Lecture-6 Auth Functions & Authorization

Agenda

- Auth functions
- Authorization, protecting passwords

Auth Methods & route

Signup

- **Purpose:** Create a new user account.
- **Inputs:** Username, email, password.
- **Steps:**
    1. Validate input data (e.g., ensure username and email are unique, password meets complexity requirements).
    2. Store user information in a database, including username, password, email, and any other relevant details.
    3. Generate an authentication token (e.g., JWT) and send it to the client for subsequent requests.

```
async function signupHandler(req, res){
    try{
        const userObject = req.body;
        // 1. user -> data get, check email, password
        if(!userObject.email || !userObject.password){
            return res.status(400).json({
                message:"required data missing",
                status : "failure"
            })
        }
        // 2. email se check -> if exist ->already loggedIn
        const user = await UserModel.findOne({email: userObject.email});
        if(user){
            return res.status(400).json({
                "message":"user is already logged in",
                status: "failure"
            })
        }
        const newUser = await UserModel.create(userObject);
```

```
        // send a response
        res.status(201).json({
            "message":"user signup successfully",
            user: newUser,
            status:"failure"
        })
    }catch(err){
        console.log("err",err);
        res.status(500).json({
            message: err.message,
            status:"failure"
        })
    }
}

app.post("/signup", signupHandler);
```

Login

- **Purpose:** Authenticate an existing user.
- **Inputs:** Username or email, password.
- **Steps:**
    1. Validate input data.
    2. Retrieve the user's information from the database based on the provided username or email.
    3. If the passwords match, generate an authentication token and send it to the client.

```
JWT_SECRET_KEY = "abrakadabra"
```

```
const jwt = require("jsonwebtoken");
const util = require("util");
const promisify = util.promisify;
const promisifiedJWTsign = promisify(jwt.sign);
async function loginHandler(req,res){
    try{
        const {email, password} = req.body;
        const user = await UserModel.findOne({email});
        if(!user){
            return res.status(404).json({
```

```
                message:"Invalid email or password",
                status : "failure"
            })
        }
        const areEqual = password == user.password;
        if(!areEqual){
            return res.status(400).json({
                message:"Invalid email or password",
                status:"failure"
            })
        }
        // generate token
        const authToken = await
promisifiedJWTsign({id:user["_id"]},process.env.JWT_SECRET_KEY)
        // token -> cookies
        res.cookie("jwt", authToken, {
            maxAge: 1000 * 60 * 60 *24,
            httpOnly:true, // it can only be accessed by the server
        })
        // res send
        res.status(200).json({
            message:"login successfully",
            status:"success",
            user: user
        })
    }catch(err){
        console.log("err",err);
        res.status(500).json({
            message:err.message,
            status:"failure"
        })
    }
}

app.post("/login", loginHandler);
```

Protect-route

- **Purpose:** Restrict access to certain routes based on authentication.
- **Implementation:**
    1. Use a middleware function to check if an authentication token is present in the request headers.

2. If a token is present, verify its validity and extract the user's information from it.
3. If the token is valid and the user is authenticated, allow access to the protected route. Otherwise, return an error response.

```
const cookieParser = require("cookie-parser");
app.use(cookieParser());

const promisifiedJWTverify = promisify(jwt.verify);

async function protectRouteMiddleware(req, res, next){
    try{
        const token = req.cookies.jwt
        if(!token){
            return res.status(401).json({
                message:"unauthorized access",
                status:"failure"
            })
        }
        const decryptedToken = await promisifiedJWTverify(token,
process.env.JWT_SECRET_KEY);
        req.id = decryptedToken.id
        next();
    }catch(err){
        console.log("err",err);
        res.status(500).json({
            message: err.message,
            status:"failure"
        })
    }
}

// app.use(protectRouteMiddleware);
```

Profile

- **Purpose:** Retrieve and display the user's profile information.
- **Implementation:**
    1. Use a protected route to access the profile function.
    2. Extract the user's information from the authentication token.
    3. Retrieve the user's profile data from the database.

4. Return the profile data as a response.

```
async function profileHandler(req,res){
    try{
        const userId = req.id;
        const user = await UserModel.findById(userId);
        if(!user){
            return res.status(404).json({
                message:"user not found",
                status:"failure"
            })
        }
        res.json({
            message:"profile worked",
            status:"success",
            user: user
        })
    }catch(err){
        console.log("err",err);
        res.status(500).json({
            message: err.message,
            status:"failure"
        })
    }
}

app.get("/profile", protectRouteMiddleware, profileHandler);
```

Logout

- **Purpose:** Invalidate an existing authentication token.
- **Implementation:**
    1. Extract the user's information from the authentication token.
    2. Invalidate the token by removing it from a blacklist or marking it as expired in the database.
    3. Redirect the user to a login page or display a success message.

