Software Project 1

Tutorial 8

Course Staff

❑ Dr. Nada Sharaf	nada.hamed@giu-uni.de A1.211
❑ Eng. Donia Ali	donia.ali@giu-uni.de A1.110
❑ AL. Amany Hussein	amany.hussein@giu-uni.de A1.122
❑ Eng. Hania Ashraf	hania.ashraf@giu-uni.de A1.222
❑ Eng. Omar Ashraf	Berlin omar.ashraf@giu-uni.de 602

Next Js

- Rendering Modes
  - server side
  - client side
- authentication
  - Hashing password
  - JWT
  - cookies
- authorization
  - Role based Authorization
- Implementation
  - Backend
    - CORS
    - Guard
    - Reflector
    - Metadata
  - Frontend

Rendering Modes

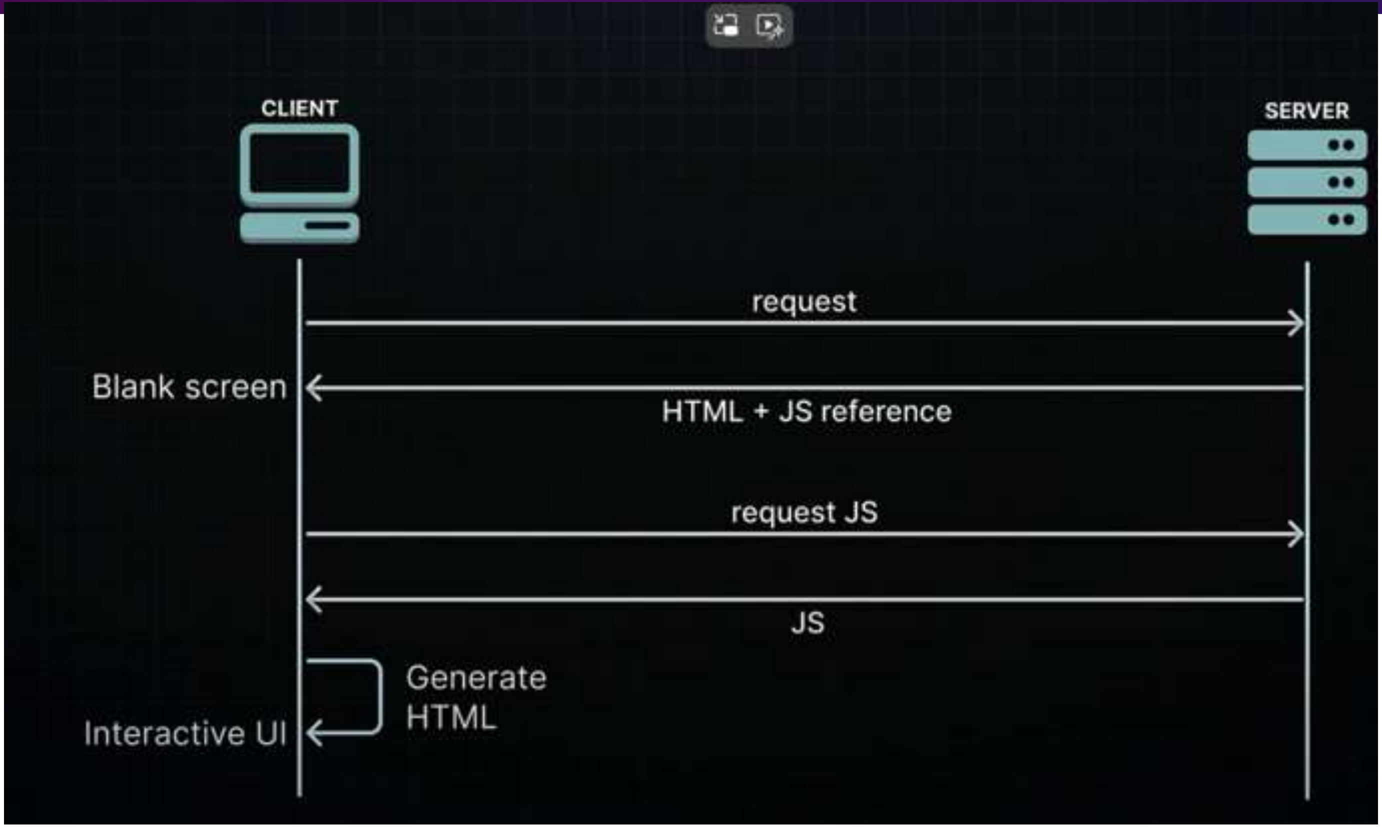
- The **Environments** your application code can be executed in: the **server** and the **client**.
- The **Request-Response** Lifecycle that's initiated when a user visits or interacts with your application.
- The **Network Boundary** that separates server and client code.

CSR

Client-Side Rendering (CSR)

- **Where HTML is Generated**: HTML is generated on the **client-side** (in the browser). When a user visits a page, JavaScript runs in the browser to fetch data and generate the final HTML.
- **How It Works**:
  - Initially, the browser receives a minimal HTML file with JavaScript files. contain root div tag
  - JavaScript then executes in the browser, fetching data from APIs and rendering components to build the complete HTML dynamically by attaching components to root div
- **Advantages**:
  - **Better Interactivity**: Once the initial JavaScript is loaded, interactions on the page are quick because updates happen directly in the browser.
  - **Easier to Scale**: CSR typically places less load on the server, as the server primarily serves static JavaScript files.
- **Disadvantages**:
  - **Slower Initial Load**: Users may experience a delay on the first page load, often referred to as the "white screen" effect, because the browser has to download and execute JavaScript before displaying content. as it handles all work
  - **SEO Limitations**: CSR can be less SEO-friendly because search engines need to execute JavaScript to "see" the content. so generating html that contains single dev is not optimal for SEo as it provides little content for search engine to index

CSR


CLIENT
SERVER
request
Blank screen
HTML + JS reference
request JS
JS
Generate HTML
Interactive UI

SSR

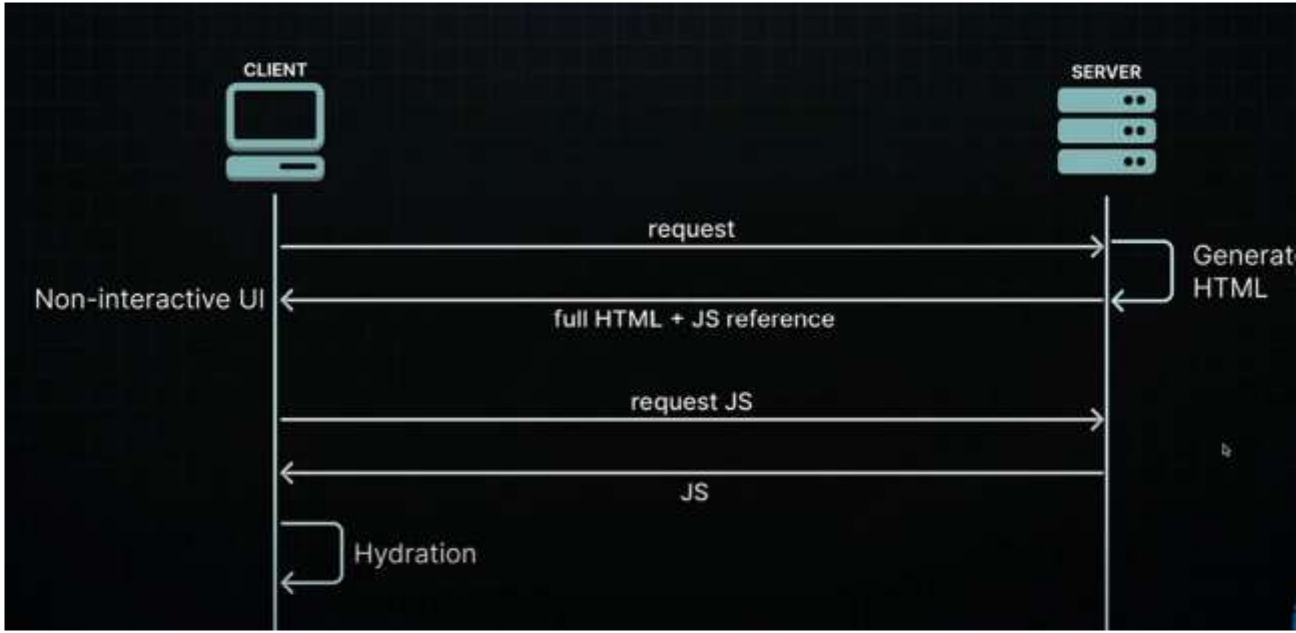
Server-Side Rendering (SSR)

Where HTML is Generated: HTML is generated on the **server-side**. The server assembles the HTML with content before sending it to the browser.

