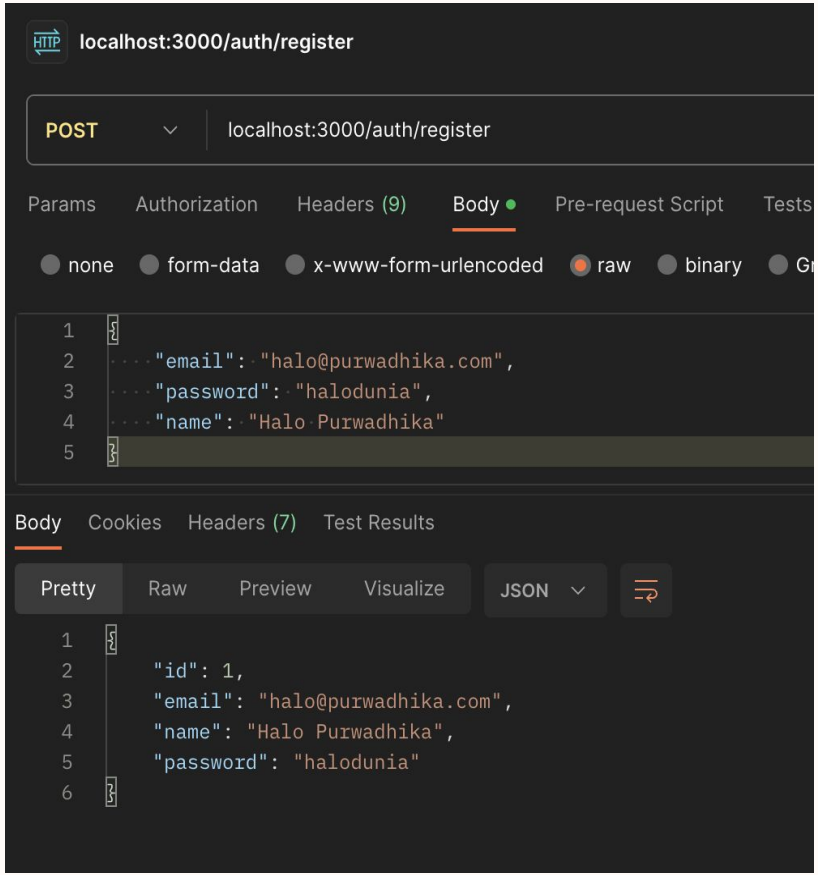
Still in the same file, create another method for login after the register.

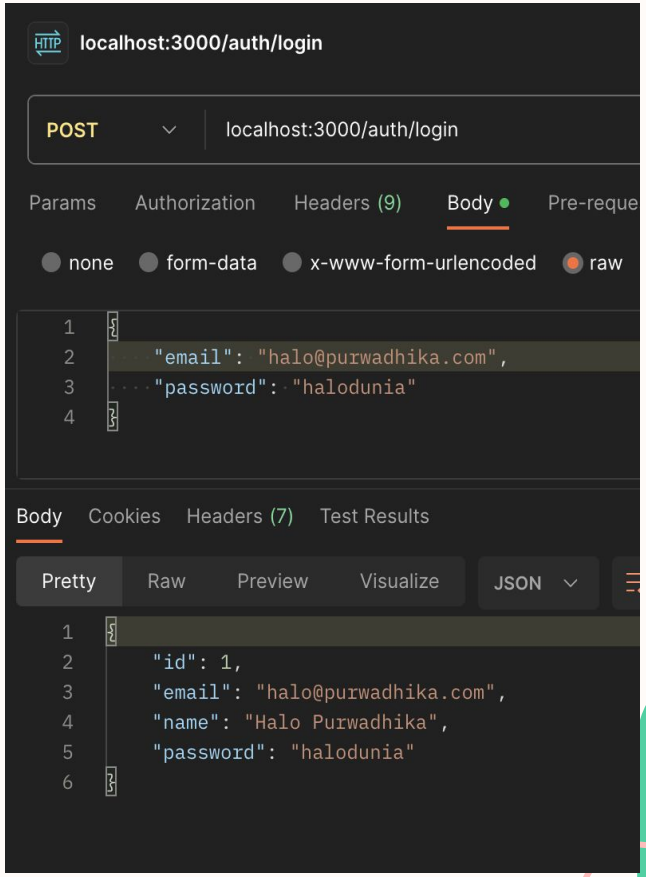
Based on authentication rules, only the one who login that knows email and password for him/her self. If its not, authentication process will send an error.

