Lecture 09

1. Implement Authentication

1.1 RequestHeader based authentication

We will now add authentication which is based on a Request header value within our API. The authentication logic expects a *RequestHeader* called X-My-Request-Header to be present and have a specific value of Abc123!!!. If either the Header is missing, or the value provided does not match the expected one, we will return an *Unauthorized* API response.

In order to implement the above, do the following in the **CourseAdminSystemBackend** project

1. Add a new folder named **Middleware** under the project *CourseAdminSystem.API*.
2. Add a new class **HeaderAuthenticationMiddleware** inside this folder.
3. Replace the contents of the file as follows.

```
namespace CourseAdminSystem.API.Middleware;

public class HeaderAuthenticationMiddleware {
   private const string MY_SECRET_VALUE = "Abc123!!!";
   private readonly RequestDelegate _next;

   public HeaderAuthenticationMiddleware(RequestDelegate next) {
      _next = next;
   }

   public async Task InvokeAsync(HttpContext context) {
      // 1. Try to retrive the Request Header containing our secret value
      string? authHeaderValue = context.Request.Headers["X-My-Request-Header"];

      // 2. if not found, then return with Unauthorized response
      if (string.IsNullOrWhiteSpace(authHeaderValue)) {
         context.Response.StatusCode = 401;
         await context.Response.WriteAsync("Auth Header value not provided");
         return;
      }

      // 3. If the secret value is NOT correct, return with Unauthorized response
      if (!string.Equals(authHeaderValue, MY_SECRET_VALUE)) {
         context.Response.StatusCode = 401;
         await context.Response.WriteAsync("Auth Header value incorrect");
         return;
      }
```

```
        // 4. Continue with the request
        await _next(context);
    }
}


public static class HeaderAuthenticationMiddlewareExtensions {
    public static IApplicationBuilder UseHeaderAuthenticationMiddleware(this
IApplicationBuilder builder)  {
        return builder.UseMiddleware<HeaderAuthenticationMiddleware>();
    }
}
```

4. To enable the above authentication, go to the *Program.cs* file and update the code as follows.

```
...
app.UseHeaderAuthenticationMiddleware();
app.UseAuthorization();
...
```

5. Run the code and try out one the endpoints to verify that we are now getting an *Unauthorized* exception.
6. Open Postman and try one of the endpoints, e.g. Get students by creating a new request with the following specifications.
    1. Http Method: *GET*
    2. Url: *http://localhost:5016/api/Student*
7. Run the request and verify that it still fails.
8. Now we will add the expected Request header and its value in the Postman request as follows.
    1. Under the *Header* section, add a new *Key* named **X-My-Request-Header** and *Value* as **Abc123!!!**
    2. Verify that the request goes through and we can see the list of students as expected.

1.2 BasicAuthentication

Basic authentication is similar to the above, with a few differences. First of all, the Request Header used is one of the standard ones, called **Authorization**. The value of this is provided as the following

1. Concatenate the *Username* and the *Password*, separated by a semicolon.
2. Convert the concatenated string to a base64encoded string.
3. Prefix the value with **Basic** separated by a space.
4. Use this as the value for the *Authorization* request header.

An example of a value looks like the following

```
Basic am9obi5kb2U6VmVyeVNlY3JldCE=
```

In order to implement this in the API, do the following.

