CSM
breaking thought barriers

Angular Training

Session -22

Outlines

CSM
breaking thought barriers

- Lazy Loading
- Routing Guards
- Statement Management using NgRX
- JWT Authentication

Lazy Loading

Lazy loading in Angular is a technique that allows you to load parts of your application only when they are needed, reducing the initial load time and improving performance.

It's commonly used for optimizing large applications with multiple routes.

Step 1: Create an Angular Application

Step 2: Create Feature Modules

Feature modules are modules that contain components, services, and other code related to a specific feature of your application.

ng generate module products
ng generate module orders

Step 3: Define Routes for Feature Modules

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
const routes: Routes = [
  { path: 'products', loadChildren: () => import('./products/ products.module').then(m => m.ProductsModule) },
  { path: 'orders', loadChildren: () => import('./orders/ orders.module').then(m => m.OrdersModule) },
  // Add other routes for non-lazy-loaded components here
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Step 4: Create Components

Within each feature module (products and orders), create the components that belong to that feature.
For instance, In the products module, create a **ProductsListComponent**, and in the orders module, create an **OrdersListComponent**.

Step 5: Update Feature Modules

In each feature module, you need to configure the routes specific to that module. Create a routing module for the feature module and define its routes.

In the products module, create a file called **products- routing.module.ts** :

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ProductsListComponent } from './products-list/products- list. component';
const routes: Routes = [
  { path: '', component: ProductsListComponent },
];
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ProductsRoutingModule { }
```

Step -6 : Similarly, create a routing module for the orders module (orders-routing.module.ts).

Step 7 : Load Feature Modules

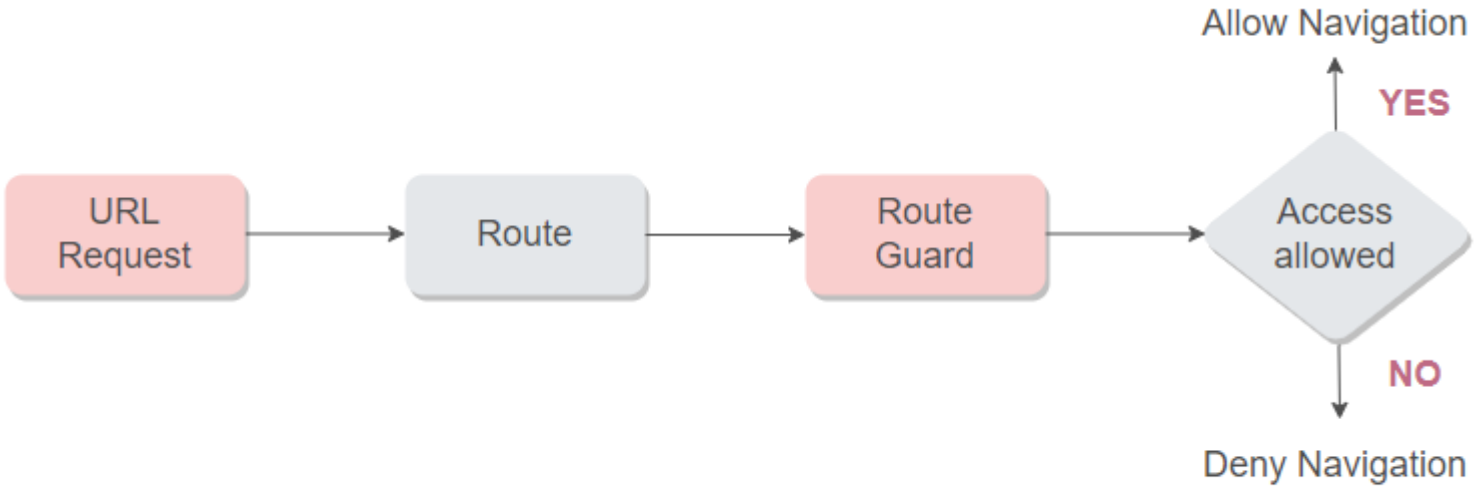
Ensure that the feature modules are properly imported and included in the imports array of your main app module (app.module.ts). Angular's Ahead-of-Time (AOT) compiler will automatically create separate bundles for lazy-loaded modules.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Routing Guards

Angular’s route guards are interfaces that can tell the router whether or not it should allow navigation to a requested route.



Route guard

Scenarios for using a route guard

There are a number of scenarios when you would use a route guard. For example:

- Checking that the user has logged into the application before going to a route
- Checking that the user has the right permissions to go to a certain route
- Loading data before a route has fully loaded
- Saving data when going from one route to another
- Checking whether the user wants to save any unsaved data before leaving to go to another route

Guard interface

In order to cover these different use cases, Angular has a set of five guard interfaces. These are as follows:

Guard interface	Purpose
CanActivate	Check that access is allowed before going to a route
CanActivateChild	Check that access is allowed before going to a child route
CanDeactivate	Decides if a route can be deactivated
Resolve	Retrieves data before loading a route
CanLoad	Checks that access is allowed before loading a module

While each Guard handles different use cases, we can have a class that implements multiple route guards, so we could create a guard that checks to see if we have access to a route.

Implementing a Route Guard

Step 1: Create a TypeScript class

Step 2: Add guard to the route

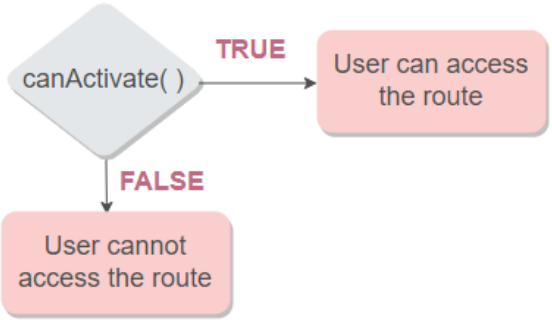
Step 3: Add Guard to AppModule

Step 1: Create a TypeScript class

To implement a route guard, we need to create a TypeScript class that uses one or more of these route interfaces.

By implementing an interface, we need to have a function in our class that the interface says we should have.

If we implement the **CanActivate** interface, we need to have a **canActivate()** function in our class. This function returns true or false. If true, the user can access the route; if false, then they can't.