In this example, login process was using email and password as a terms of login. If email and password are same as on database, it would show login success message.

Try to login through post method:
http://localhost:3000/auth/login

Put email and password through body


localhost:3000/auth/register
POST
localhost:3000/auth/register
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
none
form-data
x-www-form-urlencoded
raw
binary
{
"email": "halo@purwadhika.com",
"password": "halodunia",
"name": "Halo Purwadhika"
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": 1,
"email": "halo@purwadhika.com",
"name": "Halo Purwadhika",
"password": "halodunia"
}


localhost:3000/auth/login
POST
localhost:3000/auth/login
Params
Authorization
Headers (9)
Body
Pre-reque
none
form-data
x-www-form-urlencoded
raw
{
"email": "halo@purwadhika.com",
"password": "halodunia"
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": 1,
"email": "halo@purwadhika.com",
"name": "Halo Purwadhika",
"password": "halodunia"
}

Password Encryption

Let's check your users table on the db. You can see an user password since it was written clearly. Lets secure his password using bcrypt. This package would be useful to encrypt and decrypt any information through your projects.

Read more for documentation:

https://www.npmjs.com/package/bcrypt

Password Encryption - Register

```
export const register = async (user: User) => {
  try {
    const { email, password, name } = user;
    const existingUser = await prisma.user.findUnique({
      where: { email },
    });

    if (existingUser) {
      throw new Error("email has been used");
    }

    const salt = await genSalt(10);
    const hashedPassword = await hash(password, salt);
    const newUser = await prisma.user.create({
      data: {
        email,
        password: hashedPassword,
        name,
      },
    });

    return newUser;
  } catch (error) {
    throw error;
  }
};
```

Let’s modify our auth controller and implement bcrypt while encrypting user password in register process.

Don't forget to import bcrypt packages at the top of your code. After that try to register with another account. Check out the password in users table it wont plain ever again.

Password Encryption - Login

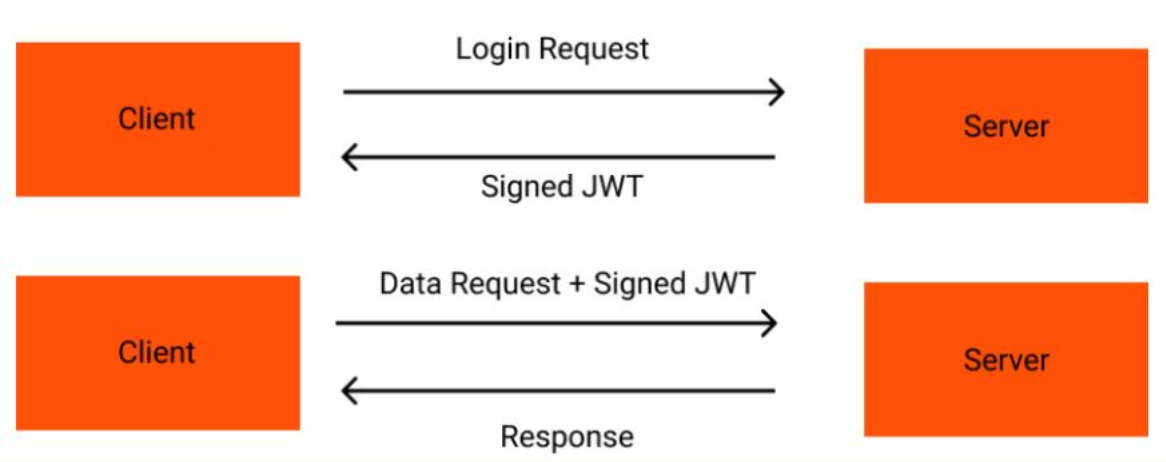
```
export const login = async (email: string, password: string) => {
  try {
    const user = await prisma.user.findUnique({
      where: { email },
    });

    if (!user) {
      throw new Error("invalid email or password");
    }

    const isValidPassword = await compare(password, user.password);

    if (!isValidPassword) {
      throw new Error("invalid email or password");
    }

    return user;
  } catch (error) {
    throw error;
  }
};
```

Next, we need to modify login. In this code you will compare between plain password with the encrypted password in users table using bcrypt.compare method. Try to login with the new register user and check out the response.