1. Add a new class **BasicAuthenticationMiddleware.cs** inside the *Middleware* folder.
2. Replace the contents of this file as follows.

```
using System.Text;
using Microsoft.AspNetCore.Authorization;
```

```
namespace CourseAdminSystem.API.Middleware;

public class BasicAuthenticationMiddleware {
   // Ideally, we would want to verfy them against a database
   private const string USERNAME = "john.doe";
   private const string PASSWORD = "VerySecret!";

   private readonly RequestDelegate _next;

   public BasicAuthenticationMiddleware(RequestDelegate next) {
      _next = next;
   }

   public async Task InvokeAsync(HttpContext context) {
      // Bypass authentication for [AllowAnonymous]
      if (context.GetEndpoint()?.Metadata.GetMetadata<IAllowAnonymous>() != null) {
         await _next(context);
         return;
      }

      // 1. Try to retrieve the Request Header containing our secret value
      string? authHeader = context.Request.Headers["Authorization"];

      // 2. If not found, then return with Unauthrozied response
      if (authHeader == null) {
         context.Response.StatusCode = 401;
         await context.Response.WriteAsync("Authorization Header value not provided");
         return;
      }

      // 3. Extract the username and password from the value by splitting it on space,
      // as the value looks something like 'Basic am9obi5kb2U6VmVyeVNlY3JldCE='
      var auth = authHeader.Split([' '])[1];

      // 4. Convert it form Base64 encoded text, back to normal text
      var usernameAndPassword = Encoding.UTF8.GetString(Convert.FromBase64String(auth));

      // 5. Extract username and password, which are separated by a semicolon
      var username = usernameAndPassword.Split([':'])[0];
      var password = usernameAndPassword.Split([':'])[1];
      // 6. Check if both username and password are correct
      if (username == USERNAME && password == PASSWORD) {
         await _next(context);
      }
      else {
         // If not, then send Unauthorized response
         context.Response.StatusCode = 401;
         await context.Response.WriteAsync("Incorrect credentials provided");
         return;
      }
   }
}

public static class BasicAuthenticationMiddlewareExtensions {
   public static IApplicationBuilder UseBasicAuthenticationMiddleware(this
IApplicationBuilder builder) {
      return builder.UseMiddleware<BasicAuthenticationMiddleware>();
   }
}
```

3. To enable the above authentication, go to the *Program.cs* file and update the code as follows.

```
...
//app.UseHeaderAuthenticationMiddleware();
app.UseBasicAuthenticationMiddleware();
app.UseAuthorization();
...
```

4. Open Postman, and browse to the request we created earlier, and modify it with the following specifications.
   1. Under the *Auth* section, choose **Basic Auth**.
   2. Provide the *username* and *password* as we specified in our code.
   3. Verify that the request goes through and we can see the list of students as expected.

2. Update Angular App to work with protected endpoints

2.1. Update Service to handle authorization

Our angular app is not yet configured to provide the authentication mechanism that we introduced above. So we will have to update our code where we call the API endpoints, so that we can provide the necessary authorization headers.

1. Open the Angular app and run it.
2. Verify that it is failing with *Unauthorized* response.
3. Edit the *student.service.ts* file and update one of the methods, e.g. getStudents as follows.

```
...
authHeader: string = "Basic am9obi5kb2U6VmVyeVNlY3JldCE=";
...
getStudents(): Observable<Student[]> {
  return this.http.get<Student[]>(this.baseUrl + "/student", {
    headers: {
      "Authorization": this.authHeader
    }
  });
}
```

4. Save the file and try to see if the list of the students is now being loaded.

Exercise

Update the other methods by providing them with the authorisation header values and make sure they work.

3. Implementing security using a login page

Having an authorization header saved within Angular code is not so practical or safe. So we will update our code so that we will use the API to validate the credentials and if correct, return an authorization header which will be used by the Angular app. In order to do so, we'll update our API and Angular app accordingly by following the steps below.

3.1 Update API with a Login Controller

1. Add a new class **Login.cs** under the *Model* folder and replace the contents as follows.

```
using System;
using System.Text.Json.Serialization;

namespace CourseAdminSystem.API.Model;

public class Login {
   [JsonPropertyName("username")]
   public string Username { get; set; }

   [JsonPropertyName("password")]
   public string Password { get; set; }
}
```

