**SUPERSET ID: 6364957  
JWT Microservice Implementation**

**Part 1: Project Setup**

**Step 1: Creating Project**

Create and open the project:

dotnet new webapi -n JwtMicroservice  
cd JwtMicroservice  
code .

**Step 2: Installing Required Packages**

In the terminal:

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer  
dotnet add package System.IdentityModel.Tokens.Jwt

**Part 2: Assignment 1 – Basic JWT Authentication**

**Step 3: Creating the Models**

* Creating a Models folder.
* Adding LoginModel.cs:

namespace JwtMicroservice.Models  
{  
 public class LoginModel  
 {  
 public string Username { get; set; } = string.Empty;  
 public string Password { get; set; } = string.Empty;  
 }  
}

* Adding User.cs:

namespace JwtMicroservice.Models  
{  
 public class User  
 {  
 public int Id { get; set; }  
 public string Username { get; set; } = string.Empty;  
 public string Password { get; set; } = string.Empty;  
 public string Role { get; set; } = string.Empty;  
 }  
}

**Step 4: Updating appsettings.json**

Replacing contents with:

{  
 "Logging": {  
 "LogLevel": {  
 "Default": "Information",  
 "Microsoft.AspNetCore": "Warning"  
 }  
 },  
 "AllowedHosts": "\*",  
 "Jwt": {  
 "Key": "ThisIsASecretKeyForJwtTokenWithAtLeast256Bits",  
 "Issuer": "MyAuthServer",  
 "Audience": "MyApiUsers",  
 "DurationInMinutes": 60  
 }  
}

**Step 5: Configure JWT Authentication (Program.cs)**

Replacing contents with:

using Microsoft.AspNetCore.Authentication.JwtBearer;  
using Microsoft.IdentityModel.Tokens;  
using System.Text;  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddControllers();  
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();  
  
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)  
 .AddJwtBearer(options =>  
 {  
 options.TokenValidationParameters = new TokenValidationParameters  
 {  
 ValidateIssuer = true,  
 ValidateAudience = true,  
 ValidateLifetime = true,  
 ValidateIssuerSigningKey = true,  
 ValidIssuer = builder.Configuration["Jwt:Issuer"],  
 ValidAudience = builder.Configuration["Jwt:Audience"],  
 IssuerSigningKey = new SymmetricSecurityKey(  
 Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]!))  
 };  
 });  
  
builder.Services.AddAuthorization();  
  
var app = builder.Build();  
  
if (app.Environment.IsDevelopment())  
{  
 app.UseSwagger();  
 app.UseSwaggerUI();  
}  
  
app.UseHttpsRedirection();  
app.UseAuthentication();  
app.UseAuthorization();  
  
app.MapControllers();  
app.Run();

**Step 6: AuthController**

* Creating Controllers/AuthController.cs:

using Microsoft.AspNetCore.Mvc;  
using Microsoft.IdentityModel.Tokens;  
using System.IdentityModel.Tokens.Jwt;  
using System.Security.Claims;  
using System.Text;  
using JwtMicroservice.Models;  
  
namespace JwtMicroservice.Controllers  
{  
 [ApiController]  
 [Route("api/[controller]")]  
 public class AuthController : ControllerBase  
 {  
 private readonly IConfiguration \_configuration;  
  
 public AuthController(IConfiguration configuration)  
 {  
 \_configuration = configuration;  
 }  
  
 [HttpPost("login")]  
 public IActionResult Login([FromBody] LoginModel model)  
 {  
 if (IsValidUser(model))  
 {  
 var token = GenerateJwtToken(model.Username);  
 return Ok(new { Token = token, Message = "Login successful" });  
 }  
 return Unauthorized(new { Message = "Invalid username or password" });  
 }  
  
 private bool IsValidUser(LoginModel model)  
 {  
 return model.Username == "admin" && model.Password == "password123";  
 }  
  
 private string GenerateJwtToken(string username)  
 {  
 var claims = new[]  
 {  
 new Claim(ClaimTypes.Name, username),  
 new Claim(ClaimTypes.NameIdentifier, "1"),  
 new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())  
 };  
  
 var key = new SymmetricSecurityKey(  
 Encoding.UTF8.GetBytes(\_configuration["Jwt:Key"]!));  
 var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);  
  
 var token = new JwtSecurityToken(  
 issuer: \_configuration["Jwt:Issuer"],  
 audience: \_configuration["Jwt:Audience"],  
 claims: claims,  
 expires: DateTime.Now.AddMinutes(  
 Convert.ToDouble(\_configuration["Jwt:DurationInMinutes"])),  
 signingCredentials: creds);  
  
 return new JwtSecurityTokenHandler().WriteToken(token);  
 }  
 }  
}

**Step 7: Testing Authentication**

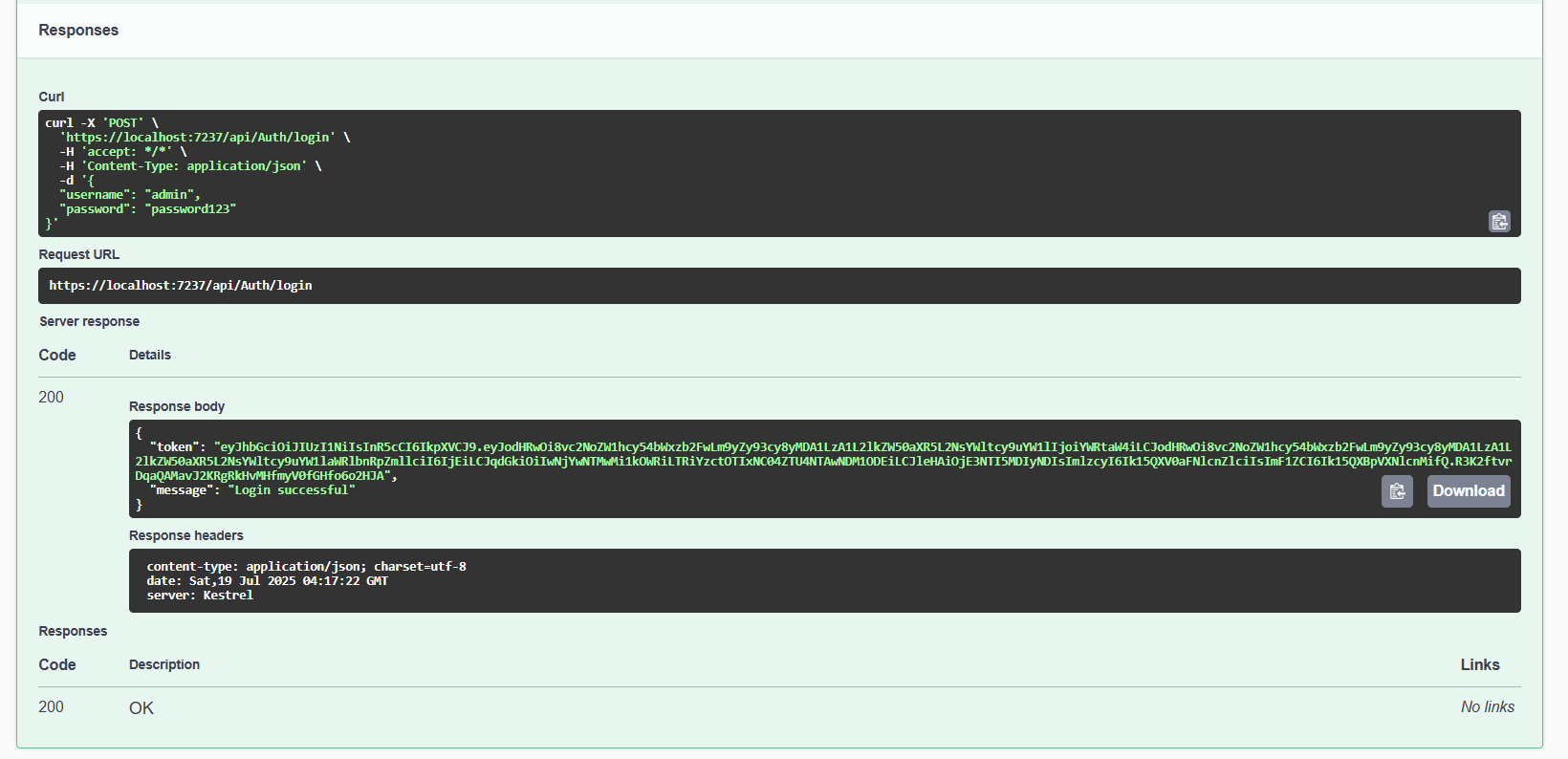
* Runing with:

