Task 5: JWT Authentication and Role-Based Authorization in Web API

Objectives

- Explain and implement CORS for local access to Web API.
- Secure Web API using Bearer and JWT token authentication.
- Use `[Authorize]` and `[AllowAnonymous]` attributes appropriately.
- Add claims, roles, and expiration handling in JWT.
- Test secured endpoints using Postman.

1. What is CORS?

CORS (Cross-Origin Resource Sharing) is a browser security feature that restricts web pages from making requests to a different domain. When developing client-side applications (like Angular or React) that access a local Web API, CORS must be enabled.

Enabling CORS in .NET Core:

Install the CORS package (optional for .NET Core >= 3.0):

```
dotnet add package Microsoft.AspNetCore.Cors
```

Register CORS in `Program.cs`:

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAll", builder =>
    {
        builder.AllowAnyOrigin()
               .AllowAnyHeader()
               .AllowAnyMethod();
    });
});
```

Enable it in the middleware pipeline:

```
app.UseCors("AllowAll");
```

2. JWT Authentication Setup

Step 1: Configure JWT in `Program.cs`

```
string securityKey = "mysuperdupersecret";
var symmetricSecurityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(securityKey));

builder.Services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(x =>
{
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "mySystem",
        ValidAudience = "myUsers",
        IssuerSigningKey = symmetricSecurityKey
    };
});
```

Enable authentication in the middleware:

```
app.UseAuthentication();
```

---

3. Create `AuthController` to Generate JWT Token

```
[AllowAnonymous]
[ApiController]
[Route("api/[controller]")]
public class AuthController : ControllerBase
{
    [HttpGet]
    public IActionResult GetToken()
    {
        var token = GenerateJSONWebToken(1, "Admin");
        return Ok(new { token });
    }

    private string GenerateJSONWebToken(int userId, string userRole)
    {
        var securityKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("mysuperdupersecret"));
        var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);

        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Role, userRole),
            new Claim("UserId", userId.ToString())
        };

        var token = new JwtSecurityToken(
            issuer: "mySystem",
            audience: "myUsers",
            claims: claims,
            expires: DateTime.Now.AddMinutes(2), // Can be changed to 10 for longer
session
            signingCredentials: credentials
```

```
        );

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}
```

---

4. Securing `EmployeeController` with Roles

Step 1: Remove `CustomAuthFilter`

Ensure the controller uses `[Authorize]`:

```
[Authorize(Roles = "Admin,POC")]
[ApiController]
[Route("api/[controller]")]
public class EmployeeController : ControllerBase
{
    // Methods here
}
```

---

5. Testing with Postman

a. Generate Token

- Send `GET` request to:
- `http://localhost:<port>/api/Auth`
- Copy the `token` value from response.

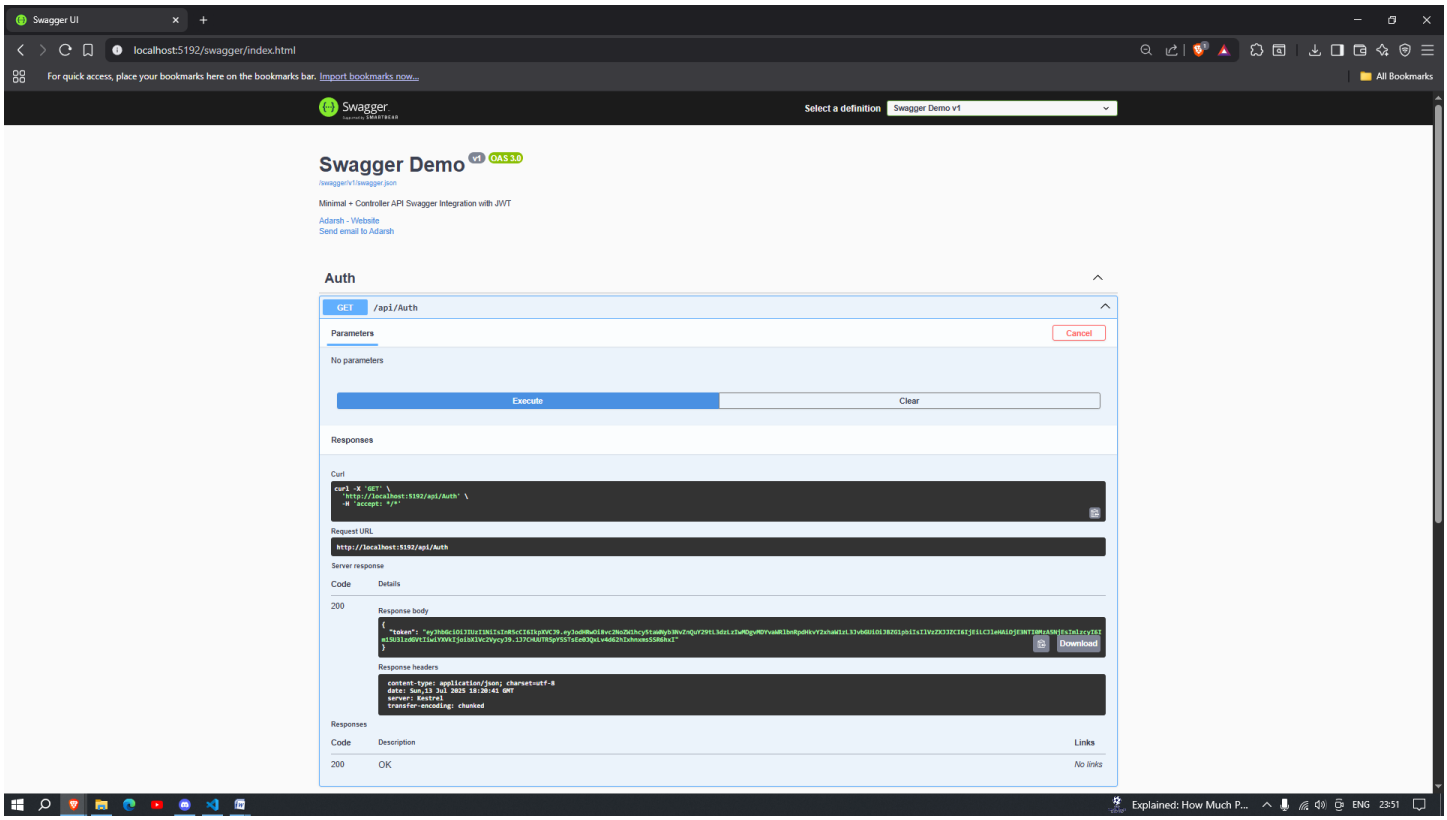
b. Call Secured Endpoint

- Set method: `GET`
- URL: `http://localhost:<port>/api/Employee`
- Headers:
- `Key: Authorization`
- `Value: Bearer <copied_token>`

Expected Results:

Test Case	Expected Result
No Authorization header	401 Unauthorized
Token expired after 2 minutes	401 Unauthorized
Wrong role in token	401 Unauthorized
Correct token with Admin/POC role	200 OK with response data

Screenshots



Conclusion

In this task, JWT-based authentication was successfully implemented in the Web API. The `AuthController` dynamically generates tokens with role-based claims, and the `[Authorize]` attribute secures sensitive endpoints. This pattern ensures that only authorized users can access protected resources, and tokens are validated based on their issuer, audience, and expiration.