```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class CanActivateViaAuthGuard implements CanActivate {
    constructor(private authService: AuthService) {}
    canActivate() {
        return this.authService.isLoggedIn();
   }
}
```

Step 2: Add guard to the route

Once we have this guard, we need to add it to the route we want to check.

We do this by adding the CanActivateViaAuthGuard class to the route in the Routes array of our **app-routing . module.ts** file.

```
{
   path: 'client',
   component: ClientPageComponent,
   canActivate: [
      ‘CanAlwaysActivateGuard',
      CanActivateViaAuthGuard
   ]
}
```

Step 3: Add Guard to AppModule

The final step of adding in a route guard is to add the Guard class to the list of providers in our AppModule so that Angular knows where to find the route guard class:

```
@NgModule({
    ...
   providers: [
      AuthService,
      CanActivateViaAuthGuard
   ]
})
export class AppModule {}
```

Authentication and Authorization

Angular

Introduction

- **Authentication** is the process of validating a user on the credentials (username and password) and provide access to the web application(ex: Email)

- **Authorization** helps you to control access rights by granting or denying specific permissions to an authenticated user (Ex: User / Manager / Admin).

- Authorization is applied after the user is authenticated. Typically users are assigned with rights / permissions based on which appropriate section(s) are loaded in the web application

- The user interacts with the server on Authorized sections of the application which results in data exchange. In order to protect security and integrity of data other security components (ex: Encryption) comes into picture