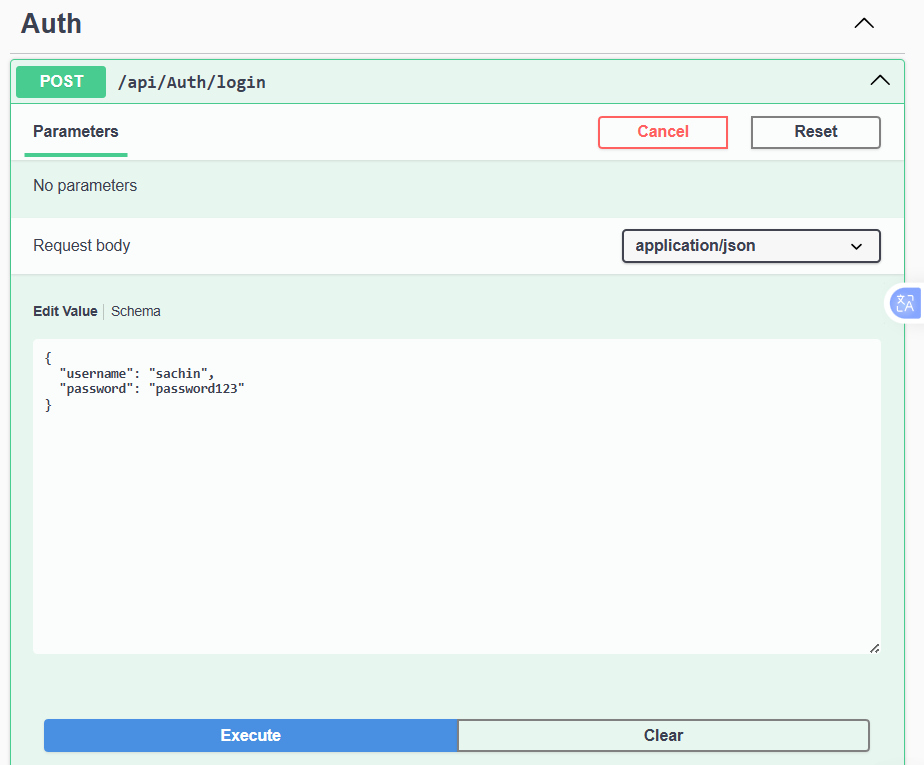
dotnet run

* Going to https://localhost:7237/swagger.
* Testing login by clicking “Try it out” at POST /api/Auth/login with JSON:

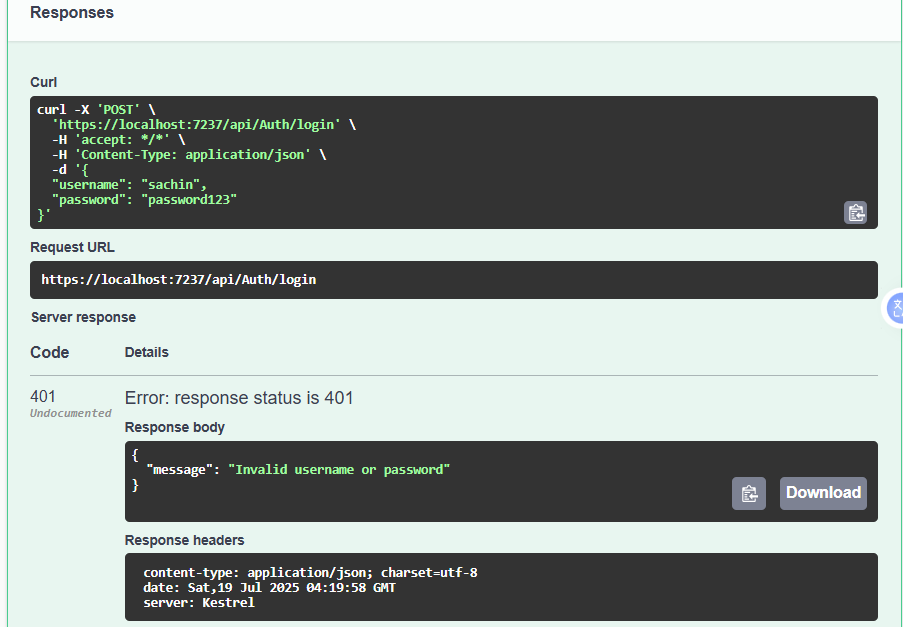


**Execution Result:**



* Testing with incorrect credentials to get an unauthorized error.  
  