- **How It Works**:
  - When a user requests a page, the server processes the request, fetches necessary data, and sends a fully rendered HTML page to the client. but non interactive UI
  - then browser request for js file to be downloaded
  - When the browser loads the page, it also downloads and executes the associated JavaScript bundle.
    React "hydrates" the HTML:
  - React attaches event listeners and reactivates components' state and behavior on the existing HTML elements.
  - The result is that the pre-rendered HTML now behaves like a fully interactive React app.
- **Advantages**:
  - **Faster Initial Load**: The user can see a complete page as soon as it's loaded, making SSR ideal for content-heavy pages.
  - **SEO-Friendly**: Since the HTML is pre-rendered, search engines can easily crawl and index the content, improving SEO.
- **Disadvantages**:
  - **More Load on the Server**: Each page request requires server processing to render the HTML, which can increase server load, especially with high traffic.
  - **Slower Interactivity**: Although the initial page load is faster, additional interactions may feel slower if they require additional server calls.
  - waiting for database for example, load all data , so server can delay in responding to browser
  - load everything before hydrating anything
  - you have to hydrate everything before interacting with anything
    ALL or nothing problems

SSR

Hydration in the context of web development and frameworks like React and Next.js refers to the process of attaching **client-side JavaScript logic and interactivity** to static HTML that was pre-rendered on the server. It bridges the gap between the **server-rendered content** and the **fully interactive client-side application.**



SSR

Server-Side Rendering (SSR)

problem: ALL or nothing Waterfall

- waiting for database for example, load all data , so server can delay in responding to browser
- load everything before hydrating anything
- you have to hydrate everything before interacting with anything

React introduces **Suspeanse SSR ARCH**.

using <Suspence> to unlock two features:

1. HTML streaming on server

solves first problem like parallel routes as react will stream the missing component

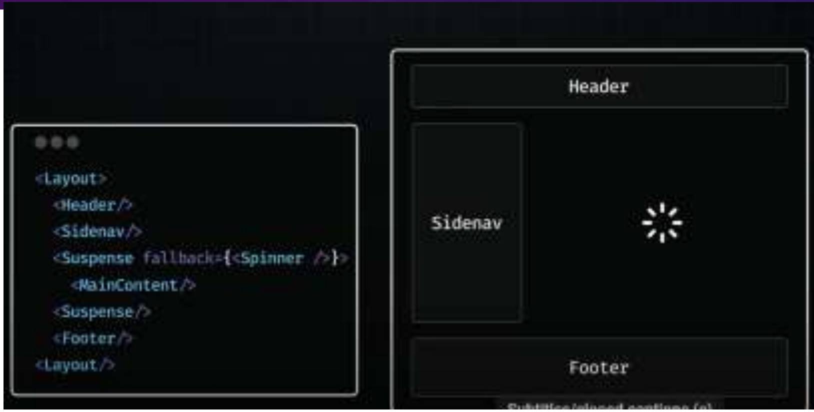
1. selective hydration on client

allows hydration of sections as they become available before rest of elements are loaded so for example main section is hydrated once itis loaded

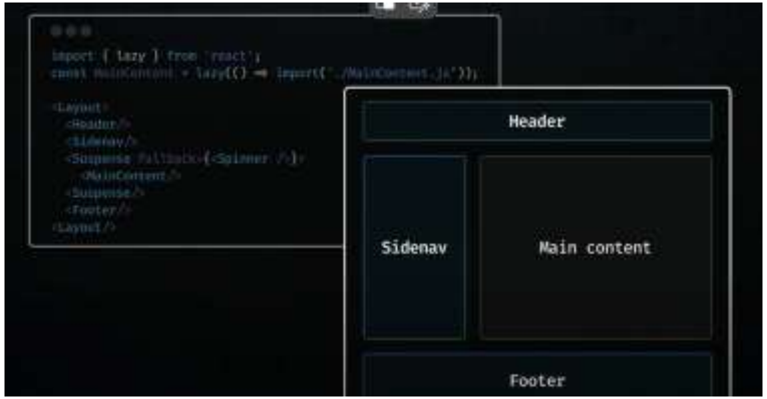
so issues are solved

Drawbacks:

1. entire js must be downloaded to browser , download much data?
2. all component undergo hydration even if it is not interactive?
3. js execution done on user device, so slow down performance if devices are not powerful so , work should done on user device?



```
<Layout>
  <Header/>
  <Sidenav/>
  <Suspense fallback={<Spinner />}>
    <MainContent/>
  <Suspense/>
  <Footer/>
<Layout/>
```



```
import { lazy } from 'react';
const MainContent = lazy(() => import('./MainContent.js'));

<Layout>
  <Header/>
  <Sidenav/>
  <Suspense fallback={<Spinner />}>
    <MainContent/>
  <Suspense/>
  <Footer/>
<Layout/>
```

RSC

CSR-> SSR-> SUSPENSE SSR moving to…

React server components

developed by react team to leverage strength of CSR and SSR

so arch introduces a dual component model:

- ◎ client components
- ◎ server components

so app router in next is built around RSC arch

Components are server ones by default until specifying client by 'use client'

- ◎ Client component:
  - ○ rendered on client side but can be executed once on server for optimization
  - ○ have access to client env as: browsers, state, effects, local storage
- ◎ server component:
  - ○ operated on server and stays on server and never downloaded to client
  - ○ eliminate need to download js files for these components and remove hydration step
  - ○ direct access to server side resources like database
  - ○ improved data fetching

SSG

- **Static Site Generation (SSG)** is a pre-rendering technique where HTML pages are generated at **build time**, rather than at runtime.
- Once the HTML is generated, it is served as static files, which can be cached by CDNs (Content Delivery Networks), resulting in fast page loads.
- `getStaticProps`: A special function in Next.js used to fetch data during build time.

Benefits of SSG

- **Faster Page Loads**: Static pages are served from CDNs, ensuring quick load times.
- **Improved SEO**: Since the content is already rendered, search engines can easily crawl and index the pages.
- **Scalability**: No need for a server to render pages at request time. This reduces server load and improves scalability.
- **Security**: As the pages are static, there's less surface area for attacks compared to dynamic pages.

When to Use SSG?

- Ideal for **content-driven websites** like blogs, product pages, documentation, and portfolios where content doesn't change frequently.
- Great for pages that **don’t need real-time data** or have low-frequency updates.

Authentication vs Authorization

- **Authentication** is the process of verifying a user's identity
- **Authorization** is the process of granting or denying access rights based on the authenticated user's permissions.
- Both authentication and authorization are crucial components of access control in systems to ensure secure and controlled user interactions.

Authentication
USER NAME
******
login
Confirms users are who they say they are.
Authorization
Gives users permission to access a resource.

Authentication

- Authentication is the process of verifying a user's identity. It answers the question: **"Who are you?"**
- Install dependencies : @nestjs/jwt and @nestjs/passport.
- used to Implement login/signup with token issuance.

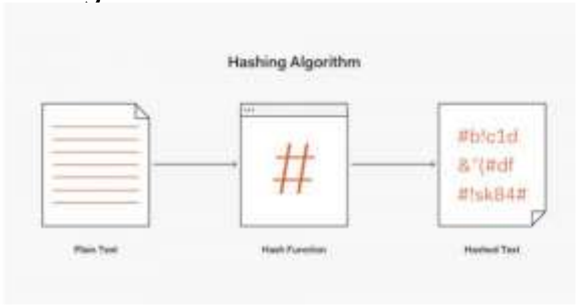
How It Works:

1. **Credentials:** The user provides credentials such as:
   - Username and password
   - Biometric data (e.g., fingerprint, face recognition)
   - Multi-Factor Authentication (MFA)
2. **Validation:** The system checks the credentials against its database.
3. **Outcome:** If credentials are valid, the user is authenticated.