2. Add a new controller **LoginController** under the *Controllers* folder, and replace the code as below.

```
using CourseAdminSystem.API.Model;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace CourseAdminSystem.API.Controllers {
    [Route("api/[controller]")]
    [ApiController]
    public class LoginController : ControllerBase {
        // In real world application, these would be saved in a database
        private const string USERNAME = "john.doe";
        private const string PASSWORD = "VerySecret!";

        [AllowAnonymous]
        [HttpPost]
        public ActionResult Login([FromBody] Login credentials) {
            if (credentials.Username == USERNAME && credentials.Password == PASSWORD) {
                // 1. Concatenate username and password with a semicolon
                var text = $"{credentials.Username}:{credentials.Password}";

                // 2. Base64encode the above
                var bytes = System.Text.Encoding.Default.GetBytes(text);
                var encodedCredentials = Convert.ToBase64String(bytes);

                // 3. Prefix with Basic
                var headerValue = $"Basic {encodedCredentials}";
                return Ok(new { headerValue = headerValue });
            }
            else {
                return Unauthorized();
            }
        }
    }
}
```

3. Notice the use of **[AllowAnonymous]**. This is to ensure that we can bypass the authenticating check for this particular endpoint.
4. Test it in swagger to see if the above endpoints works.

3.2 Update Angular App

1. Add a new Service using the following commands.

```
ng generate service services/Auth
```

2. Add a new model to store the response from the Auth API.

```
ng generate interface model/Login
```

3. Replace the contents of *login.ts* as follows..

```
export interface Login {
   headerValue: string
}
```

3. Replace the contents of the *auth.service.ts* as follows.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Login } from '../model/login';

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  baseUrl: string = "http://localhost:5016/api";

  constructor(private http: HttpClient) {
  }

  authenticate(username: String, password: String): Observable<Login> {
    return this.http.post<Login>(`${this.baseUrl}/login`, {
      username: username,
      password: password
    });
  }
}
```

4. Add a new Component called Login using the following commands from within the **app** directory.

```
ng generate component Login
```

5. Replace the contents of *login.component.html* as follows.

```
<div>
   <label>Username</label>
   <input type="text" [(ngModel)]="username"/>
</div>
<div>
   <label>Password</label>
   <input type="password" [(ngModel)]="password" />
</div>
<div>
```

```
   <button type="button" (click)="login()">Login</button>
</div>
@if(!this.authenticated) {
   <span>Credentials invalid</span>
}
```

6. Replace the contents of *login.component.ts* as follows.

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AuthService } from '../services/auth.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './login.component.html',
  styleUrl: './login.component.css'
})
export class LoginComponent {
  username!: String;
  password!: String;

  authenticated = false;

  constructor(public auth: AuthService, private router: Router) {}

  login() {
    if (this.username != null && this.password != null) {
      this.auth.authenticate(this.username, this.password).subscribe((auth) => {
        if (auth != null) {
          // Save to the local storage
          localStorage.setItem('headerValue', auth.headerValue);
          this.authenticated = true;
          this.router.navigate(['student-list'])
        }
      });
    }
  }
}
```

7. Update *student.service.ts* to update the **authHeader** as follows.

```
...
get authHeader(): string { return localStorage["headerValue"]; }
...
```

8. Update the *app.routes.ts* to add a new route for the login page.

```
...
{ path: 'login', component: LoginComponent }
...
```

9. Update *student-list.component.ts* as follows, so that if we are not authenticated when trying to access Student list, then we automatically get redirected to *Login* component

```
...
constructor(private studentService: StudentService, private router: Router) {}


...
ngOnInit(): void {
    if (this.studentService.authHeader == null) {
        this.router.navigate(["login"]);
        return;
    }
    this.studentService.getStudents().subscribe((students) => {
        this.studentList = students;
    });
}
...
```

Exercise

1. Update other components e.g. Dashboard, so that if not authenticated, then user should automatically be redirected to *Login* component.
2. Try implementing a Logout functionality.
    1. Add a new button the AppComponent.
    2. It should only be visible if the user is not already authenticated.
    3. When clicking on it, it should delete the localStorage and redirect to login page.