**Execution Result:**



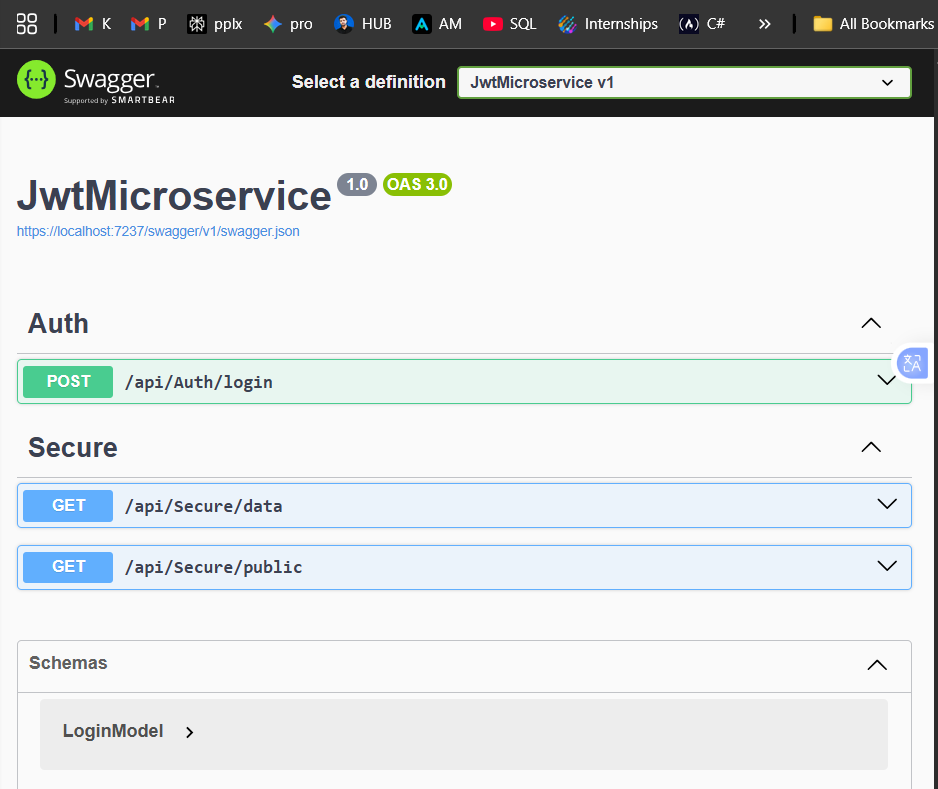
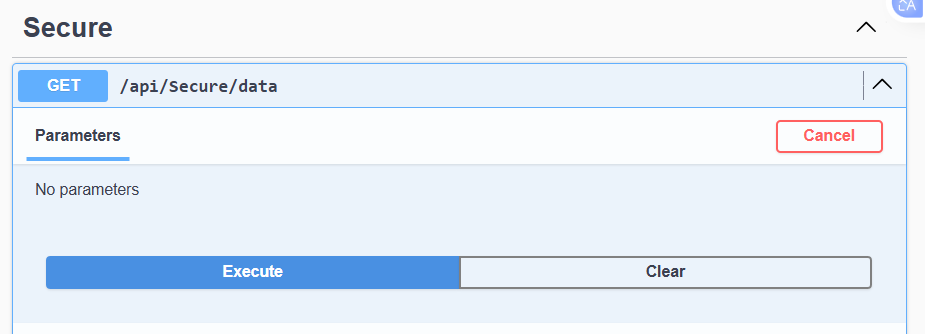
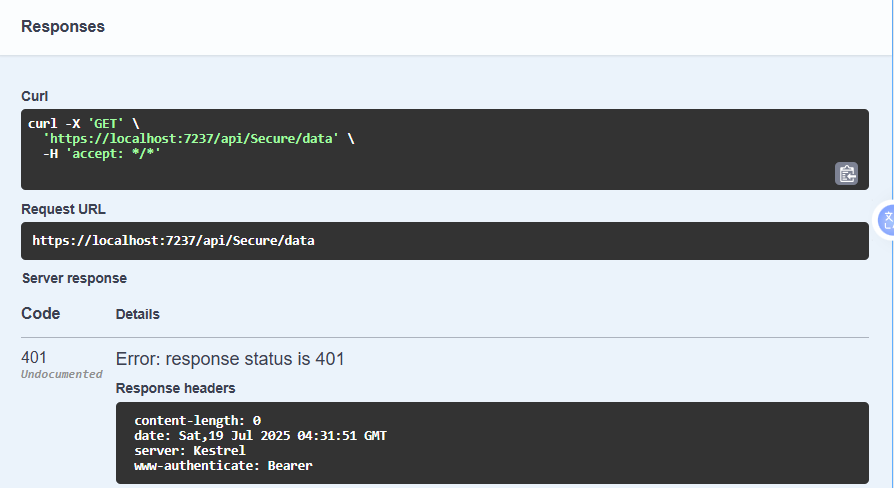
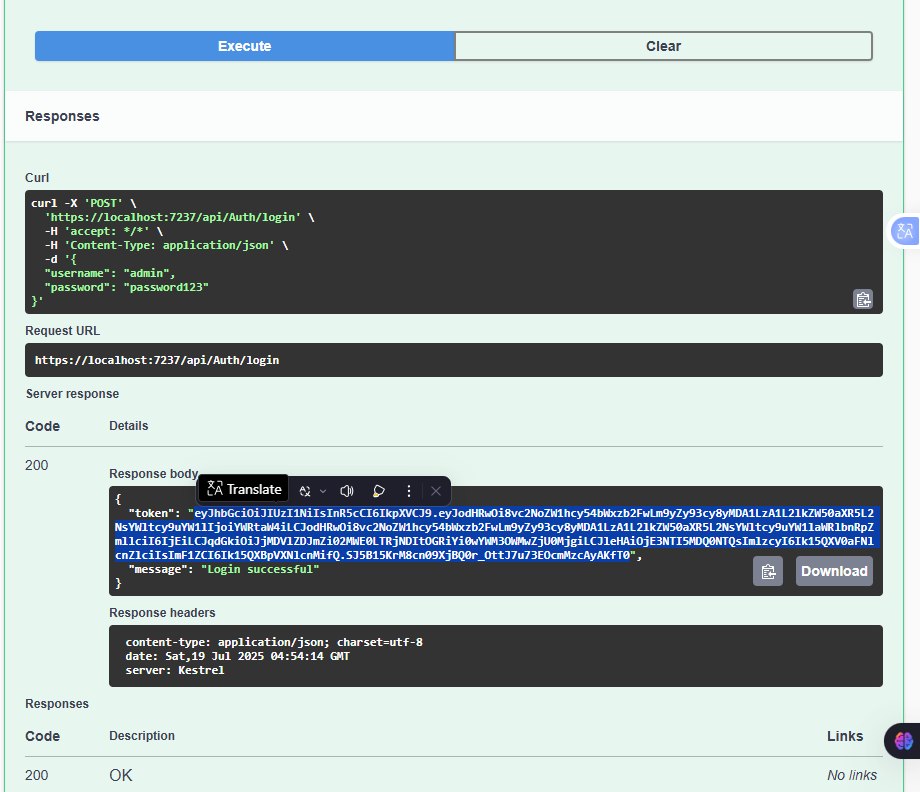