Hashing

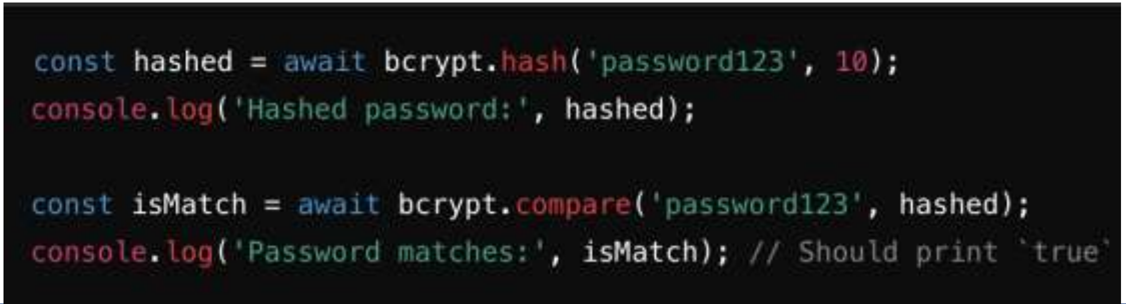
Hashing Password

- Hashing secures user passwords by converting them into fixed-length strings.
  - Hashing function is one way algorithm.
  - However, same passwords then produce same hashes!
  - Solution :
  - We use a randomly generated text… “salt”
    - 1. Add salt
    - 2. Hash with salt



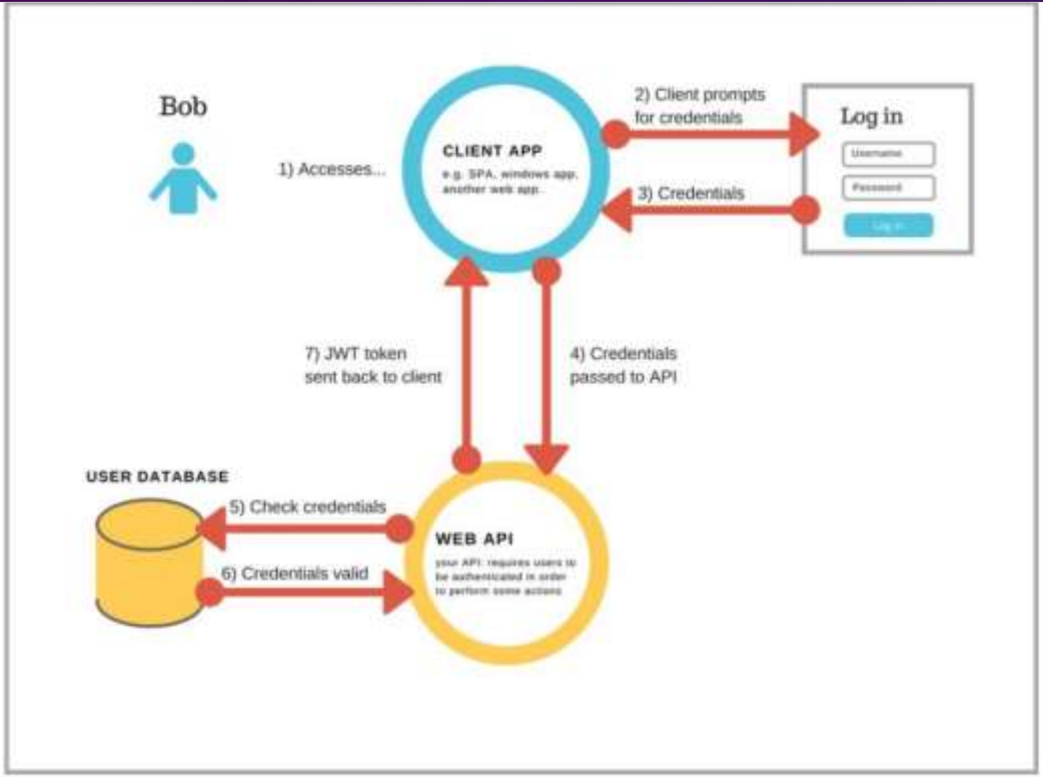
npm install bcrypt
npm install --save-dev @types/bcrypt

- **Use bcrypt in NestJS for hashing**:
  - Install bcrypt.
  - Hash passwords before saving them to the database.
  - Compare hashed passwords during login.
  - However, it would be tedious to re-ask them to login in just before every single secure operation



```
const hashed = await bcrypt.hash('password123', 10);
console.log('Hashed password:', hashed);

const isMatch = await bcrypt.compare('password123', hashed);
console.log('Password matches:', isMatch); // Should print `true`
```

Hashing and tokens


Bob
1) Accesses...
CLIENT APP
e.g. SPA, windows app, another web app.
2) Client prompts for credentials
Log in
Username
Password
Log in
3) Credentials
7) JWT token sent back to client
4) Credentials passed to API
USER DATABASE
5) Check credentials
6) Credentials valid
WEB API
your API: requires users to be authenticated in order to perform some actions

Cookies

Cookies are small pieces of information that allow a website to recognize a user and their preferences.

Cookies that are set during a session are stateful elements. They hold data that the server transmits to the browser for short-term storage.

Both the client and the server store the authentication data contained in a cookie. The server maintains a database of active sessions, while the browser keeps track of the active session's identification.

When a request is made to the server, the session id is used to check if the session is still valid by looking up information such as user roles or rights for authentication.

Cookies

- Cookies are small pieces of data that the server sends to the client (usually the browser).
- They can be used to maintain sessions, store preferences, or keep track of login status.
- In **NestJS**, you can work with cookies to store things like JWT tokens, which allows the server to identify a user on subsequent requests **without needing them to log in** every time.
- To work with cookies in NestJS, you need to install the `cookie-parser` middleware:

```
npm install cookie-parser
```

Why Use Cookies?

- **Session Persistence**: Cookies are often used to store session data (like user authentication tokens) to keep the user logged in.
- **Security**: Cookies can be flagged as `HttpOnly`, which prevents JavaScript from accessing them (helps mitigate XSS attacks).
- **Automatic Handling**: Browsers automatically include cookies in every request to the same domain.

Token

Tokens are pieces of data that carry just enough information to facilitate the process of determining a user's identity or authorizing a user to perform an action.

All in all, tokens are artifacts that allow application systems to perform the authorization and authentication process.

unique approval given to you (more like a digital signature) for accessing any resource. The application issues it to you once you have authenticated yourself with valid credentials

Access token

- What's an access token?

When a user logins in, the authorization server issues an access token, which is an artifact that client applications can use to make secure calls to an API server.
When a client application needs to access protected resources on a server on behalf of a user, the access token lets the client signal to the server that it has received authorization by the user to perform certain tasks or access certain resources.

limited expiration ie ~15 minutes

Refresh token

- for security purposes, access tokens may be valid for a short amount of time.

- Once they expire, client applications can use a refresh token to "refresh" the access token. That is, a refresh token is a credential artifact that lets a client application get new access tokens without having to ask the user to log in again.

- a longer expiration ie ~3 months

JSON Web Tokens (JWT)

- JWT is a secure way to share information between two parties—like a server and a client.
- It is widely used in authentication to verify users and provide access to resources.
- **Structure:**
  - Header: Metadata (e.g., algorithm).
  - Payload: User claims (e.g., userId, role).
  - Signature: Ensures authenticity(Ensures the token is valid and hasn't been altered).
- Commonly used for stateless authentication.
- Think of a JWT as a digital "pass" that confirms your identity.

Why Use JWT?