- Security is an inherent and critical feature of a web application. With rich data available in the web server, any compromise results in bigger issues in socio / political ecosystem

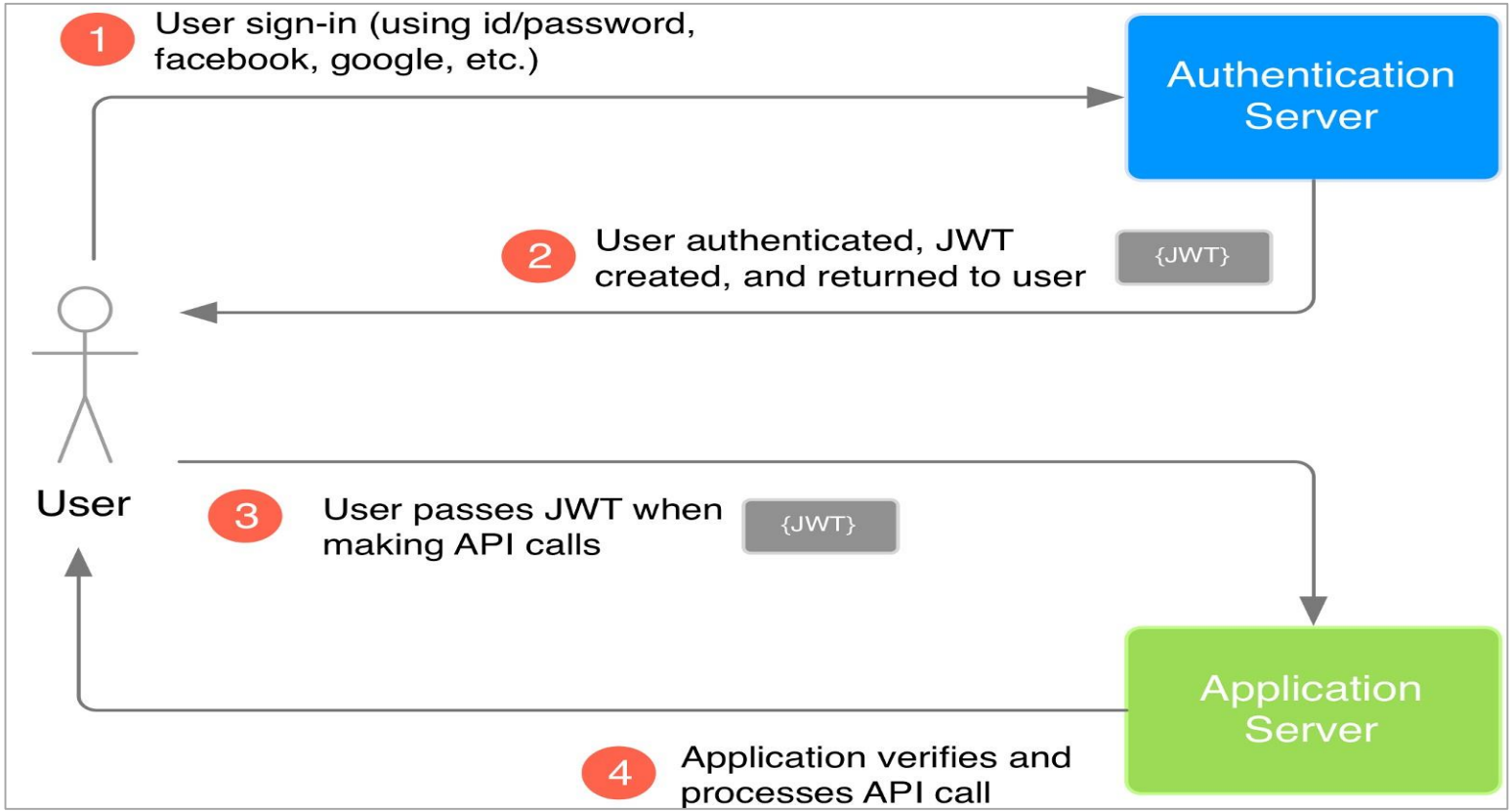
- There are many algorithms, standards and tools in security which is quite vast in nature

- Our idea is to understand security from Angular Authentication and Authorization perspective by practically implementing them in front-end web applications

- We will enhance our understanding of Routes and display / hide certain components based on the user authorization

JSON Web Tokens (JWT)

- JSON Web Token (JWT) is an open standard defined in **RFC 7519**.
- It is a compact and self-contained way for securely transmitting information between parties (ex: Web client and server) as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- JWTs are signed using a secret (ex: HMAC algorithm) which is only known to client & server
- The signed token ensures the data integrity and security


1
User sign-in (using id/password, facebook, google, etc.)
Authentication Server
2
User authenticated, JWT created, and returned to user
{JWT}
User
3
User passes JWT when making API calls
{JWT}
Application Server
4
Application verifies and processes API call

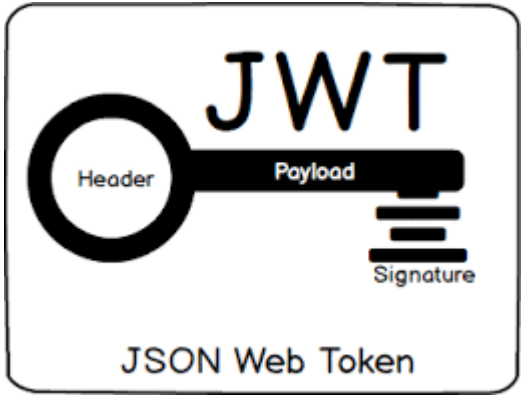
JSON Web Tokens (JWT) – Usage

- JWTs are used in web based authorization once the user is successfully authenticated with valid username & password.

- Each transaction between the client after authorization are done in a secure manner as the data is encrypted.

JSON Web Tokens (JWT) – Structure

- JWT has three parts that are separated by a (.) character
- **Header, Payload and Signature** (ex: xxxx.yyyy.zzzz)
- Each of them have a unique meaning and significance
- An example JWT will look as follows




