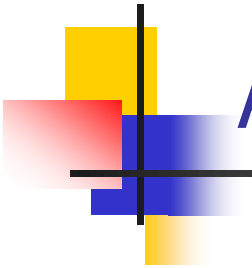
Advanced Application Development (CSE-214)

week-6 : Application Security Concepts

Dr. Alper ÖZCAN

Akdeniz University

alper.ozcan@gmail.com



Application Security Concepts

- Authentication
- Authorization
- OAuth 2
- JSON Web Tokens (JWT)

Authentication

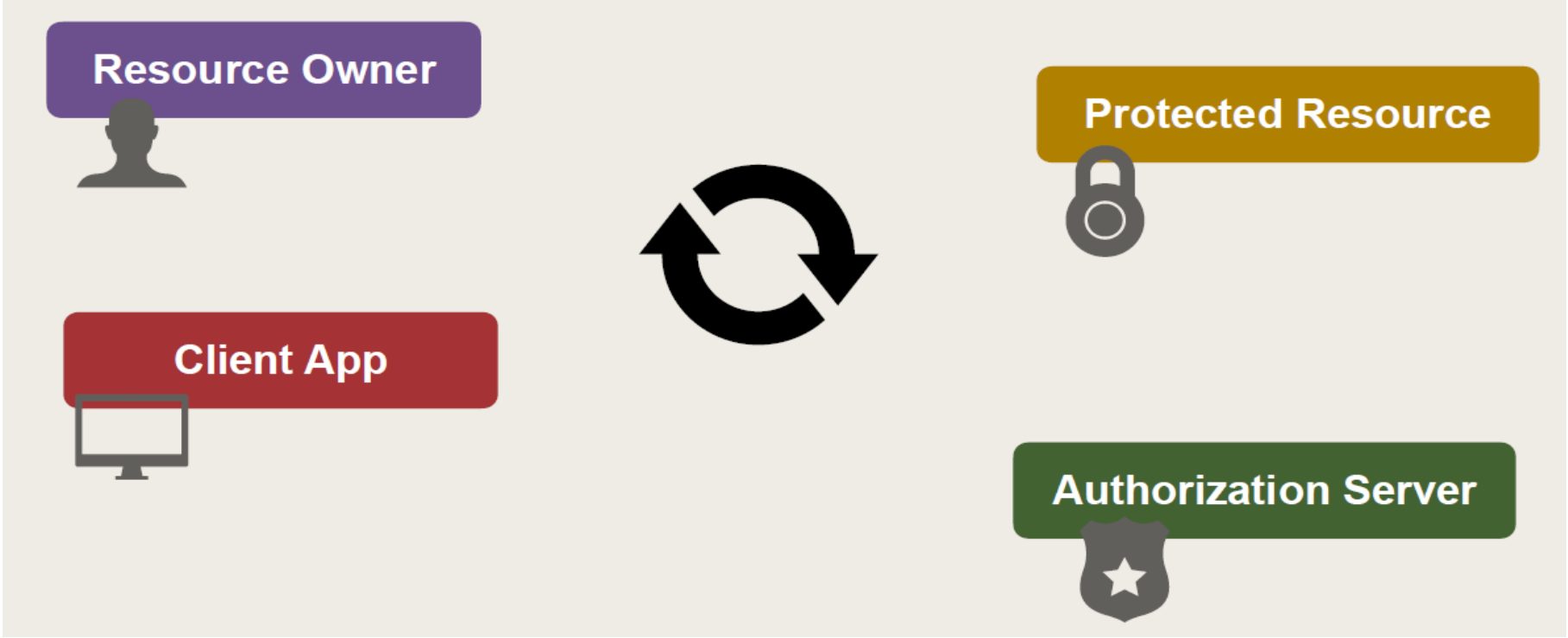
- The process of validating whether a user/app is who they claim to be
  - User name / password
  - Token / pin
  - Finger print / retina scan

Authorization

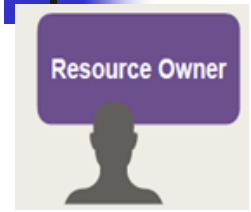
- Process of determining the actions a user / app can perform
- Commonly understood as roles
  - Guest user: minimal actions (read only)
  - Authorized user: read/write data in user account
  - Admin: full access to all accounts system wide

OAuth 2

Authorization framework that enables applications to have limited Access to a resource on behalf of a resource owner (user)



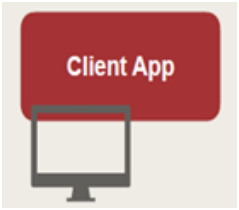
OAuth 2



Resource Owner (User)

•The person who **owns the data** and grants access.