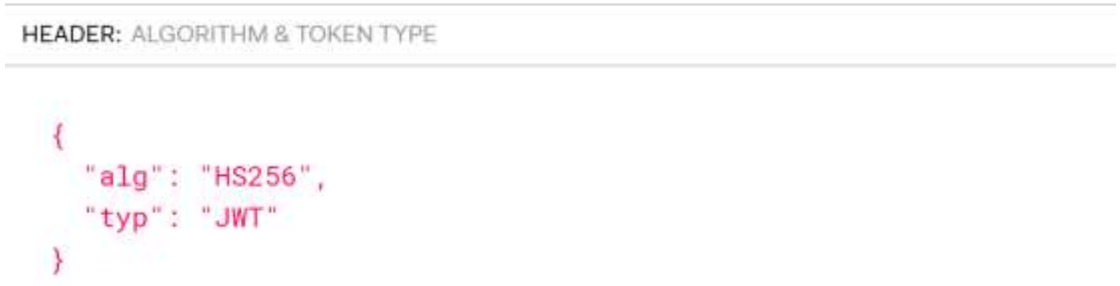
- **Stateless Authentication**: Unlike sessions, JWTs don't require server-side storage. Everything needed to verify the user is inside the token.
- **Secure Communication**: JWTs are signed, meaning the server can verify that the token hasn't been tampered with.
- **Compact and Fast**: Tokens are lightweight and can be easily passed between the client and server.

Dependencies:

- npm install @nestjs/jwt @nestjs/passport
- npm install passport passport-jwt

JWT Header

- **typ** is ‘type’, indicates that Token type here is JWT.
- **alg** stands for ‘algorithm’ which is a hash algorithm for generating Token signature. In the code above, HS256 is HMAC-SHA256 – the algorithm which uses Secret Key.

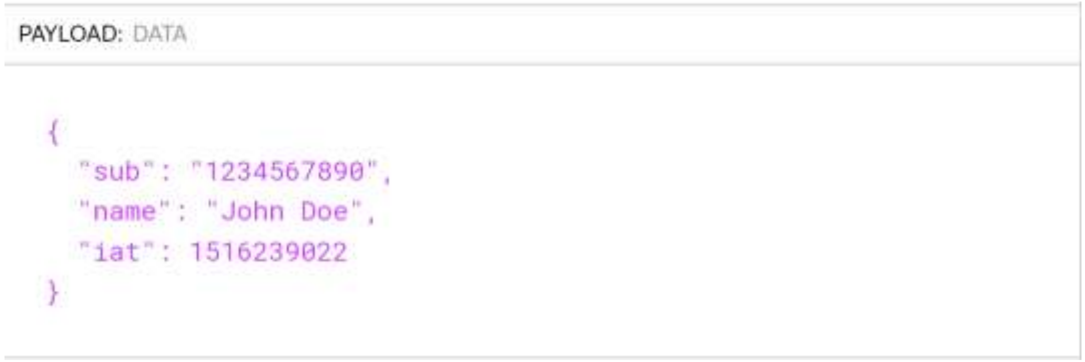


HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

JWT payload

- The Payload helps us to answer: What do we want to store in JWT?
- Sub: subject whom the token refers to
- Name: same as sub
- Iat : issued at time the JWT was issued at
- exp (Expiration Time): JWT expiration time

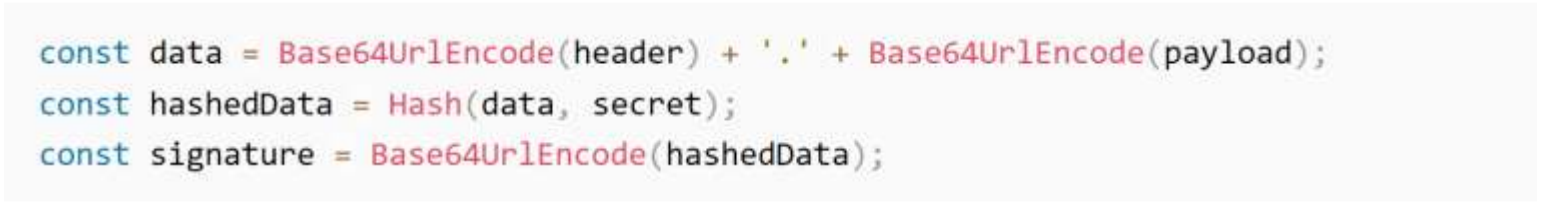


PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

JWT signature

- First, we encode Header and Payload, join them with a dot .
- data = '[encodedHeader].[encodedPayload]'
- Next, we make a hash of the data using Hash algorithm (defined at Header) with a secret string.
- Finally, we encode the hashing result to get Signature.



```
const data = Base64UrlEncode(header) + '.' + Base64UrlEncode(payload);
const hashedData = Hash(data, secret);
const signature = Base64UrlEncode(hashedData);
```

Role Based Authorization

- **Authorization**: This is the process of determining whether a user has the right role and permissions to access a particular resource or perform a specific action.
- **Role-based authorization (RBA)** iis a way to ensure that users can only access certain parts of a system or perform specific actions based on their assigned role, helping to keep the system secure and organized.

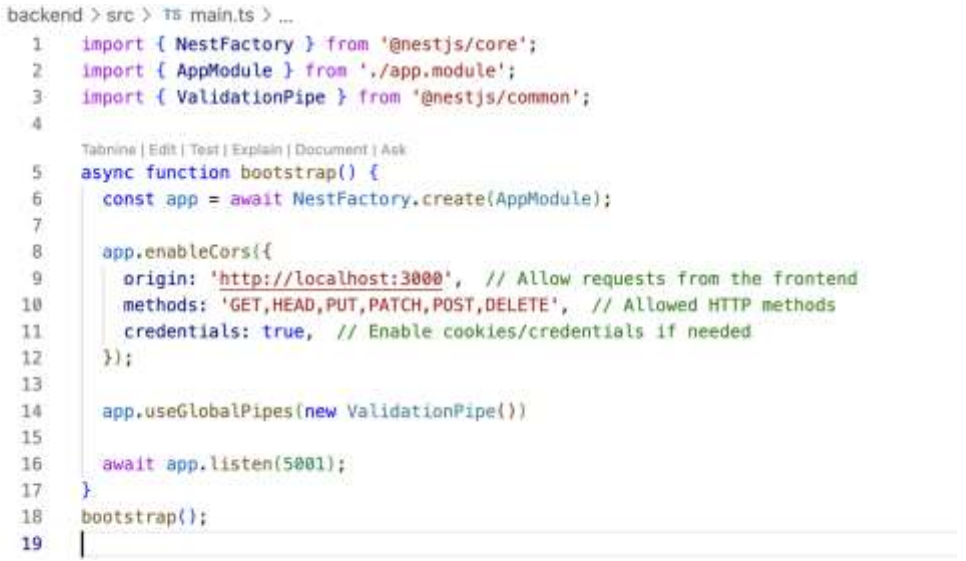
How it works:

1. When a user logs in, their role is identified (usually from a JWT token or session).
2. Based on the role, the system checks if the user has the necessary permissions to access a specific resource or perform an action.
3. If the user has the right permissions for their role, they can proceed. If not, access is denied.

CORS (Cross-Origin Resource Sharing)

- CORS is a security feature implemented in browsers to prevent malicious websites from making requests to another domain without permission.
- In a system with role-based authorization, CORS ensures that only authorized origins (such as specific websites) can communicate with the backend.