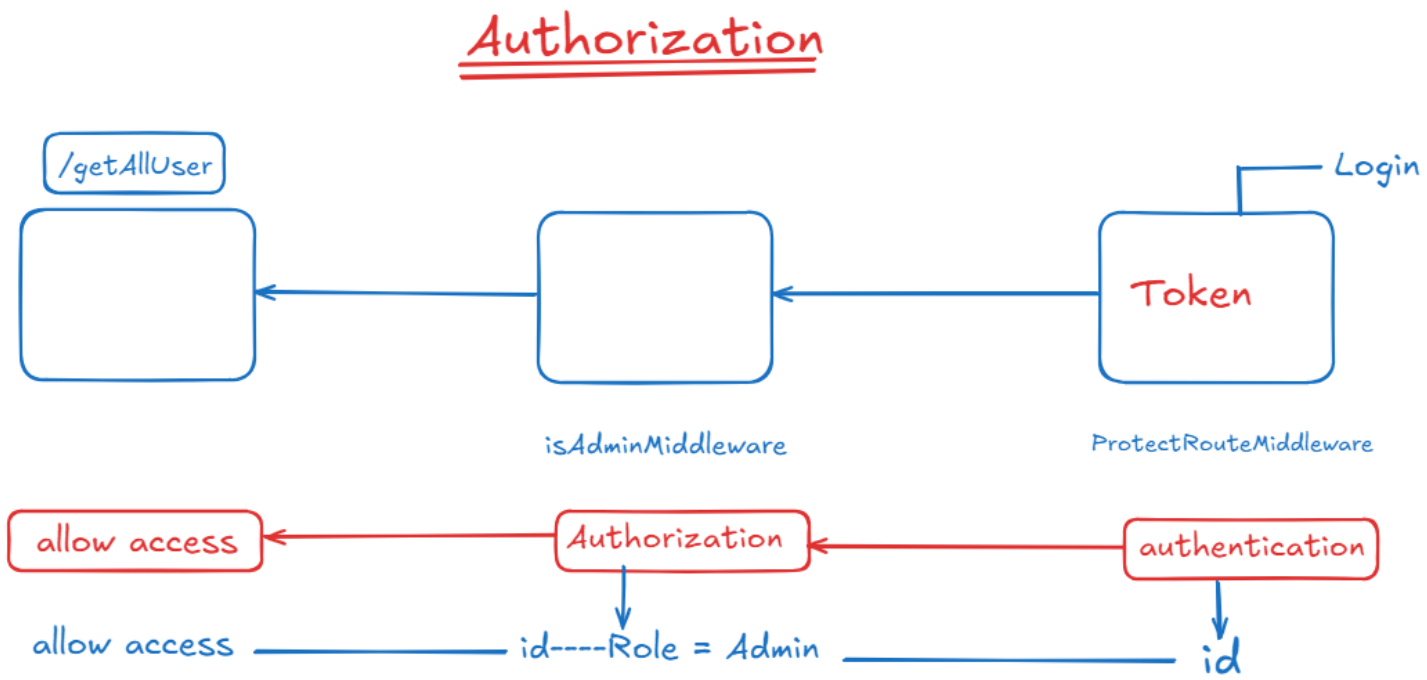
```
async function logoutHandler(req,res){
    try{
        res.clearCookie('jwt',{path: "/"});
        res.json({
```

```
            message:"logout successfully",
            status:"success"
        })
    }catch(err){
        res.status(500).json({
            message:err.message,
            status:"failure"
        })
    }
}

app.get("/logout", logoutHandler);
```

Authorization

Authorization is the process of determining whether a user has the permission to access a resource or perform an action. It's a crucial component of security that ensures only authorized entities can interact with sensitive data or systems.
To implement authorization:

1. **Define roles:** Create roles like "user," "admin," and "moderator."
2. **Assign permissions:** Grant different permissions to each role. For example, a "user" might only be able to view their own data, while an "admin" can view and modify all data.
3. **Use a token-based approach:** Generate a JWT token when a user logs in and include their role information in the token.
4. **Protect routes:** Use middleware to verify the token on each request and check if the user has the necessary permissions to access the requested resource.