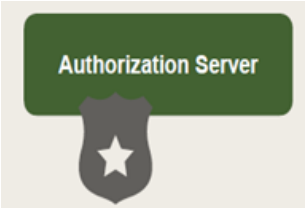
•Example: A **user** who owns their Google Drive files.



Client Application (Frontend App)

•The application that requests access on behalf of the user.

•Example: A **third-party Angular app** that wants to read Google Drive files.



Authorization Server

•Issues **access tokens** after authentication.

•Example: **Google OAuth server** (https://accounts.google.com/o/oauth2/auth).

Resource Server (API)

•Hosts the **protected resource** that requires authentication.

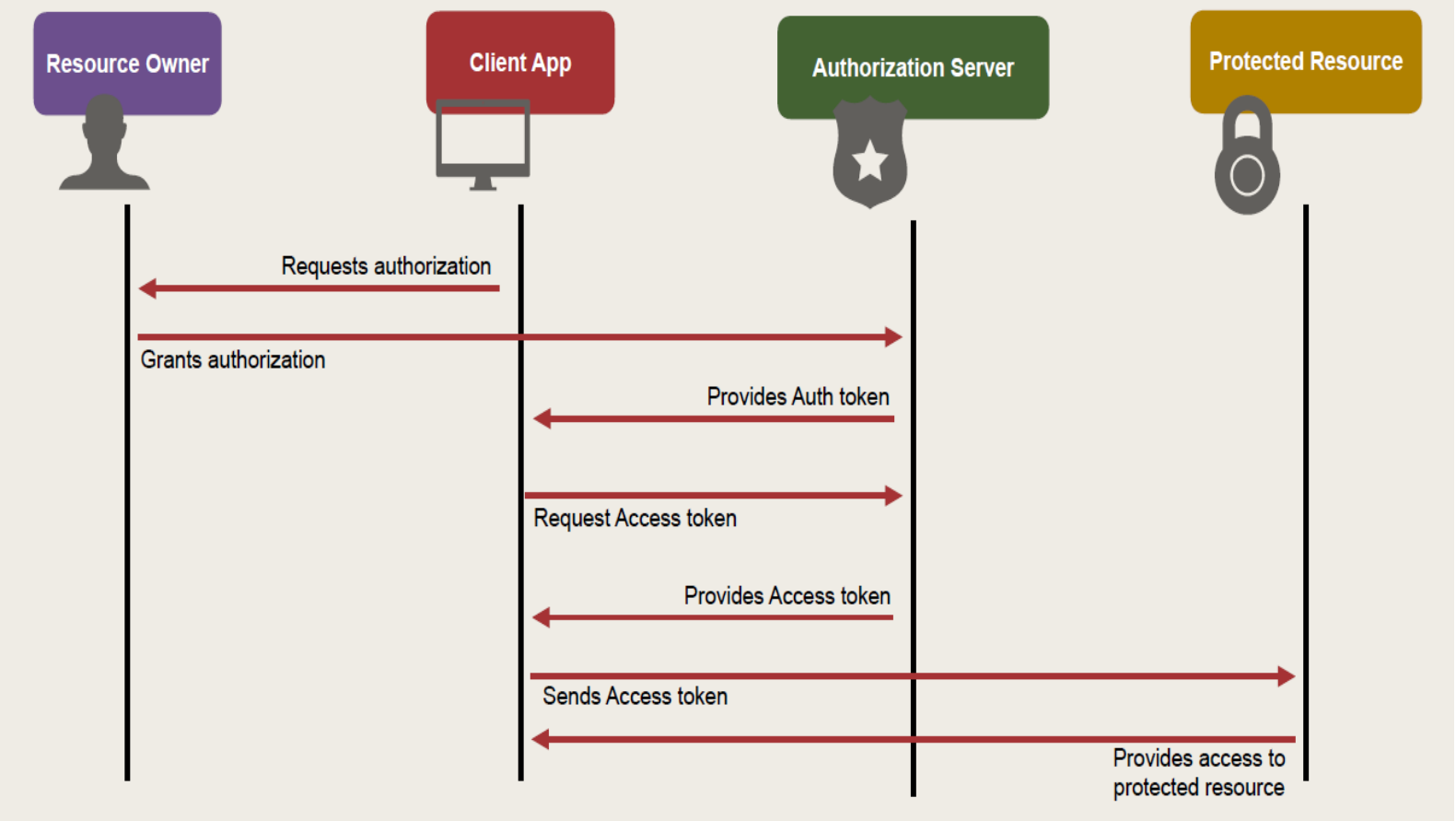
•Example: **Google Drive API** (https://www.googleapis.com/drive/v3/files).