```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

JWT - Structure

- **Part-I (Header):** Typically consists of two parts:
  - Type of the token (ex: jwt)
  - Hashing algorithm used (ex: HMAC SHA256)

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

- **Part-II (Payload):** It contains claims. Claims are statements about an entity (typically, the user) and additional data.

- Both Header & Payload are encoded using **base64 encoding** and made as a **first and second part of the JWT**

```
{
  "sub": "1234567890",
  "name": "WSA",
  "admin": true
}
```

JWT - Structure

- **Part-III (Signature):** The signature is nothing but a hash algorithm applied on header and payload

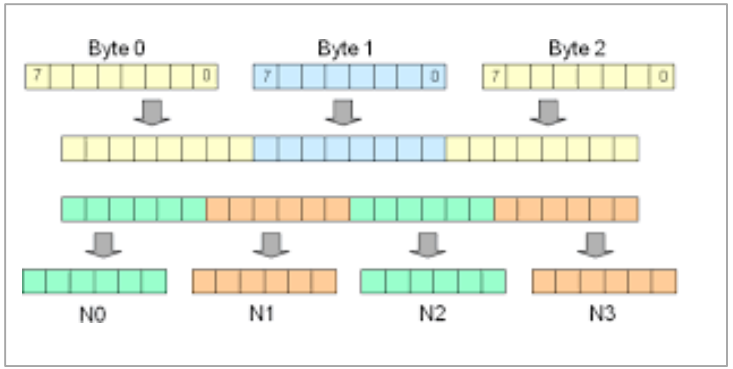
- To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