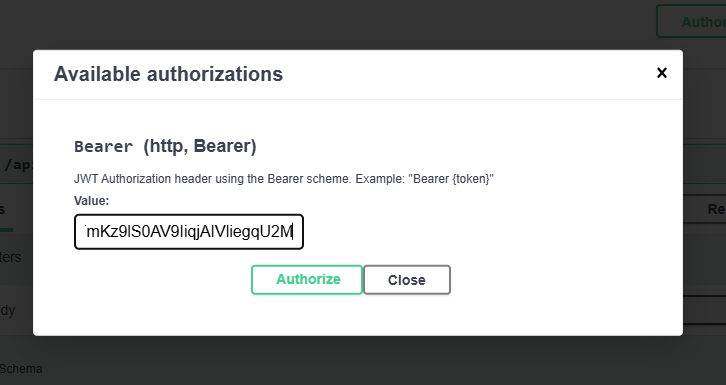
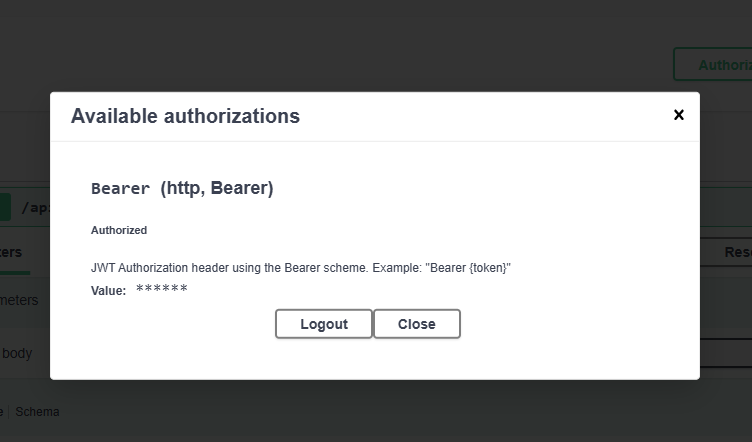
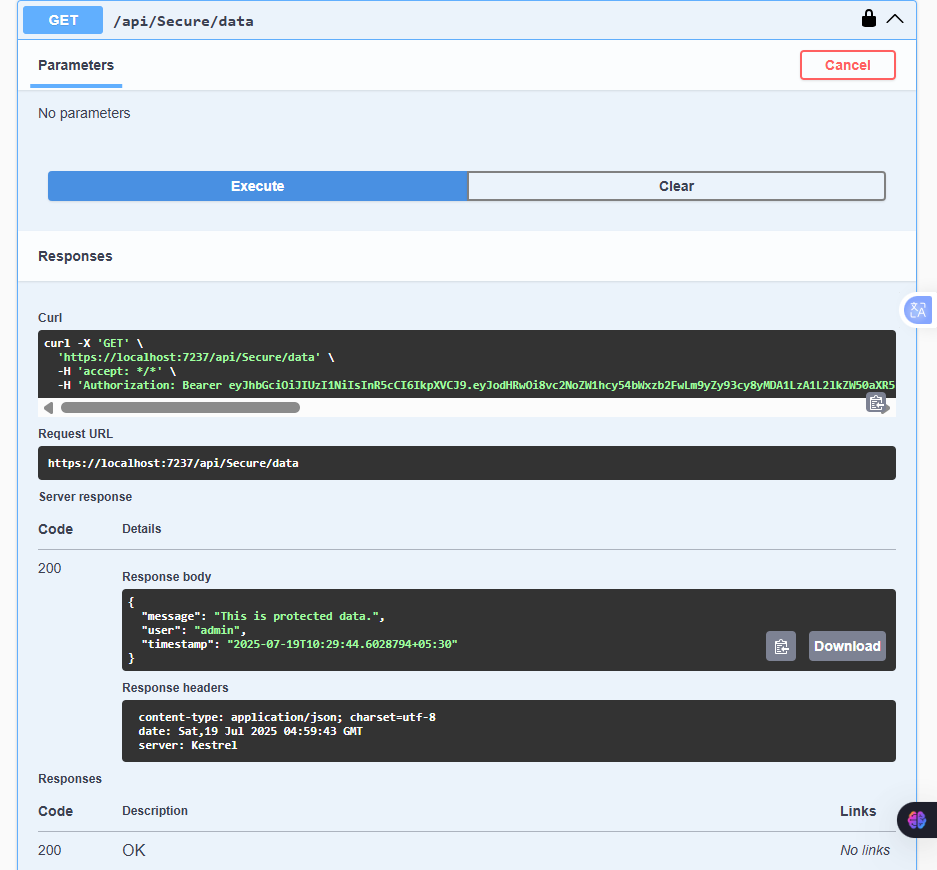
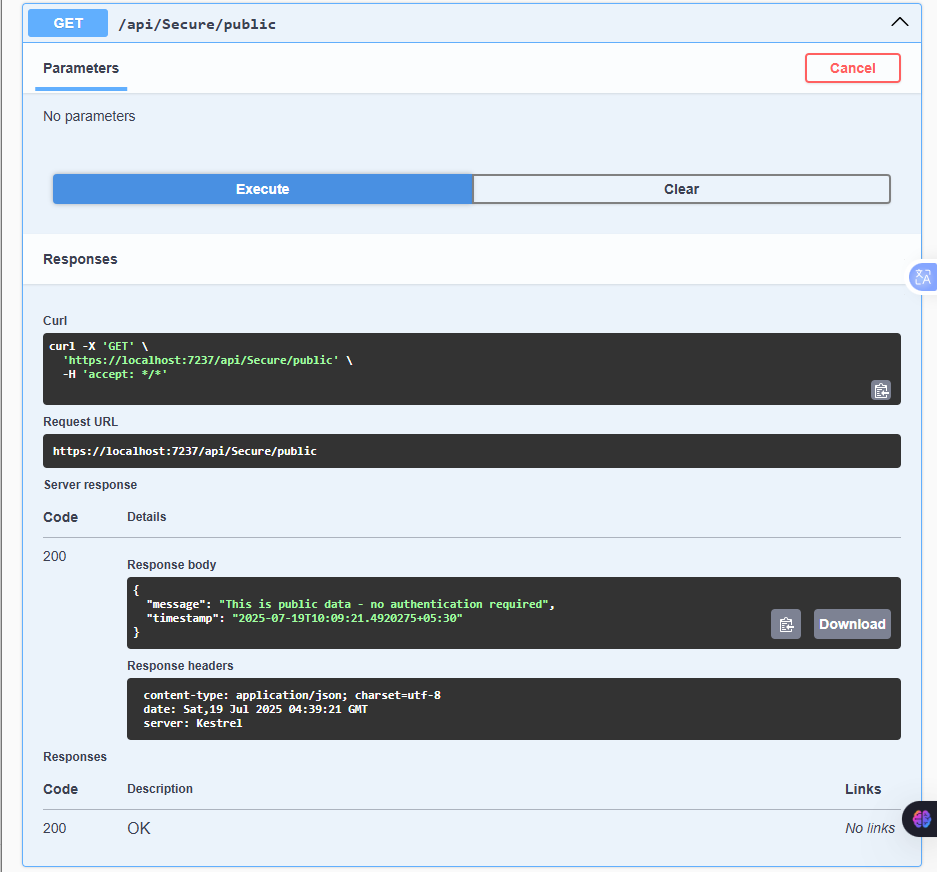
**Part 3: Assignment 2 – Secure Endpoints**