- **For example**, if your frontend is hosted at http://localhost:3000 and your backend is at http://localhost:5000, CORS ensures the frontend can safely interact with the backend.


backend > src > TS main.ts > ...
Tabnine | Edit | Test | Explain | Document | Ask

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.enableCors({
    origin: 'http://localhost:3000',  // Allow requests from the frontend
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',  // Allowed HTTP methods
    credentials: true,  // Enable cookies/credentials if needed
  });

  app.useGlobalPipes(new ValidationPipe())

  await app.listen(5001);
}
bootstrap();
```

Guard

- A **Guard** is a mechanism used in backend frameworks (like NestJS) to control access to routes or resources.
- It intercepts a request before it reaches the route handler and checks conditions, such as:
  - Is the user authenticated?
  - Does the user have the right role?

Why is it important?

- Protects sensitive routes and ensures only authorized users can access specific resources.
- Adds an extra layer of security to your application.

How does it work in NestJS?

- You create a Guard (e.g., `RolesGuard`) that checks if a user has the required role for a route.
- Attach the Guard to routes using `@UseGuards()`.

Reflector

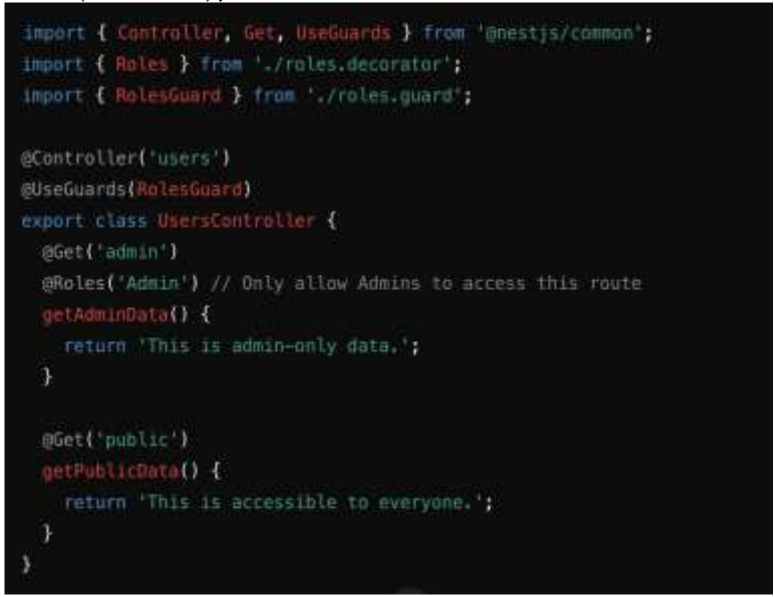
- A Reflector is a tool in NestJS that helps read metadata attached to classes, methods, or routes.
- For example, if you define which roles can access a route (@Roles('Admin')), the Reflector reads this information.

Why is it important?

- Makes it easy to access and use metadata (like required roles) in Guards or other parts of the application.
- Simplifies the process of enforcing role-based authorization.

How does it work?

- Use the Reflector in a Guard to check if the route requires specific roles and validate the user's role.



```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { Roles } from './roles.decorator';
import { RolesGuard } from './roles.guard';

@Controller('users')
@UseGuards(RolesGuard)
export class UsersController {
  @Get('admin')
  @Roles('Admin') // Only allow Admins to access this route
  getAdminData() {
    return 'This is admin-only data.';
  }

  @Get('public')
  getPublicData() {
    return 'This is accessible to everyone.';
  }
}
```

MetaData

- Metadata is extra information added to code elements (like classes or methods) to provide context.
- In role-based authorization, metadata specifies the roles allowed to access a route.

Why is it important?

- Allows you to define access rules for routes in a clear and reusable way.
- Guards can use metadata to check if a request meets the access requirements.

How does it work in NestJS?

- Add metadata to routes using decorators like `@Roles('Admin')`.
- Guards use the Reflector to read this metadata and enforce access control.

.env

A .env file, often referred to as an "environment file," is a simple text file that is commonly used to store configuration settings or sensitive information for an application. It is typically placed in the root directory of a project and named .env.
The purpose of using a .env file is to separate configuration details from the codebase. Instead of hard-coding configuration values directly in your code, you can store them in the .env file. This approach offers several benefits:

1. **Configuration management**: Storing configuration values in a central file makes it easier to manage and update settings for different environments (e.g., development, staging, production) without modifying the code.
2. **Security**: Sensitive information such as API keys, database credentials, or access tokens can be stored in the .env file. Since the file is typically excluded from version control, it helps prevent accidental exposure of sensitive data.
3. **Flexibility**: The .env file allows you to customize the application's behavior by modifying the configuration values without the need to modify the code. This makes it easier to deploy the same codebase with different configuration settings.
4. **Collaboration**: The .env file can be shared among team members, making it easier to maintain consistent configuration settings across different development environments.

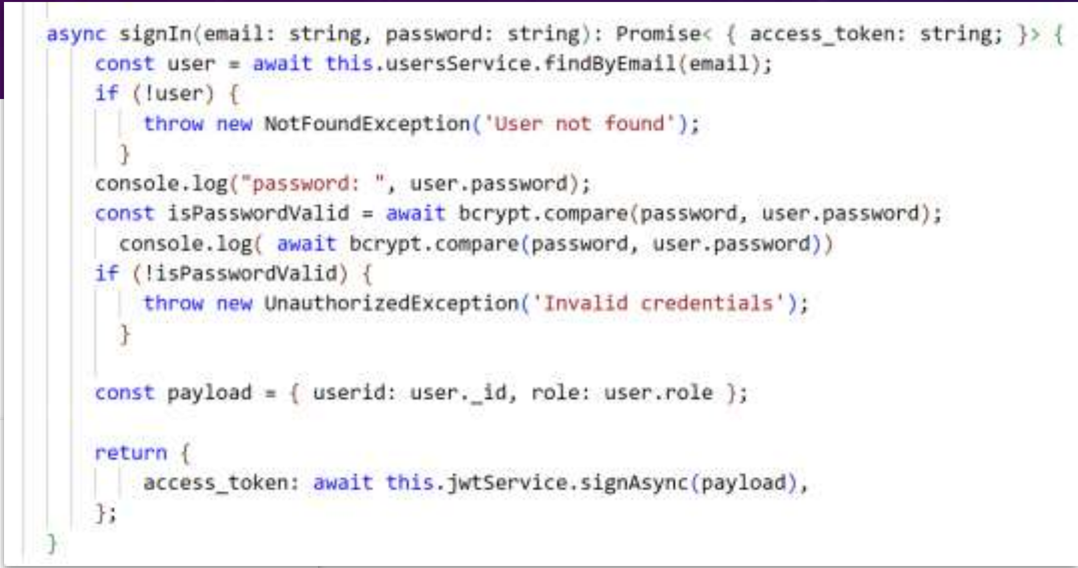
It's important to note that the .env file itself does not have any specific standardized format. However, the most common convention is to use a key-value pair format, where each line represents an environment variable assignment.
For example: VARIABLE_NAME=variable_value.

To use the values stored in a .env file, you need to load them into your application code using appropriate methods or libraries specific to your programming language or framework.

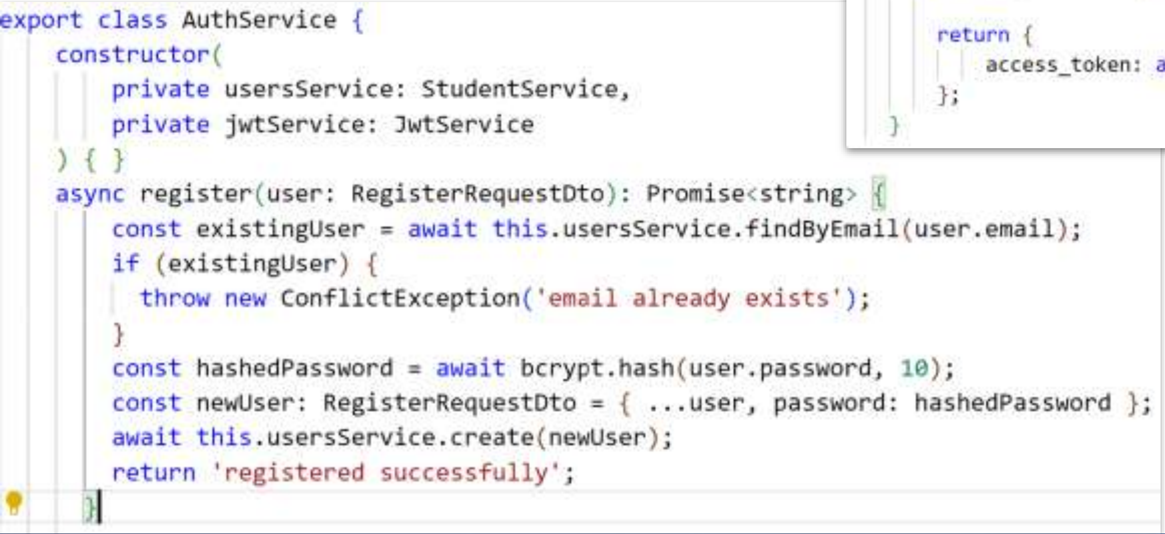
Implementation

Auth Service

- login
- register