Access Token

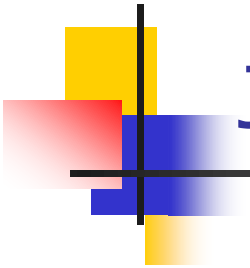
•A **temporary key** (JWT token) that allows access to resources.

•Example: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9... (JWT)

OAuth 2



1. **User logs in** and grants permission.
2. **Authorization server** issues an **authorization code**.
3. **Client app exchanges** the authorization code for an **access token**.
4. The **client app sends the access token** to the resource server.
5. The **resource server validates the token** and allows access.



JSON Web Token (JWT)

- Open standard that defines self-contained way of describing tokens
- Secure and digitally signed to guarantee integrity
- Used by OAuth

JSON Web Token (JWT) and Refresh Token

JSON Web Token (JWT) is a **compact, self-contained, and digitally signed token** used for **authentication and authorization** in web applications. JWTs are widely used in **OAuth 2.0 authentication flows** and **stateless authentication** for APIs.

How JWT Authentication Works:

- **User logs in** with **username & password**.
- **Server verifies credentials** and **creates a JWT**.
- **Client stores the JWT** (LocalStorage, SessionStorage, or HttpOnly Cookie).
- **Client sends JWT** in the Authorization header for each request.

The Problem with JWT Expiry:

•JWTs have an **expiration time (exp)**.

•If the token **expires**, the user is **logged out** and must log in again.

Solution: Refresh Token

•A **refresh token** is a long-lived token stored securely (e.g., in an HttpOnly cookie).

•When the **access token expires**, the refresh token **requests a new JWT** without requiring user login.

Cookies

A **cookie** is a small piece of data stored by a web browser at the request of a website. When a user visits a website, the server can send a **Set-Cookie** header to store a cookie in the user's browser.

```
Set-Cookie: sessionId=abc123; Path=/; Secure; HttpOnly; Expires=Wed, 20 Mar 2025 12:00:00 GMT
```

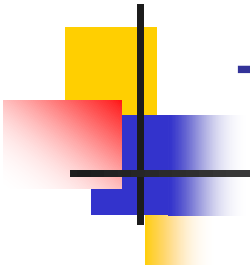
This cookie will be sent automatically with **every request** made to the domain.

Secure : Ensures the cookie is **only sent over HTTPS** (not HTTP).
Without Secure, an attacker intercepting network traffic (e.g., public Wi-Fi) could steal session cookies.

HttpOnly : Prevents JavaScript from accessing the cookie. Protects against **Cross-Site Scripting (XSS)** attacks.

Domain : Specifies **which domains can access the cookie**. This allows **subdomains** (sub.example.com) to access the cookie.

Path : Defines **which paths** can access the cookie. **If Path is /**, the cookie is sent to **all pages**.

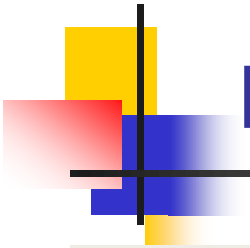


Two Types of Web Storage

- **Session Storage**
  - Data is stored in web browser's memory
- **Local Storage**
  - Data is stored on client side computer

Session Storage

- Stores the data in the web browser's session (memory)
  - Data is never sent to the server (don't confuse with HttpSession)
- Each web browser tab is it's own "session"
  - Data is not shared between web browser tabs
- Once a web browser tab is closed then data is no longer available