You just secure your password data.

JSON Web Token

JSON Web Token (JWT) are a good way of securely transmitting information between parties because they can be signed (Information Exchange). Even though we can use JWT with any type of communication method, today JWT is very popular for handling authentication and authorization via HTTP.

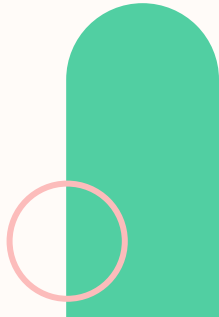


Authorization with JWT

In this case, we would like to create user access level in our projects. Which mean, only permitted user role could access some of information in our server. Let's say we separate our user role into two:

- Basic-user
- Admin-user.

Basic user is not allowed to accessing list of all users from database, while admin user is granted to access all users information.



Update Schema

Let's take a look at user schema. Add a new “role” column and we will use it to validate the user's role.

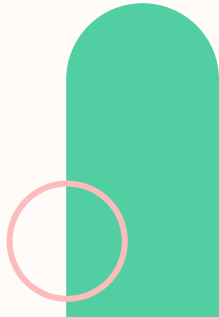
```
model User {
  id       Int    @id @default(autoincrement())
  email    String @unique
  name     String
  password String
  role     String @default("user")
}
```

Authorization with JWT

Before we re going to next step, lets install JSON Web Token package into our projects.

Find out more about the documentation:

https://www.npmjs.com/package/jsonwebtoken



Authorization with JWT

```
export const login = async (email: string, password: string) => {
  try {
    const user = await prisma.user.findUnique({
      where: { email },
    });

    if (!user) {
      throw new Error("invalid email or password");
    }

    const isValidPassword = await compare(password, user.password);

    if (!isValidPassword) {
      throw new Error("invalid email or password");
    }

    const jwtPayload = { email, role: user.role };
    const token = sign(jwtPayload, "yourSecretKey", { expiresIn: "1h" });

    return { user, token };
  } catch (error) {
    throw error;
  }
};
```

Go to auth controller and focus on login process.

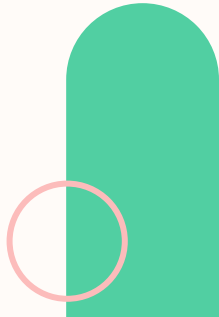
Import JSON Web Token package into this file. After that use method jwt.sign to generate token.

First parameter would be a data to convert, the second one is secret key to handle your token, and the last params is optional, but in this case we could set expires token time.

Authorization with JWT

In **sign** method we put expires time. This would be useful to set session time for user access. For example mobile banking apps implement session login time. Token was created when user login and has been set expires time. When token expired, user login session would be no longer available and need to create a new one and suddenly you were log out from the apps.

```
const jwtPayload = { email, role: user.role };
const token = sign(jwtPayload, "yourSecretKey", { expiresIn: "1h" });
```



Authorization with JWT

Try to login, if its success would show response like image shown below.


```
{
    "user": {
        "id": 2,
        "email": "halo@purwadhika.com",
        "name": "halodunia",
        "password": "$2b$10$Koj3mE23wPsjKatciWul7uMjw7jkiMGx12vbfd4NnxTbThhz1wXgC",
        "role": "user"
    },
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJlbWFpbCI6ImhhbG9AcHVyd2FkaGlrYS5jb20iLCJyb2xlIjoidXNlciIsImlhdCI6MTcwMzIwNjYyMiwiZXhwIjoxNzAzMjEwMjIyfQ.
        xlzmUBeIvn1nhcZFo9fcbAXvq_kNFBzPmxvdkLhodgs"
}
```