```
async signIn(email: string, password: string): Promise< { access_token: string; }> {
    const user = await this.usersService.findByEmail(email);
    if (!user) {
        throw new NotFoundException('User not found');
    }
    console.log("password: ", user.password);
    const isPasswordValid = await bcrypt.compare(password, user.password);
     console.log( await bcrypt.compare(password, user.password))
    if (!isPasswordValid) {
        throw new UnauthorizedException('Invalid credentials');
    }

    const payload = { userid: user._id, role: user.role };

    return {
        access_token: await this.jwtService.signAsync(payload),
    };
}
```

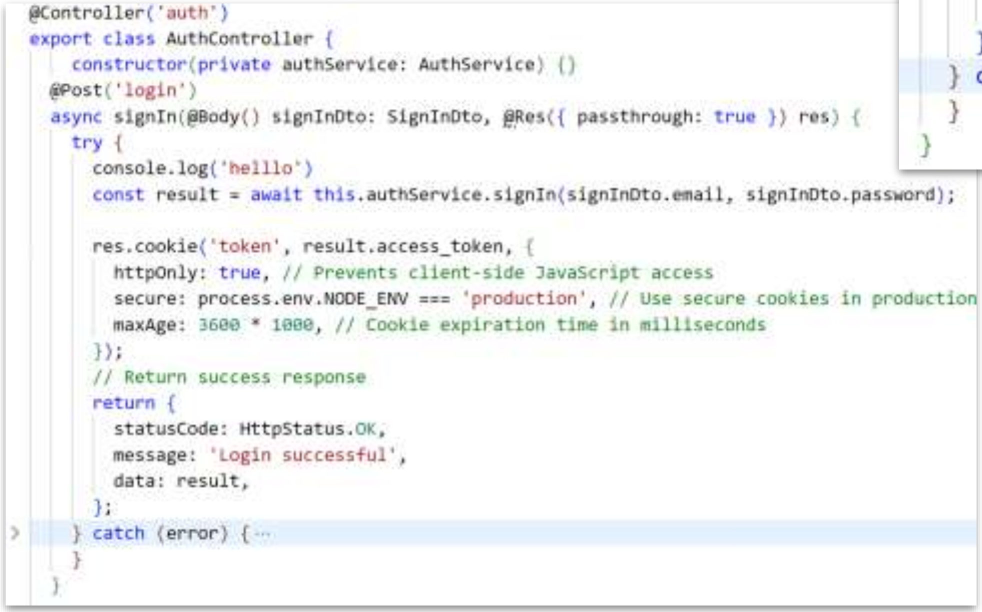


```
export class AuthService {
    constructor(
        private usersService: StudentService,
        private jwtService: JwtService
    ) { }
    async register(user: RegisterRequestDto): Promise<string> {
        const existingUser = await this.usersService.findByEmail(user.email);
        if (existingUser) {
          throw new ConflictException('email already exists');
        }
        const hashedPassword = await bcrypt.hash(user.password, 10);
        const newUser: RegisterRequestDto = { ...user, password: hashedPassword };
        await this.usersService.create(newUser);
        return 'registered successfully';
    }
```

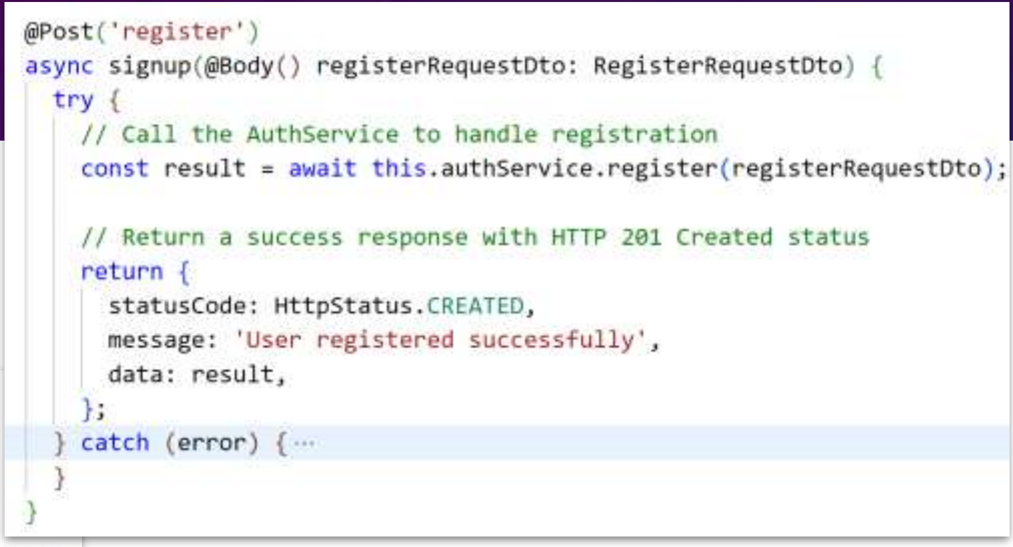
Implementation

Auth Controller

- login
- register



```
@Controller('auth')
export class AuthController {
    constructor(private authService: AuthService) {}
  @Post('login')
  async signIn(@Body() signInDto: SignInDto, @Res({ passthrough: true }) res) {
    try {
      console.log('helllo')
      const result = await this.authService.signIn(signInDto.email, signInDto.password);

      res.cookie('token', result.access_token, {
        httpOnly: true, // Prevents client-side JavaScript access
        secure: process.env.NODE_ENV === 'production', // Use secure cookies in production
        maxAge: 3600 * 1000, // Cookie expiration time in milliseconds
      });
      // Return success response
      return {
        statusCode: HttpStatus.OK,
        message: 'Login successful',
        data: result,
      };
    } catch (error) {···
    }
  }
```

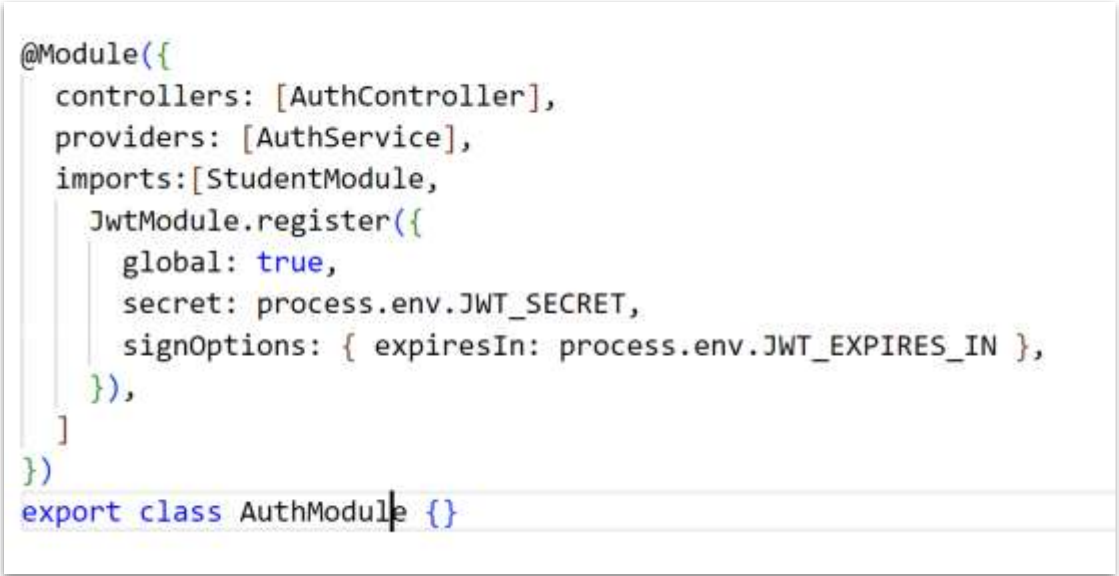