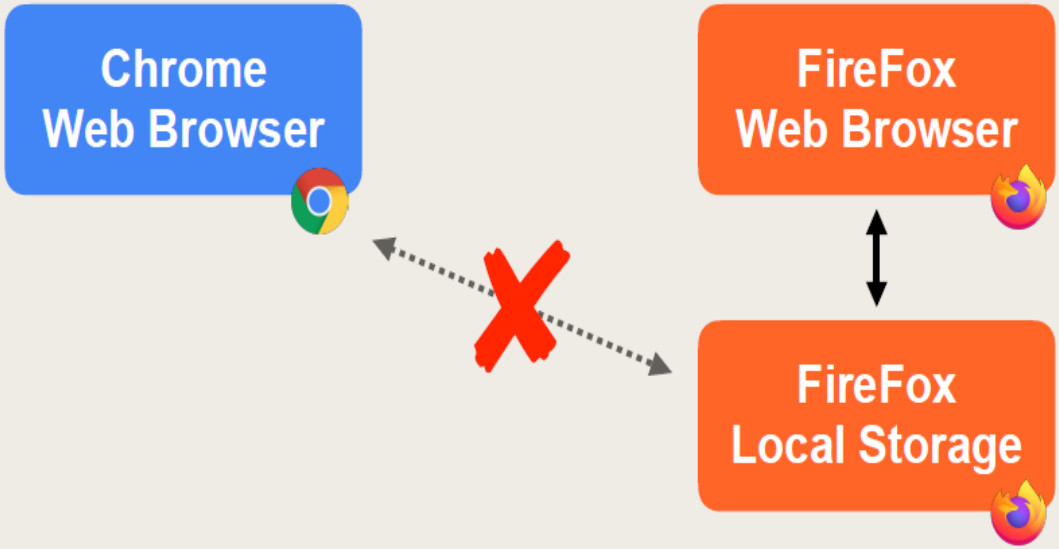


Local Storage

- Stores the data locally on the client web browser computer
  - Data is never sent to the server
- Data is available to tabs of the same web browser for same origin
  - App must read data again ... normally with a browser refresh
- Data persists even if the web browser is closed
  - No expiration date on the data
  - Can clear data using JavaScript or clearing web browser cache

Local Storage

- For Local Storage ... data is NOT shared between different web browsers
- For example ... Chrome can not access Local Storage of FireFox etc ...



Data Scoping

Only pages from the same origin can access the data

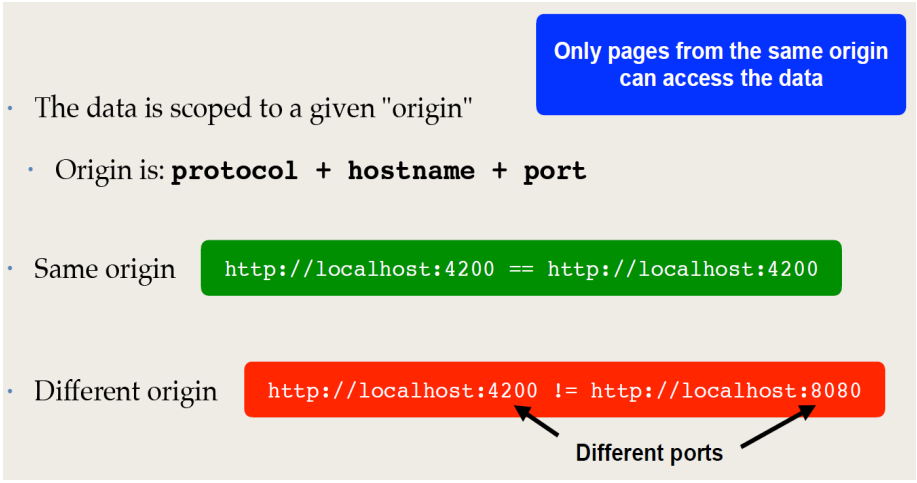
- The data is scoped to a given "origin"
  - Origin is: **`protocol + hostname + port`**
- Same origin `http://localhost:4200 == http://localhost:4200`
- Different origin `http://localhost:4200 != http://localhost:8080`

Understanding Data Scoping and Same-Origin Policy

Data **scoping** refers to how data is restricted to a specific **origin** (protocol + domain + port). Only pages from the **same origin** can access the stored data, ensuring security and preventing unauthorized access.