**Step 8: SecureController**

* Creating Controllers/SecureController.cs:

using Microsoft.AspNetCore.Authorization;  
using Microsoft.AspNetCore.Mvc;  
using System.Security.Claims;  
  
namespace JwtMicroservice.Controllers  
{  
 [ApiController]  
 [Route("api/[controller]")]  
 public class SecureController : ControllerBase  
 {  
 [HttpGet("data")]  
 [Authorize]  
 public IActionResult GetSecureData()  
 {  
 var username = User.FindFirst(ClaimTypes.Name)?.Value;  
 return Ok(new {   
 Message = "This is protected data.",  
 User = username,  
 Timestamp = DateTime.Now  
 });  
 }  
  
 [HttpGet("public")]  
 public IActionResult GetPublicData()  
 {  
 return Ok(new {   
 Message = "This is public data - no authentication required",  
 Timestamp = DateTime.Now  
 });  
 }  
 }  
}

**Step 9: Testing Secure Endpoints**

* Restarting the application.
* Going to Swagger.  
  
* Testing /api/Secure/data without a token:  
    
  **Execution Result:**   
  
* Logging via Auth endpoint,  
  
* using the token to authorize, and retrying:  
    
  **  
  Execution REUSLT for /api/Secure/data with a Bearer token:**  
  
* Checking if works without a token /api/Secure/public  
  

**Part 4: Assignment 3 – Role-Based Authorization**

**Step 10: Updating AuthController for Roles**

Replacing the relevant methods in AuthController.cs:

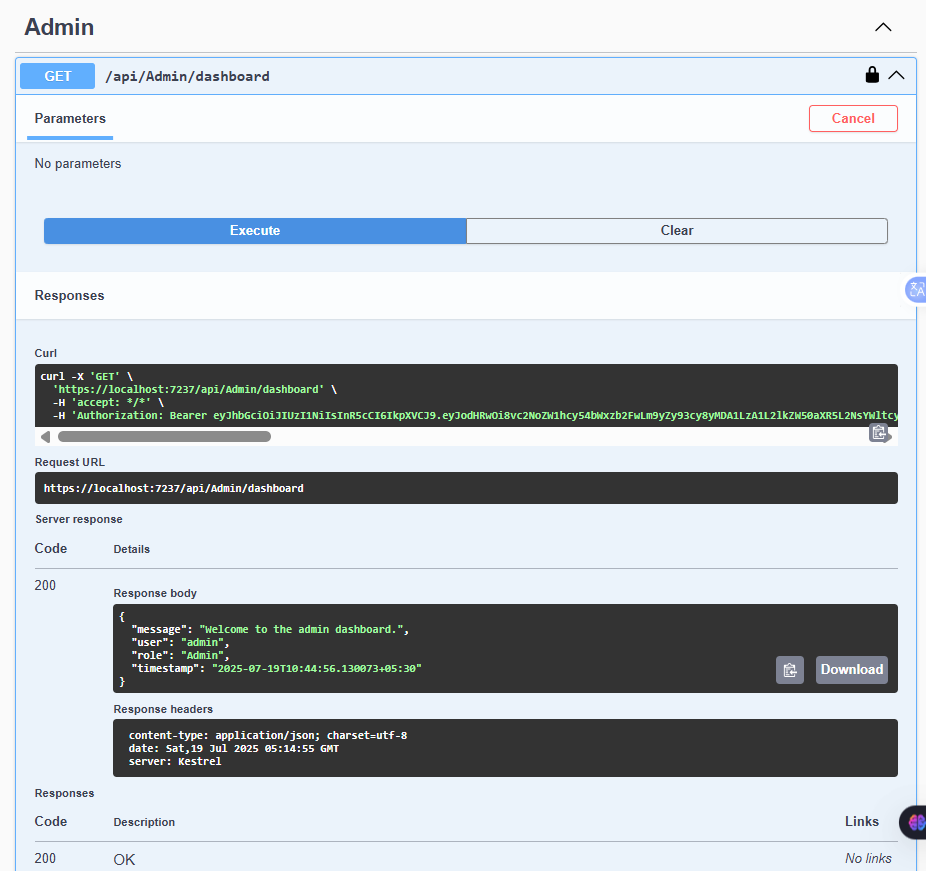
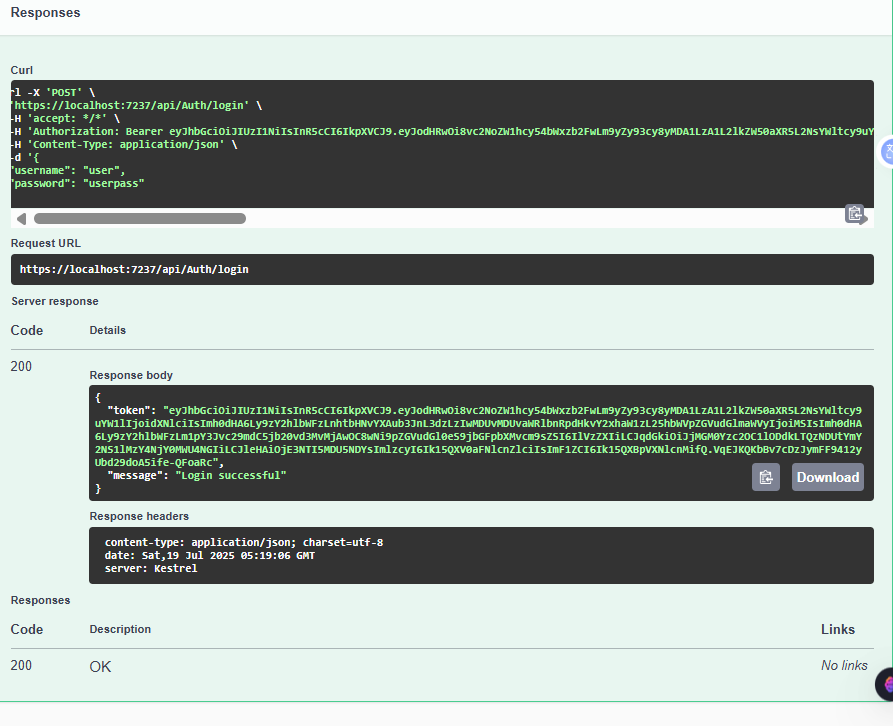
private bool IsValidUser(LoginModel model)  
{  
 var validUsers = new Dictionary<string, string>  
 {  
 { "admin", "password123" },  
 { "user", "userpass" }  
 };  
   
 return validUsers.ContainsKey(model.Username) &&   
 validUsers[model.Username] == model.Password;  
}  
  
private string GenerateJwtToken(string username)  
{  
 var role = GetUserRole(username);  
 var claims = new[]  
 {  
 new Claim(ClaimTypes.Name, username),  
 new Claim(ClaimTypes.NameIdentifier, "1"),  
 new Claim(ClaimTypes.Role, role),  
 new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())  
 };  
  
 var key = new SymmetricSecurityKey(  
 Encoding.UTF8.GetBytes(\_configuration["Jwt:Key"]!));  
 var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);  
  
 var token = new JwtSecurityToken(  
 issuer: \_configuration["Jwt:Issuer"],  
 audience: \_configuration["Jwt:Audience"],  
 claims: claims,  
 expires: DateTime.Now.AddMinutes(  
 Convert.ToDouble(\_configuration["Jwt:DurationInMinutes"])),  
 signingCredentials: creds);  
  
 return new JwtSecurityTokenHandler().WriteToken(token);  
}  
  
private string GetUserRole(string username)  
{  
 return username switch  
 {  
 "admin" => "Admin",  
 "user" => "User",  
 \_ => "User"  
 };  
}