```
@Post('register')
async signup(@Body() registerRequestDto: RegisterRequestDto) {
  try {
    // Call the AuthService to handle registration
    const result = await this.authService.register(registerRequestDto);

    // Return a success response with HTTP 201 Created status
    return {
      statusCode: HttpStatus.CREATED,
      message: 'User registered successfully',
      data: result,
    };
  } catch (error) {···
  }
}
```

Implementation

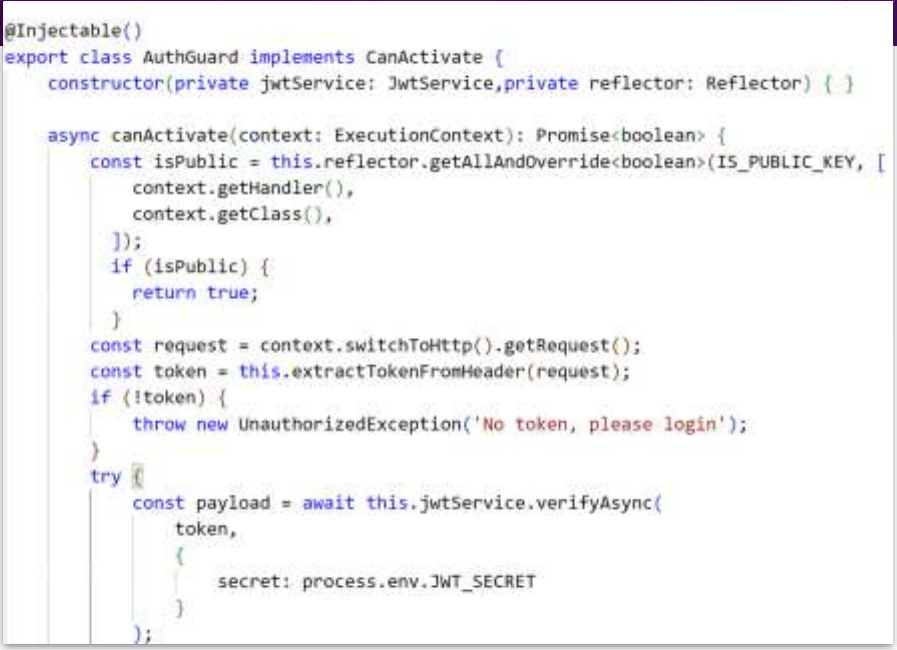
Auth Module

- adding jwt to Module



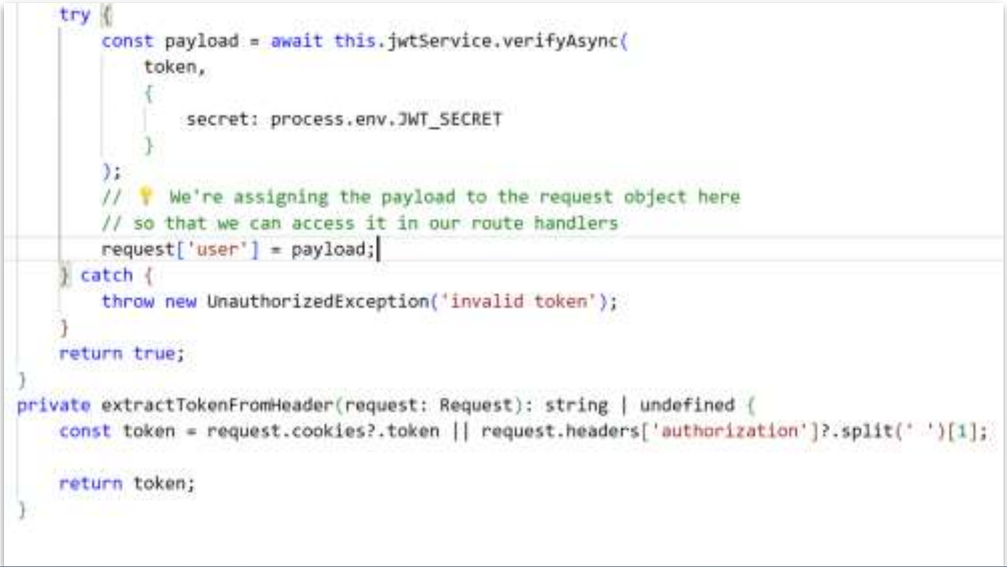
```
@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports:[StudentModule,
    JwtModule.register({
      global: true,
      secret: process.env.JWT_SECRET,
      signOptions: { expiresIn: process.env.JWT_EXPIRES_IN },
    }),
  ]
})
export class AuthModule {}
```

Authentication



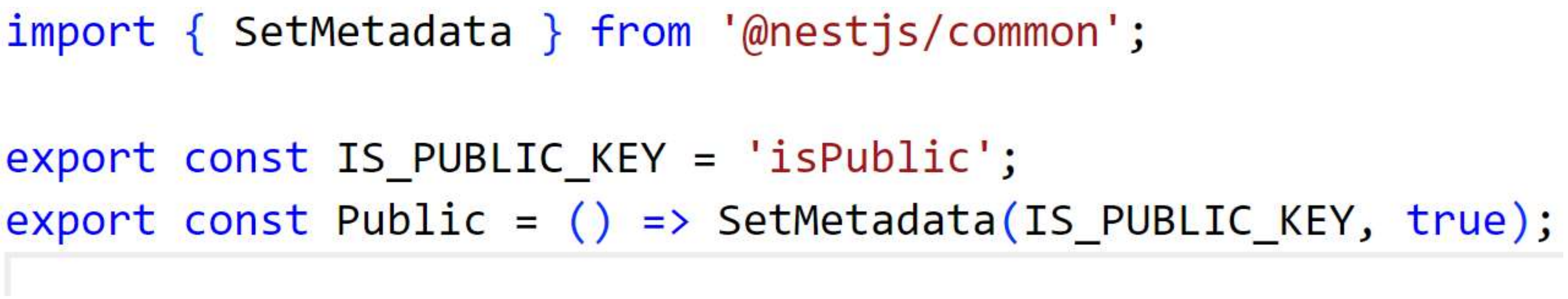
```
@Injectable()
export class AuthGuard implements CanActivate {
    constructor(private jwtService: JwtService,private reflector: Reflector) { }

    async canActivate(context: ExecutionContext): Promise<boolean> {
        const isPublic = this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
            context.getHandler(),
            context.getClass(),
          ]);
          if (isPublic) {
            return true;
          }
        const request = context.switchToHttp().getRequest();
        const token = this.extractTokenFromHeader(request);
        if (!token) {
            throw new UnauthorizedException('No token, please login');
        }
        try {
            const payload = await this.jwtService.verifyAsync(
                token,
                {
                    secret: process.env.JWT_SECRET
                }
            );
```

Authentication Guard



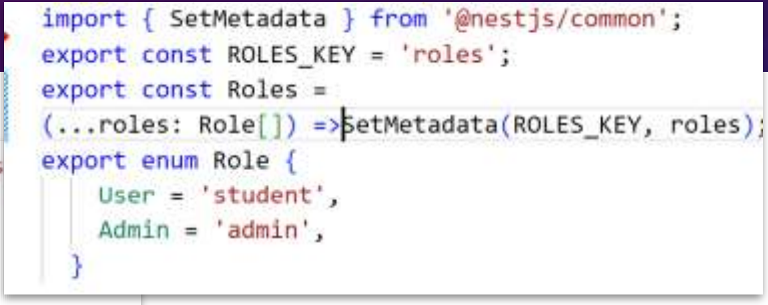
```
        try {
            const payload = await this.jwtService.verifyAsync(
                token,
                {
                    secret: process.env.JWT_SECRET
                }
            );
            // 💡 We're assigning the payload to the request object here
            // so that we can access it in our route handlers
            request['user'] = payload;
        } catch {
            throw new UnauthorizedException('invalid token');
        }
        return true;
    }
    private extractTokenFromHeader(request: Request): string | undefined {
        const token = request.cookies?.token || request.headers['authorization']?.split(' ')[1];