What is the "Same-Origin Policy"?
The **Same-Origin Policy (SOP)** is a security rule in web browsers that **prevents** web pages from accessing data stored by another origin. This applies to:

- **Cookies**
- **LocalStorage / SessionStorage**
- **IndexedDB**
- **JavaScript variables**



Example: LocalStorage Scoping

If a website https://example.com stores data in **LocalStorage**:

```
localStorage.setItem('authToken', '123456');
```

page from **https://another-site.com cannot** access it:

```
console.log(localStorage.getItem('authToken')); // ✗ Blocked by the browser
```

Example: Cookie Scoping

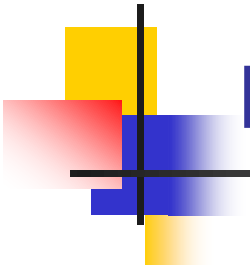
A cookie set for example.com:

```
document.cookie = "sessionId=abc123; Secure";
```

It is **accessible** by https://example.com
It is **NOT accessible** by https://api.example.com

Why is This Important?

- Prevents **cross-site scripting (XSS)** attacks from stealing user data
- Protects sensitive data like **session tokens** from unauthorized access
- Ensures **user privacy**



Factors to consider

- Data in web storage is stored as plain text ... NOT encrypted
  - Do not use it to store sensitive info such as credit card numbers etc ...
  - Be aware that users may tinker with files on their computer
- Your app should be resilient to still work if storage is not available
  - The user may clear their browser cache etc ...
  - Your app should use reasonable defaults

Step 1: Implementing Authentication Service (auth.service.ts)

```typescript
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable, tap } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private token = '';

  constructor(private http: HttpClient) {}

  // Login method to authenticate user and store the token
  login(username: string, password: string): Observable<string> {
    return this.http.post<{ token: string }>('https://fakestoreapi.com/auth/login', { username, password })
      .pipe(
        tap(response => {
          this.token = response.token;
          localStorage.setItem('authToken', this.token); // Save token for future use
        })
      );
  }

  // Logout method
  logout() {
    this.token = '';
    localStorage.removeItem('authToken');
  }

  // Get stored token
  getToken(): string | null {
    return localStorage.getItem('authToken');
  }

  // Check if user is authenticated
  isAuthenticated(): boolean {
    return !!this.getToken();
  }
}
```

- Calls the **Fake Store API** to authenticate users
- **Stores JWT token** in localStorage
- **Provides a method to retrieve and remove the token**

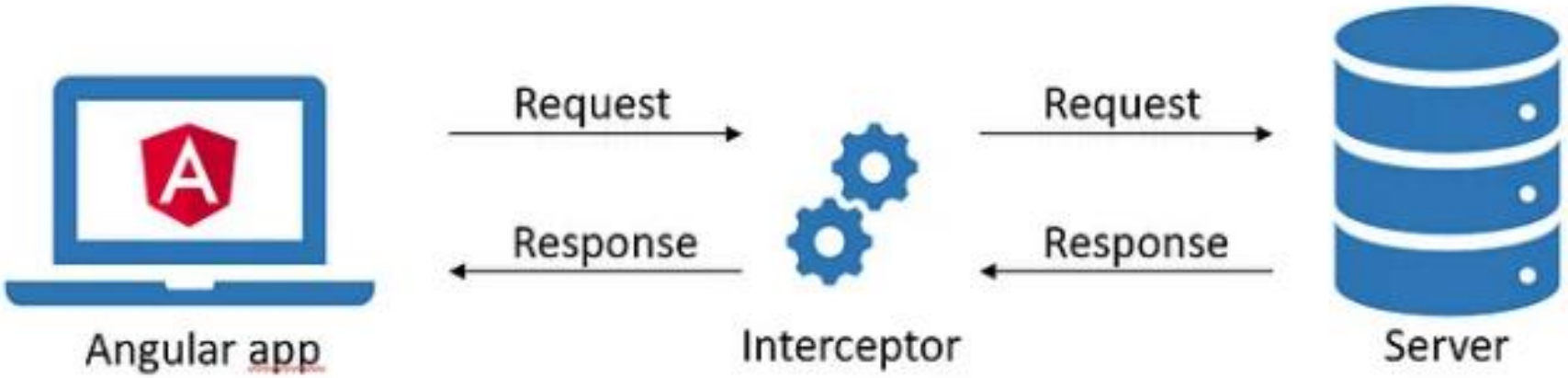
Step 2: Sending Authenticated Requests (product.service.ts)