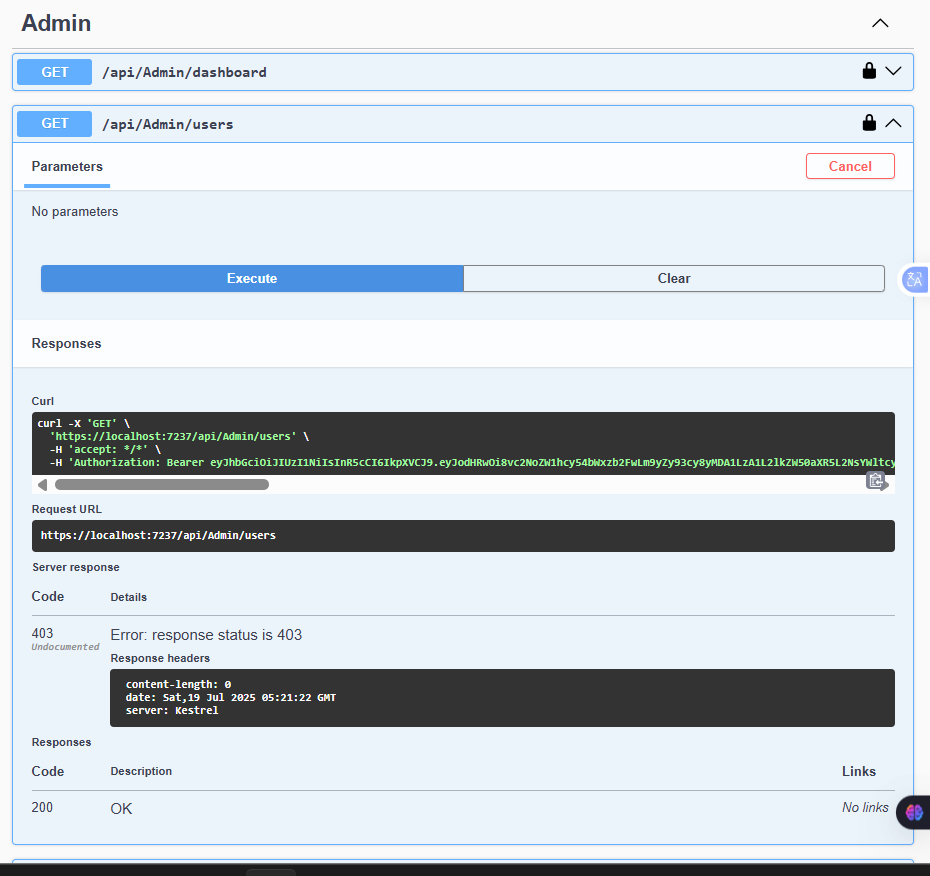
**Step 11: AdminController**

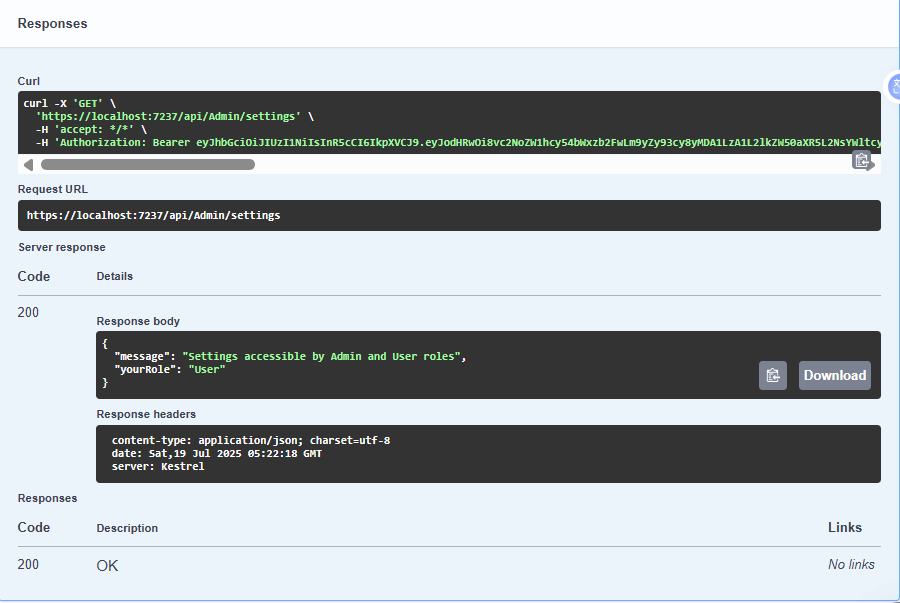
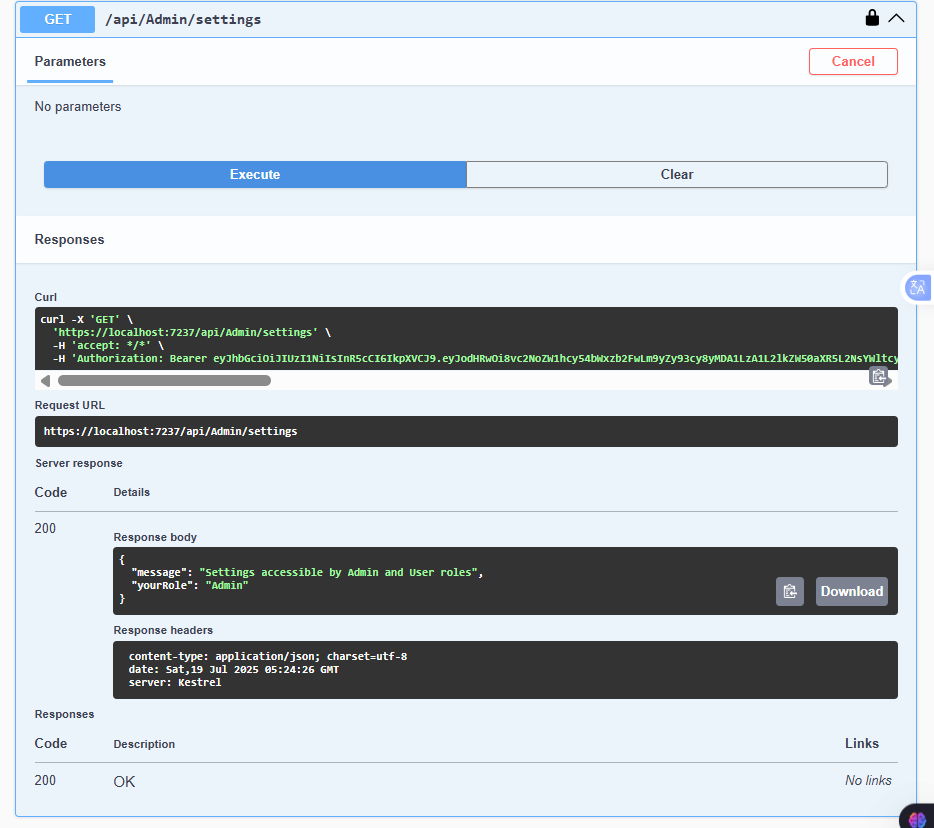
Creating Controllers/AdminController.cs:

using Microsoft.AspNetCore.Authorization;  
using Microsoft.AspNetCore.Mvc;  
using System.Security.Claims;  
  
namespace JwtMicroservice.Controllers  
{  
 [ApiController]  
 [Route("api/[controller]")]  
 public class AdminController : ControllerBase  
 {  
 [HttpGet("dashboard")]  
 [Authorize(Roles = "Admin")]  
 public IActionResult GetAdminDashboard()  
 {  
 var username = User.FindFirst(ClaimTypes.Name)?.Value;  
 var role = User.FindFirst(ClaimTypes.Role)?.Value;  
   
 return Ok(new {   
 Message = "Welcome to the admin dashboard.",  
 User = username,  
 Role = role,  
 Timestamp = DateTime.Now  
 });  
 }  
  