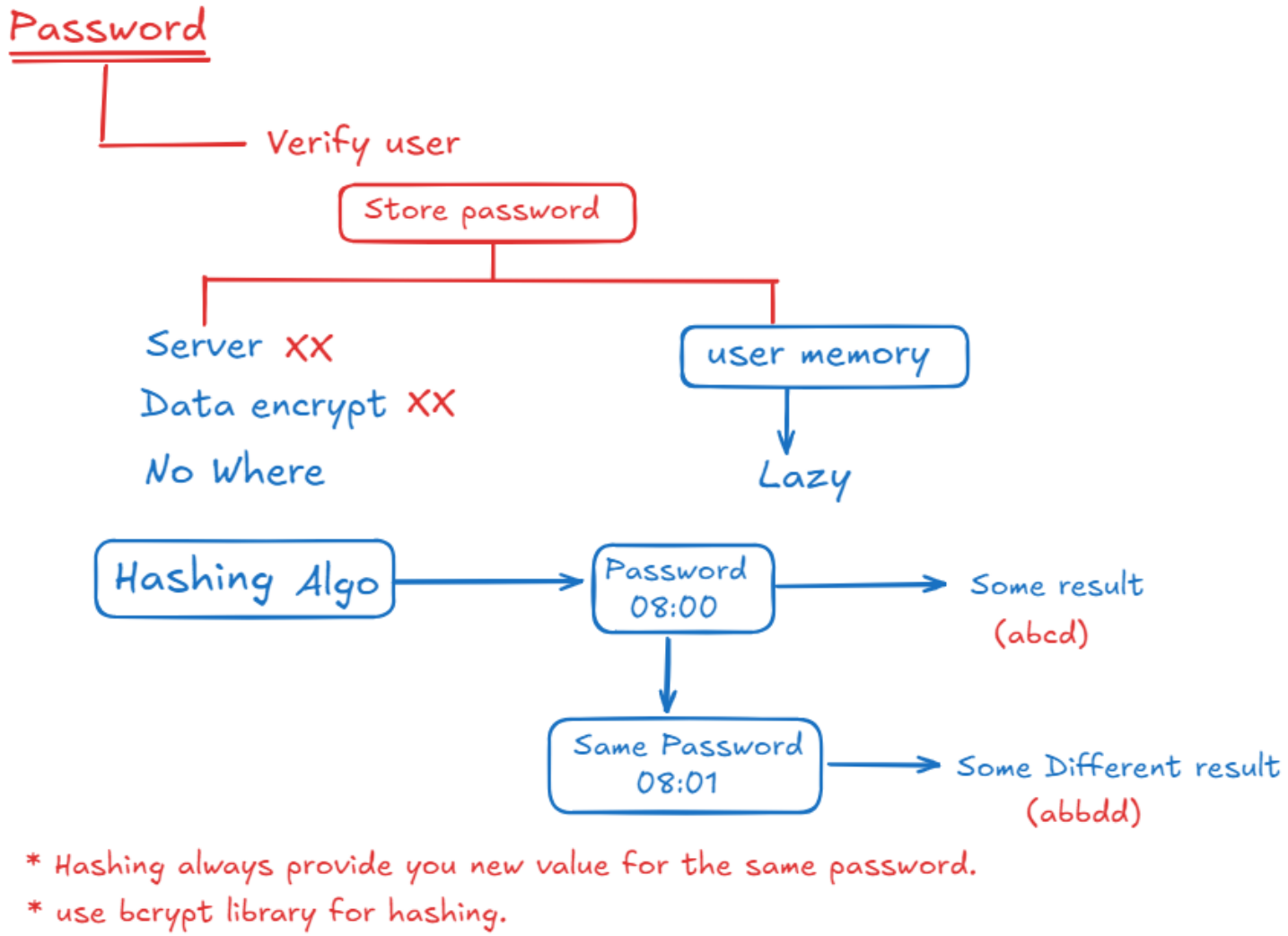

```
async function isAdminMiddleware(req, res, next){
    const id = req.id;
    const user = await UserModel.findById(id);
    if(user.role !== "admin"){
        return res.status(403).json({
            message:"you are not admin",
            status:"failure"
        })
    }else{
        next();
    }
}

app.get("/user",protectRouteMiddleware, isAdminMiddleware, getAllUser);
```

How to store password in DB

The key principles involve hashing and securely storing the passwords rather than storing them in plaintext.

- Hashing: Always hash passwords before storing them. The `bcryptjs` library is a good choice for this.
- Salting: `bcrypt` automatically handles salting, which adds an additional layer of security by ensuring that the same password generates different hashes.

- **Comparing Hashes**: When checking passwords, always use the hash comparison method provided by `bcrypt` to validate.



Install

```
npm install bcryptjs
```

Examle.js

```
const bcrypt = require('bcryptjs');

const password = "abcd";

async function create(){
    console.time();
    const randomSalt = await bcrypt.genSalt(10);
    const hash = await bcrypt.hash(password, randomSalt)
    // const hash = await bcrypt.compare(password, hash);
```

```
    console.timeEnd();
    console.log("hashed password", hash);
}
create();
```

Time Stamp

- Intro (start - 0:12:00)
- Organize Code (0:12:00 - 0:29:00)
- Auth Function (0:29:00 - 1:29:16)
- Authorization (1:29:16 - 1:40:00)
- Protecting password (1:40:00 - end)