Once the user is authenticated, we must attach the JWT token in every HTTP request.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private productsUrl = 'https://fakestoreapi.com/products';

  constructor(private http: HttpClient, private authService: AuthService) {}

  getProducts(): Observable<any[]> {
    const token = this.authService.getToken(); // Retrieve token from storage
    const headers = new HttpHeaders().set('Authorization', `Bearer ${token}`);

    return this.http.get<any[]>(this.productsUrl, { headers });
  }
}
```

- **Retrieves JWT token** from localStorage
- **Attaches token** to the Authorization header
- **Sends HTTP request** to get products

Step 3: HTTP Interceptors



- HTTP Interceptors in Angular are classes that implement the **HttpInterceptor** interface.
- They can be used to perform various tasks related to HTTP requests and responses, such as adding headers, handling errors, modifying the request or response data, logging, authentication
- **HttpInterceptor** defines a single method called **intercept**, which takes two parameters: the **HttpRequest** and the **HttpHandler**.

Step 3: Creating an Auth Interceptor (auth.interceptor.ts)

Instead of manually attaching the token in every request, we can use an **Angular HTTP interceptor**.

Implement Interceptor

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = this.authService.getToken();

    if (token) {
      const clonedRequest = request.clone({
        setHeaders: { Authorization: `Bearer ${token}` }
      });
      return next.handle(clonedRequest);
    }

    return next.handle(request);
  }
}
```

- **Intercepts HTTP requests**
- **Automatically attaches JWT token** to Authorization header
- **Reduces redundant code** in every request

Step 3: Creating an Auth Interceptor (auth.interceptor.ts)

Register Interceptor in app.module.ts

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { AuthInterceptor } from './auth.interceptor';

providers: [
  { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
]
```

Step 4: Role-Based Authorization Example (auth.guard.ts)

To **restrict access** to certain routes based on user roles, we can use an **Angular Guard**.

Create an AuthGuard:

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

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (this.authService.isAuthenticated()) {
      return true;
    }
    this.router.navigate(['/login']);
    return false;
  }
}
```

- **Redirects unauthorized users** to login page
- **Ensures only authenticated users** can access protected routes

Step 4: Role-Based Authorization Example (auth.guard.ts)

Protect Routes in app-routing.module.ts

```
import { AuthGuard } from './auth.guard';

const routes: Routes = [
  { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] }
];
```

Step 5: Refresh Token Handling

JWT tokens expire. To keep users logged in, **refresh tokens** can be implemented.

Example API Response with Refresh Token

```
{
  "access_token": "newAccessToken123",
  "refresh_token": "refreshToken789"
}
```

- **Sends expired token** to refresh API
- **Receives a new access token**
- **Updates local storage**

Modify auth.service.ts to Handle Token Refresh

```
refreshToken(): Observable<string> {
  return this.http.post<{ access_token: string }>('https://fakestoreapi.com/auth/refresh', {
    refresh_token: localStorage.getItem('refreshToken')
  }).pipe(
    tap(response => {
      localStorage.setItem('authToken', response.access_token);
    })
  );
}
```

Step 6: Full Example of Sending Authenticated HTTP Requests

```
this.http.get('https://fakestoreapi.com/products', {
  headers: new HttpHeaders().set('Authorization', `Bearer ${this.authService.getToken()}`)
}).subscribe(response => {
  console.log(response);
});
```

or using **Interceptor (Recommended)**:

```
this.http.get('https://fakestoreapi.com/products').subscribe(response => {
  console.log(response);
});
```

Step 6: Full Example of Sending Authenticated HTTP Requests

```
this.http.get('https://fakestoreapi.com/products', {
  headers: new HttpHeaders().set('Authorization', `Bearer ${this.authService.getToken()}`)
}).subscribe(response => {
  console.log(response);
});
```

or using **Interceptor (Recommended)**:

```
this.http.get('https://fakestoreapi.com/products').subscribe(response => {
  console.log(response);
});
```

Step 7: Full JWT Authorization Flow

- User logs in
- Backend returns **JWT Token**
- Token is **stored in localStorage**
- **Interceptor automatically attaches** token in requests
- Protected routes are accessible only if authenticated
- If token **expires, refresh token** is used

Feature	Implementation
Login	`AuthService.login()`
Logout	`AuthService.logout()`
Store Token	`localStorage.setItem()`
Attach Token to Requests	`HttpInterceptor`
Protect Routes	`AuthGuard`
Refresh Token	`refreshToken()`