 [HttpGet("users")]  
 [Authorize(Roles = "Admin")]  
 public IActionResult GetAllUsers()  
 {  
 return Ok(new {   
 Message = "List of all users (Admin only)",  
 Users = new[] { "admin", "user1", "user2" }  
 });  
 }  
  
 [HttpGet("settings")]  
 [Authorize(Roles = "Admin,User")]  
 public IActionResult GetSettings()  
 {  
 var role = User.FindFirst(ClaimTypes.Role)?.Value;  
 return Ok(new {   
 Message = "Settings accessible by Admin and User roles",  
 YourRole = role  
 });  
 }  
 }  
}

**Step 12: Testing Role Authorization**

* Logging in as **admin**: access /api/Admin/dashboard .  
  
* Logging in as **user**: access /api/Admin/dashboard (**should get 403 Forbidden**).  
  **Execution Result:**   
  



* Both roles can access /api/Admin/settings.  
  **Execution RESULT:**
* **User Role**
* **Admin Role**

**Part 5: Assignment 4 – Token Expiry Handling**

**Step 13: Enhanced Error Handling (Program.cs)**

Updating JWT authentication code:

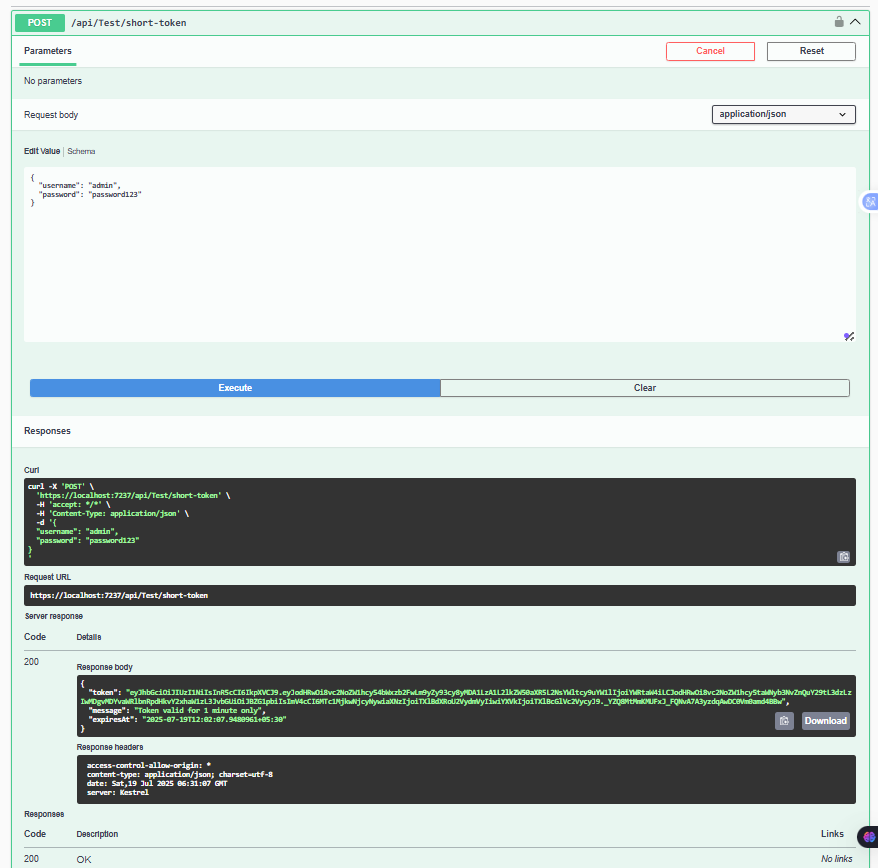
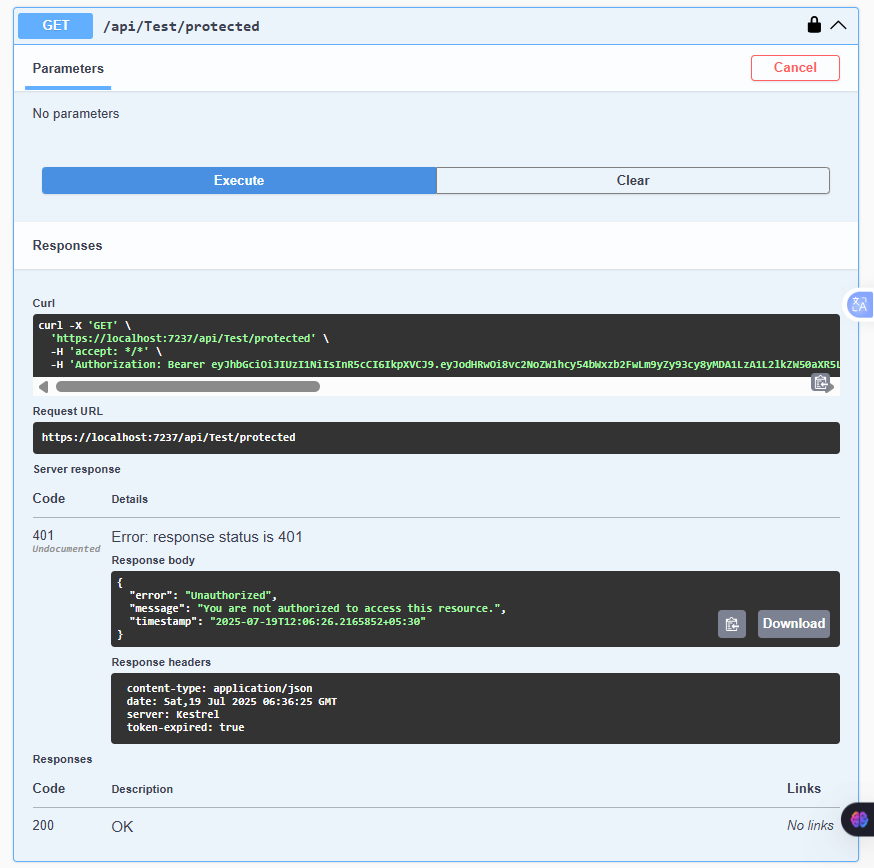
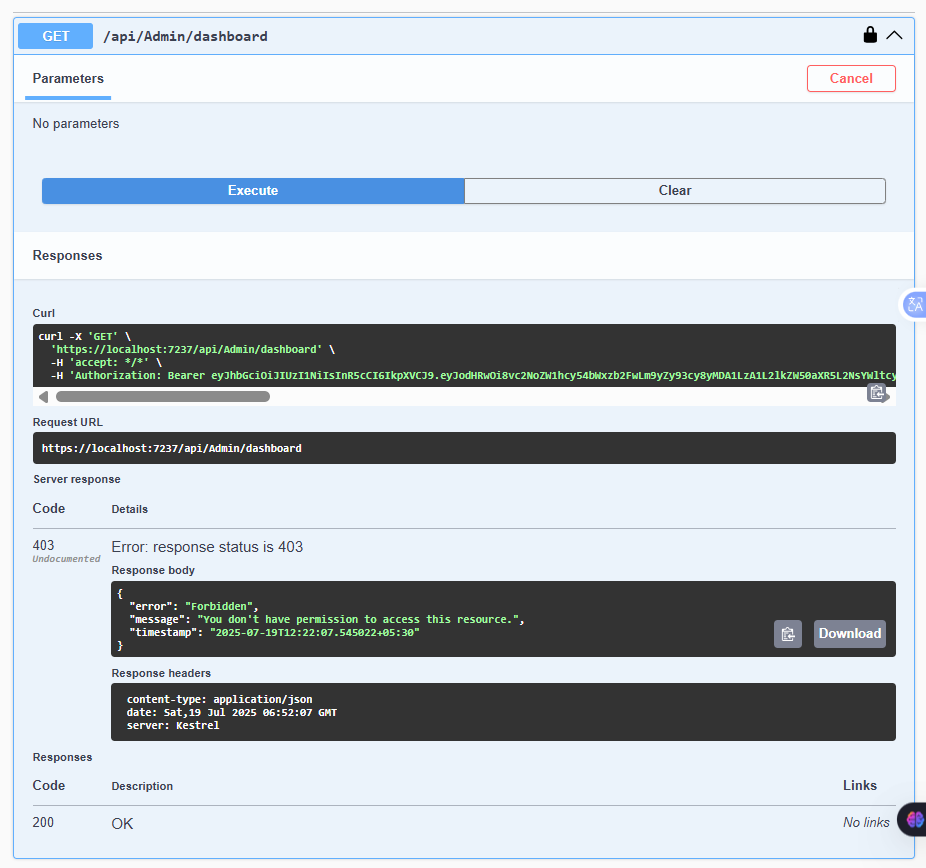
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)  
 .AddJwtBearer(options =>  
 {  
 options.TokenValidationParameters = new TokenValidationParameters  
 {  
 ValidateIssuer = true,  
 ValidateAudience = true,  
 ValidateLifetime = true,  
 ValidateIssuerSigningKey = true,  
 ValidIssuer = builder.Configuration["Jwt:Issuer"],  
 ValidAudience = builder.Configuration["Jwt:Audience"],  
 IssuerSigningKey = new SymmetricSecurityKey(  
 Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]!)),  
 ClockSkew = TimeSpan.Zero  
 };  
  