        return token;
    }
```

Public Decorator

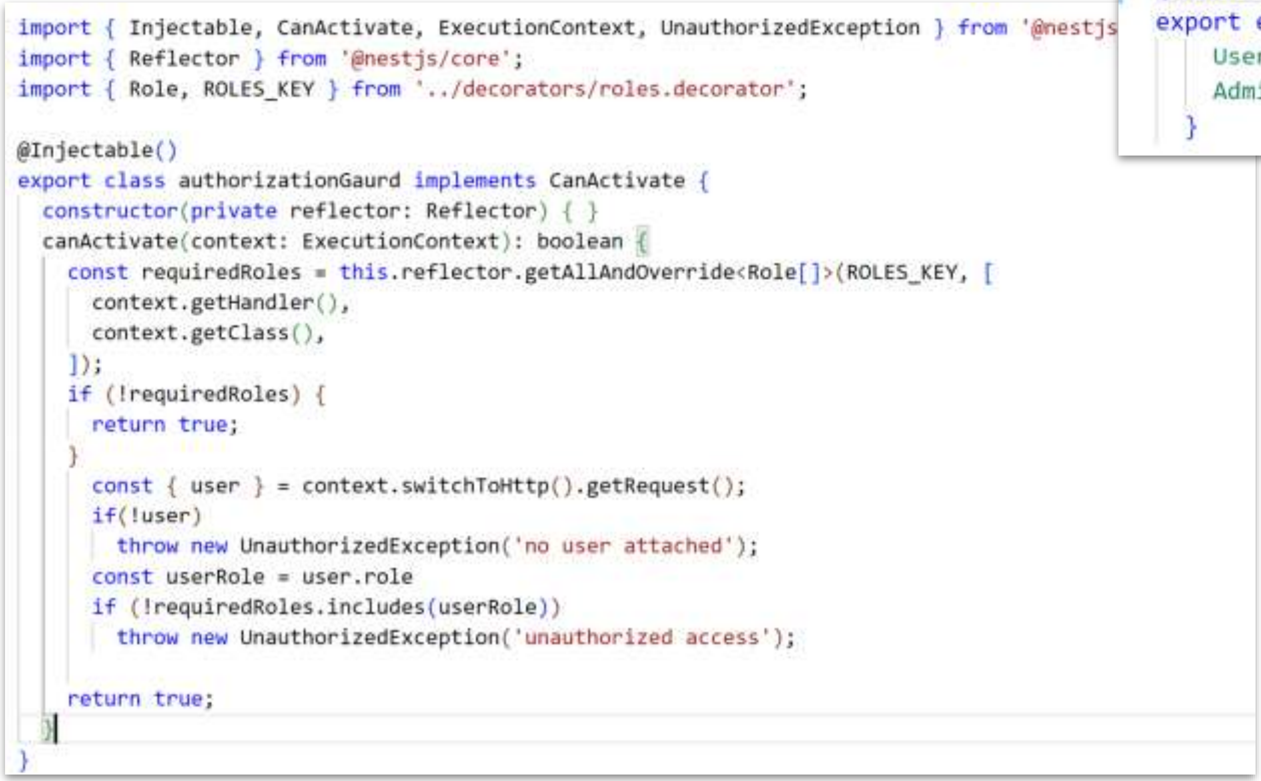


```
import { SetMetadata } from '@nestjs/common';

export const IS_PUBLIC_KEY = 'isPublic';
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

Authorization



```
import { SetMetadata } from '@nestjs/common';
export const ROLES_KEY = 'roles';
export const Roles =
(...roles: Role[]) =>SetMetadata(ROLES_KEY, roles);
export enum Role {
    User = 'student',
    Admin = 'admin',
  }
```

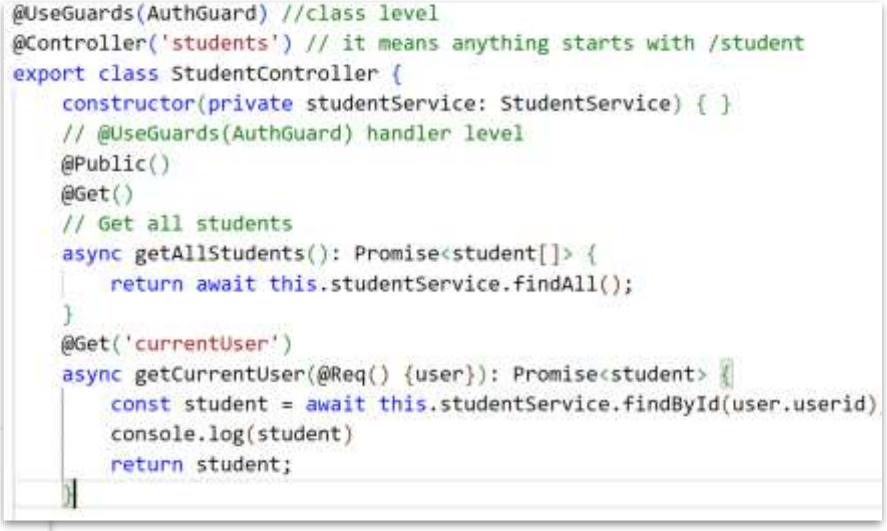


```
import { Injectable, CanActivate, ExecutionContext, UnauthorizedException } from '@nestjs
import { Reflector } from '@nestjs/core';
import { Role, ROLES_KEY } from '../decorators/roles.decorator';

@Injectable()
export class authorizationGaurd implements CanActivate {
  constructor(private reflector: Reflector) { }
  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
      const { user } = context.switchToHttp().getRequest();
      if(!user)
        throw new UnauthorizedException('no user attached');
      const userRole = user.role
      if (!requiredRoles.includes(userRole))
        throw new UnauthorizedException('unauthorized access');

    return true;
  }
}
```

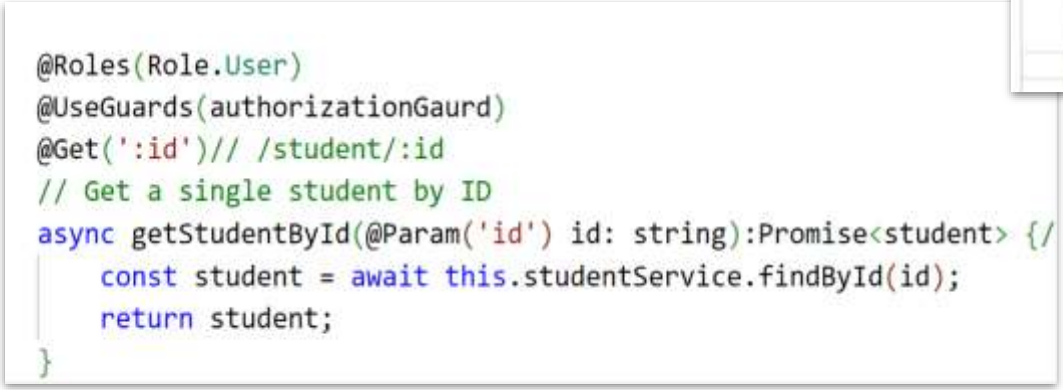
Authorization Guard & roles decorator

Authentication & authorization usage

authentication & authorization guards, public decorator, roles decorator



```
@UseGuards(AuthGuard) //class level
@Controller('students') // it means anything starts with /student
export class StudentController {
    constructor(private studentService: StudentService) { }
    // @UseGuards(AuthGuard) handler level
    @Public()
    @Get()
    // Get all students
    async getAllStudents(): Promise<student[]> {
        return await this.studentService.findAll();
    }
    @Get('currentUser')
    async getCurrentUser(@Req() {user}): Promise<student> {
        const student = await this.studentService.findById(user.userid)
        console.log(student)
        return student;
    }
```



```
@Roles(Role.User)
@UseGuards(authorizationGaurd)
@Get(':id')// /student/:id
// Get a single student by ID
async getStudentById(@Param('id') id: string):Promise<student> {/
    const student = await this.studentService.findById(id);
    return student;
}
```

THANK
YOU