Authorization with JWT

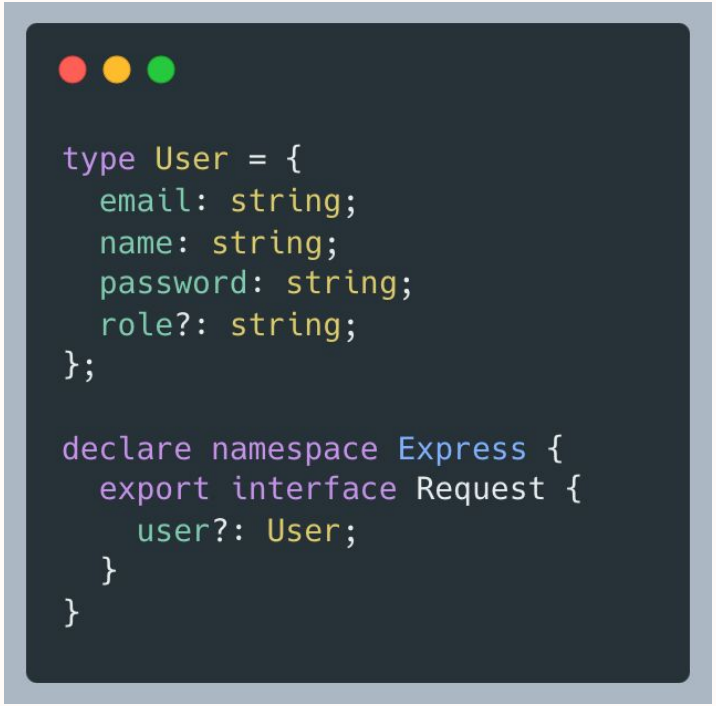
```ts
type User = {
  email: string;
  name: string;
  password: string;
  role?: string;
};

export const verifyToken = async (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  try {
    const token = req.header("Authorization")?.replace("Bearer ", "");

    if (!token) {
      return res.status(401).send("Unauthorized");
    }

    const verifiedUser = verify(token, "yourSecretKey");
    if (!verifiedUser) {
      return res.status(401).send("Unauthorized");
    }

    req.user = verifiedUser as User;

    next();
  } catch (error) {
    console.log(error);
    res.status(500).send({
      message: "Register Error",
      error: (error as Error).message,
    });
  }
};
```

Create file named as auth.middleware.ts.

We would check request authorization by token session since user has been login.

JWT verify method would check validity of your token session.

Extend Express Request Interface



```ts
type User = {
  email: string;
  name: string;
  password: string;
  role?: string;
};

declare namespace Express {
  export interface Request {
    user?: User;
  }
}
```

Request is part of the Express types and is used in Express functions, so you do not have control over it. Fortunately, TypeScript supports declaration merging.

In other words, declaration merging allows you to add additional properties and methods to an existing type, which is exactly what you want to achieve here.

Create new file with name “custom.d.ts”

Authorization with JWT

```
export const adminGuard = async (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  try {
    if (req.user?.role !== "admin") {
      return res.status(401).send("Unauthorized");
    }

    next();
  } catch (error) {
    console.log(error);
    res.status(500).send({
      message: "Register Error",
      error: (error as Error).message,
    });
  }
};
```

After checking user session time, we need to check user role. This mean only admin user could access some information. If request made by admin, then it would given a valid response.

Authorization with JWT

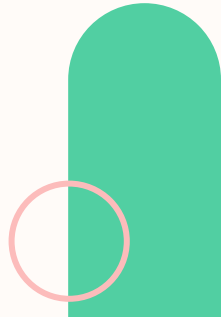
```
userRouter.get(
  "/users",
  verifyToken,
  adminGuard,
  async (req: Request, res: Response) => {
    res.send([]);
  }
);
```

Create new routes in order to retrieve all of users data from server. Put middleware that we just created before into this route. It would prevent basic-user or non sessions user from accessing this routes.

Authorization with JWT

Test your project on postman using this flow:

1. Register new user and add isAdmin in the body and set true as a value
2. Login with the new user that you just created before.
3. Copy the token strings from response.
4. Go to http://localhost:3000/users and go to tab authorization and select Bearer Token as a type and paste the token from response into token field, check image on the next slide as example.
5. Check out the response, and try with non admin user to check the difference



Thank You!