 options.Events = new JwtBearerEvents  
 {  
 OnAuthenticationFailed = context =>  
 {  
 if (context.Exception.GetType() == typeof(SecurityTokenExpiredException))  
 {  
 context.Response.Headers.Add("Token-Expired", "true");  
 }  
 return Task.CompletedTask;  
 },  
 OnChallenge = context =>  
 {  
 context.HandleResponse();  
 context.Response.StatusCode = 401;  
 context.Response.ContentType = "application/json";  
   
 var result = System.Text.Json.JsonSerializer.Serialize(new  
 {  
 error = "Unauthorized",  
 message = "You are not authorized to access this resource.",  
 timestamp = DateTime.Now  
 });  
   
 return context.Response.WriteAsync(result);  
 },  
 OnForbidden = context =>  
 {  
 context.Response.StatusCode = 403;  
 context.Response.ContentType = "application/json";  
   
 var result = System.Text.Json.JsonSerializer.Serialize(new  
 {  
 error = "Forbidden",  
 message = "You don't have permission to access this resource.",  
 timestamp = DateTime.Now  
 });  
   
 return context.Response.WriteAsync(result);  
 }  
 };  
 });

**Step 14: TestController for Token Expiry**

Adding Controllers/TestController.cs:

using Microsoft.AspNetCore.Authorization;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.IdentityModel.Tokens;  
using System.IdentityModel.Tokens.Jwt;  
using System.Security.Claims;  
using System.Text;  
using JwtMicroservice.Models;  
  
namespace JwtMicroservice.Controllers  
{  
 [ApiController]  
 [Route("api/[controller]")]  
 public class TestController : ControllerBase  
 {  
 private readonly IConfiguration \_configuration;  
  
 public TestController(IConfiguration configuration)  
 {  
 \_configuration = configuration;  
 }  
  
 [HttpPost("short-token")]  
 public IActionResult GenerateShortToken([FromBody] LoginModel model)  
 {  
 if (model.Username == "admin" && model.Password == "password123")  
 {  
 var token = GenerateShortJwtToken(model.Username);  
 return Ok(new {   
 Token = token,   
 Message = "Token valid for 1 minute only",  
 ExpiresAt = DateTime.Now.AddMinutes(1)  
 });  
 }  
 return Unauthorized();  
 }  
  
 [HttpGet("protected")]  
 [Authorize]  
 public IActionResult GetProtectedData()  
 {  
 return Ok(new {   
 Message = "Access granted!",  
 User = User.FindFirst(ClaimTypes.Name)?.Value,  
 Timestamp = DateTime.Now  
 });  
 }  
  
 private string GenerateShortJwtToken(string username)  
 {  
 var claims = new[]  
 {  
 new Claim(ClaimTypes.Name, username),  
 new Claim(ClaimTypes.Role, "Admin")  
 };  
  
 var key = new SymmetricSecurityKey(  
 Encoding.UTF8.GetBytes(\_configuration["Jwt:Key"]!));  
 var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);  
  
 var token = new JwtSecurityToken(  
 issuer: \_configuration["Jwt:Issuer"],  
 audience: \_configuration["Jwt:Audience"],  
 claims: claims,  
 expires: DateTime.Now.AddMinutes(1),  
 signingCredentials: creds);  
  
 return new JwtSecurityTokenHandler().WriteToken(token);  
 }  
 }  
}

**Step 16: Testing Expiry Handling**

* Using POST /api/Test/short-token, getting the token, and letting it expire.  
    
    
  After Expiration:
* Using this expired token at /api/Test/protected: should receive a custom 401 response with a message and timestamp. **RESULT:**   
  
* Attempting forbidden role access as user at GET /api/Admin/dashboard: seeing a custom 403 message.  
  

**CONCLUSION:   
  
Final Testing Scenarios**

* **Public access:** /api/Secure/public – no token needed.
* **Login as admin:** /api/Auth/login – get token.
* **Access protected data:** /api/Secure/data – token required.
* **Admin dashboard:** /api/Admin/dashboard – admin token required.
* **Login as user:** /api/Auth/login – get user token.
* **Try admin dashboard as user:** should get 403 Forbidden.
* **Access settings:** /api/Admin/settings – both admin and user can access.
* **Token expiry test:** generate a short-lived token and test expiry handling.

This contains all four assignments: basic JWT, secure endpoints, role-based authorization, and token expiry handling.