- For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256 (base64(header) + "." + base64(payload), secret)
```

- The output is three Base64 encoded strings separated by dots that can be easily passed in HTML and HTTP environments

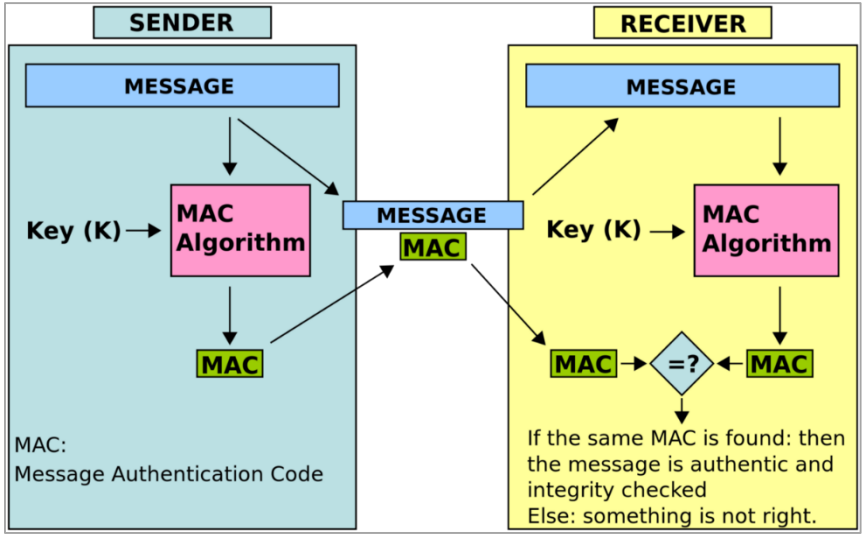
What is base64 Encoding? – A brief



Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

- Base64 converts a string of bytes into a string of ASCII characters so that they can be safely transmitted within HTTP.
- When encoding, Base64 will divide the string of bytes into groups of 6 bits and each group will map to one of 64 characters.
- In case the input is not clearly divisible in 6 bits, additional zeros are added for padding
- Similar to ASCII table a mapping table is maintained

What is HMAC SHA? – A brief



- HMAC (Hash Message Authentication Code) - SHA (Secure Hash Algorithm) is a specific type of message authentication code (MAC)
- It involves a cryptographic hash function and a secret cryptographic key. The key size can vary (ex: SHA 256)
- The secret key is known only to the sender and the receiver
- By applying hashing it generates what is known as signature of the given plain text. It can be used for validating the integrity of the message.

https://jwt.io/#debugger

https://www.base64decode.org

Local Storage

(Storing user data in the browser)

Local storage methods

Local storage supports a set of methods for dealing with the data

Method	Description
`setItem()`	Add key and value to local storage
`getItem()`	Retrieve a value by the key
`removeItem()`	Remove an item by key
`clear()`	Clear all storage

Local storage methods usage

```
localStorage.setItem('key', 'value');
localStorage.getItem('key');
localStorage.removeItem('key');
localStorage.clear();
```

Implementation

Step 1 - The Login Page

Step 2 - Creating a JWT-based user Session

Step 3 - Sending a JWT back to the client

Step 4 - Storing and using the JWT on the client side

Step 5 - Sending The JWT back to the server on each request

Step 6 - Validating User Requests

Implementation

Login Component

Home Component

Customer Component

`_interfaces`

```
export interface LoginModel {
  username: string;
  password: string;
}
```

```
export interface AuthenticatedResponse{
  token: string;
}
```

```
export class LoginComponent implements OnInit {
  invalidLogin: boolean;
  credentials: LoginModel = {username:'', password:''};

  constructor(private router: Router, private http: HttpClient) { }

  ngOnInit(): void {

  }
  login = ( form: NgForm) => {
    if (form.valid) {
      this.http.post< AuthenticatedResponse>("https://localhost:5001/api /auth /login", this.credentials, {
        headers: new HttpHeaders({ "Content-Type": "application/ json "})
      })
      .subscribe({
        next: (response: AuthenticatedResponse) => {
          const token = response.token;
          localStorage.setItem("jwt", token);
          this.invalidLogin = false;
          this.router.navigate(["/"]);
        },
        error: (err: HttpErrorResponse) => this.invalidLogin = true
      })
    }
  }
}
```

Creating Login Form