Error Handling - Authentication Service

Authentication Service with Error Handling (auth.service.ts)

```
export class AuthService {
  private tokenKey = 'authToken';
  private refreshTokenKey = 'refreshToken';
  private apiUrl = 'https://fakestoreapi.com/auth';

  constructor(private http: HttpClient) {}

  login(username: string, password: string): Observable<string> {
    return this.http.post<{ token: string }>(`${this.apiUrl}/login`, { username, password }).pipe(
      tap(response => {
        localStorage.setItem(this.tokenKey, response.token);
      }),
      catchError(this.handleError)
    );
  }

  logout(): void {
    localStorage.removeItem(this.tokenKey);
    localStorage.removeItem(this.refreshTokenKey);
  }

  getToken(): string | null {
    return localStorage.getItem(this.tokenKey);
  }

  isAuthenticated(): boolean {
    return !!this.getToken();
  }

  private handleError(error: HttpErrorResponse) {
    switch (error.status) {
      case HttpStatusCode.InternalServerError:
        console.error('Server error:', error.error);
        break;
      case HttpStatusCode.BadRequest:
        console.error('Request error:', error.error);
        break;
      case HttpStatusCode.Unauthorized:
        console.error('Unauthorized:', error.error);
        this.logout();
        break;
      default:
        console.error('Unknown error:', error.error);
    }
    return throwError(() => error);
  }
}
```

Error Handling - Using an HTTP Interceptor for Token Management (auth.interceptor.ts)

This **automatically attaches the token** to requests and handles **401 Unauthorized errors**.

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent, HttpErrorResponse, HttpStatusC
import { Observable, catchError, EMPTY, throwError } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = this.authService.getToken();

    const authReq = request.clone({
      setHeaders: token ? { Authorization: `Bearer ${token}` } : {}
    });

    return next.handle(authReq).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error.status === HttpStatusCode.Unauthorized) {
          this.authService.logout();
          return EMPTY;
        } else {
          return throwError(() => error);
        }
      })
    );
  }
}
```

- **Automatically attaches JWT tokens**
- **Handles 401 errors by logging out the user**
- **Prevents redundant code in HTTP requests**

Handling HTTP Request Errors (product.service.ts)

- When making API requests, we should **catch and handle errors properly**.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse, HttpHeaders, HttpStatusCode } from '@angular/common/http';
import { Observable, catchError, map, throwError } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private productsUrl = 'https://fakestoreapi.com/products';

  constructor(private http: HttpClient) {}

  getProducts(): Observable<any[]> {
    return this.http.get<any[]>(this.productsUrl).pipe(
      map(products => products.map(product => this.convertToProduct(product))),
      catchError(this.handleError)
    );
  }

  private convertToProduct(product: any) {
    return {
      id: product.id,
      name: product.title,
      price: product.price
    };
  }

  private handleError(error: HttpErrorResponse) {
    switch (error.status) {
      case HttpStatusCode.InternalServerError:
        console.error('Server error:', error.error);
        break;
      case HttpStatusCode.BadRequest:
        console.error('Request error:', error.error);
        break;
      default:
        console.error('Unknown error:', error.error);
    }
    return throwError(() => error);
  }
}
```

Global Error Handler (app-error-handler.ts)

Instead of handling errors in each service, we **centralize error handling**.

```
import { ErrorHandler, Injectable } from '@angular/core';
import { HttpErrorResponse, HttpStatusCode } from '@angular/common/http';

@Injectable()
export class AppErrorHandler implements ErrorHandler {
  handleError(error: any): void {
    const err = error.rejection || error;

    if (err instanceof HttpErrorResponse) {
      switch (err.status) {
        case 0:
          console.error('Client error:', error.error);
          break;
        case HttpStatusCode.InternalServerError:
          console.error('Server error:', error.error);
          break;
        case HttpStatusCode.BadRequest:
          console.error('Request error:', error.error);
          break;
        default:
          console.error('Unhandled error:', error.error);
      }
    } else {
      console.error('Unexpected error:', error);
    }
  }
}
```

Register in app.module.ts

```
import { ErrorHandler } from '@angular/core';
import { AppErrorHandler } from './app-error-handler';

providers: [
  { provide: ErrorHandler, useClass: AppErrorHandler }
]
```

- **Handles errors globally**
- **Prevents crashes by catching and logging errors**