```
<div class="container">
  <form #loginForm="ngForm" (ngSubmit)="login(loginForm)">
    <h2 class="form-signin-heading">Login</h2>
      <div *ngIf="invalidLogin" class="alert alert-danger">Invalid username or password.</div>
      <br/>
      <label for="username" class="sr-only">Email address</label>
      <input type="email" id="username" name="username" [(ngModel)]="credentials.username"
        class="form-control" placeholder="User Name" required>
      <br/>
      <label for="password" class="sr-only">Password</label>
      <input type="password" id="password" name="password" [(ngModel)]="credentials.password"
        class="form-control" placeholder="Password" required>
      <br/>
      <button class="btn btn-lg btn-primary" type="submit" [disabled]="!loginForm.valid">Log
in</button>
  </form>
</div>
```

Home Component and the Logout Function

```
isUserAuthenticated = (): boolean => {
  return false
}

logOut = () => {
  localStorage.removeItem("jwt");
}
```

```
<h1>Home Page</h1>

<br>
<div *ngIf="isUserAuthenticated()" style="color:blue;">
  <h2>
    YOU ARE LOGGED IN
  </h2>
</div>

<br>
<ul>
  <li><a routerLink="/customers">Customer</a></li>
  <li *ngIf="!isUserAuthenticated()"><a routerLink="/login">Login</a></li>
  <li *ngIf="isUserAuthenticated()"><a class="logout" (click)="logOut()">Log
out</a></li>
</ul>
```

Using Angular Jwt Library With Angular JWT Authentication

npm i @auth0/angular-jwt

```
import { FormsModule } from '@angular/forms';
import { JwtModule } from "@auth0/angular-jwt";
...
export function tokenGetter() {
  return localStorage.getItem("jwt");
}
@NgModule({
  declarations: [
    ...
  ],
  imports: [
    ...
    FormsModule,
    JwtModule.forRoot({
      config: {
        tokenGetter: tokenGetter,
        allowedDomains: ["localhost:5001"],
        disallowedRoutes: []
      }
    })
  ],
  providers: [AuthGuard],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Protecting Angular Routes with Guards

Guards/ auth.guard.ts

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot } from '@angular/router';
import { JwtHelperService } from '@auth0/angular-jwt';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate  {

  constructor(private router:Router, private jwtHelper: JwtHelperService){}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    const token = localStorage.getItem("jwt");

    if (token && !this.jwtHelper.isTokenExpired(token)){
      return true;
    }

    this.router.navigate(["login"]);
    return false;
  }
}
```

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'customers', component: CustomersComponent, canActivate:
[AuthGuard] }
];
```

home.component.ts

```
constructor(private jwtHelper: JwtHelperService) { }

isUserAuthenticated = (): boolean => {
  const token = localStorage.getItem("jwt");

  if (token && !this.jwtHelper.isTokenExpired(token)){
    return true;
  }

  return false;
}
```

Accessing Protected Resources

To access the protected resources we need to send the JWT token in the Authorization header with each request. The server is going to verify the token and grant access to protected resources.

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
@Component({
  selector: 'app-customers',
  templateUrl: './customers.component.html',
  styleUrls: ['./customers.component.css']
})
export class CustomersComponent implements OnInit {
  customers: any;
  constructor(private http: HttpClient) { }
  ngOnInit(): void {
    this.http.get("https://localhost:5001/api/customers")
    .subscribe({
      next: (result: any) => this.customers = result,
      error: (err: HttpErrorResponse) => console.log(err)
    })
  }
}
```

```
<h1>Customers</h1>
<ul>
  <li *ngFor="let cust of customers">{{ cust }}</li>
</ul>
```