**WEB API**

**1. First Web API using .NET Core:**

### Step 1: Install .NET SDK

Make sure you have the .NET SDK installed. You can download it from the official [.NET website](https://dotnet.microsoft.com/download).

### Step 2: Create a New Web API Project

Open your terminal or command prompt and run the following command:

dotnet new webapi -n MyFirstApi

This will create a new directory named MyFirstApi with a Web API template.

### Step 3: Navigate to the Project Directory

Change your directory to the newly created project:

cd MyFirstApi

### Step 4: Create a Controller with Read/Write Permissions

Inside the Controllers folder, create a new controller named ValuesController.cs. You can use the following code:

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

namespace MyFirstApi.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ValuesController : ControllerBase

{

private static readonly List<string> Values = new List<string> { "Value1", "Value2" };

// GET: api/values

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

return Values;

}

// POST: api/values

[HttpPost]

public ActionResult Post([FromBody] string value)

{

Values.Add(value);

return CreatedAtAction(nameof(Get), new { id = Values.Count - 1 }, value);

}

}

}

### Step 5: Run the Application

Run the application using the following command:

dotnet run

### Step 6: Test the GET Action Method

You can test the GET action method by navigating to the following URL in your web browser or using a tool like Postman:

http://localhost:5000/api/values

You should see a response similar to:

["Value1", "Value2"]

### Step 7: Test the POST Action Method

You can test the POST action method using Postman or any API testing tool by sending a POST request to:

http://localhost:5000/api/values

With a JSON body like:

"Value3"

Received a response indicating that the new value has been added.

**2.Web Api using .Net core with Swagger**

To set up a .NET Core Web API with Swagger and test it as described, follow these steps:

### Step 1: Create or Use Existing Web API Project

If you haven't created a project yet, follow these commands to create a new Web API project:

dotnet new webapi -n MyEmployeeApi

cd MyEmployeeApi

### Step 2: Install Swashbuckle.AspNetCore Package

Install the Swashbuckle package for Swagger support:

dotnet add package Swashbuckle.AspNetCore

### Step 3: Modify Startup.cs

#### ConfigureServices Method

Open Startup.cs and locate the ConfigureServices method. Add the following code to register Swagger:

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo

{

Title = "Swagger Demo",

Version = "v1",

Description = "TBD",

TermsOfService = null,

Contact = new Microsoft.OpenApi.Models.OpenApiContact()

{

Name = "John Doe",

Email = "john@xyzmail.com",

Url = new Uri("http://www.example.com")

},

License = new Microsoft.OpenApi.Models.OpenApiLicense()

{

Name = "License Terms",

Url = new Uri("http://www.example.com")

}

});

});

}

#### Configure Method

In the same Startup.cs, locate the Configure method and add the following code to enable Swagger:

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseAuthorization();

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "Swagger Demo");

});

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

### Step 4: Run the Application

Execute the application with:

dotnet run

### Step 5: Access Swagger UI

Open a browser and navigate to:

https://localhost:[port number]/swagger

Replace [port number] with the actual port number displayed in your console.

* You should see the Swagger UI with the title "Swagger Demo" and the contact details.
* The Values controller's HTTP verbs should be listed.

### Step 6: Test the GET Method in Swagger

Click on the GET method for the Values controller, then click Try it out, followed by Execute. You should see the response in the panel below.

### Step 7: Test GET Method with Postman

1. Open Postman.
2. Set the request type to GET and enter the URL:

http://localhost:[port number]/api/values

1. Click Send.

* Verify that the list of values is displayed in the response body.
* Check the status code on the right side (it should be 200 OK).

### Step 8: Modify Employee Controller Route

If you have an EmployeeController, change its route attribute as follows:

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

// Change to:

[Route("api/Emp")]

}

### Step 9: Test New Route with Postman

1. In Postman, change the URL to:

http://localhost:[port number]/api/Emp

1. Click Send and verify the response.

**3.Web API using custom model class**

To implement a Web API with a custom model class, custom action filters, and exception handling in .NET Core, follow these steps:

### Step 1: Create the Employee Model Class

First, create a custom class named Employee in a new folder called Models.

// Models/Employee.cs

using System;

using System.Collections.Generic;

public class Employee

{

public int Id { get; set; }

public string Name { get; set; }

public int Salary { get; set; }

public bool Permanent { get; set; }

public Department Department { get; set; }

public List<Skill> Skills { get; set; }

public DateTime DateOfBirth { get; set; }

}

public class Department

{

public string Name { get; set; }

}

public class Skill

{

public string Name { get; set; }

}

### Step 2: Create EmployeeController

Create a new controller named EmployeeController in the Controllers folder.

// Controllers/EmployeeController.cs

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

[ApiController]

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

private readonly List<Employee> employees;

public EmployeeController()

{

employees = GetStandardEmployeeList();

}

private List<Employee> GetStandardEmployeeList()

{

return new List<Employee>

{

new Employee { Id = 1, Name = "John Doe", Salary = 50000, Permanent = true,

Department = new Department { Name = "IT" },

Skills = new List<Skill> { new Skill { Name = "C#" }, new Skill { Name = "SQL" } },

DateOfBirth = new DateTime(1990, 1, 1) },

new Employee { Id = 2, Name = "Jane Smith", Salary = 60000, Permanent = false,

Department = new Department { Name = "HR" },

Skills = new List<Skill> { new Skill { Name = "Recruitment" } },

DateOfBirth = new DateTime(1985, 5, 5) }

};

}

[HttpGet]

[ProducesResponseType(200)]

public ActionResult<List<Employee>> Get()

{

return Ok(employees);

}

[HttpPost]

public ActionResult<Employee> Post([FromBody] Employee employee)

{

employees.Add(employee);

return CreatedAtAction(nameof(Get), new { id = employee.Id }, employee);

}

[HttpPut("{id}")]

public ActionResult<Employee> Put(int id, [FromBody] Employee employee)

{

var existingEmployee = employees.Find(e => e.Id == id);

if (existingEmployee == null)

{

return NotFound();

}

existingEmployee.Name = employee.Name;

existingEmployee.Salary = employee.Salary;

existingEmployee.Permanent = employee.Permanent;

existingEmployee.Department = employee.Department;

existingEmployee.Skills = employee.Skills;

existingEmployee.DateOfBirth = employee.DateOfBirth;

return NoContent();

}

[HttpGet("standard")]

[ProducesResponseType(500)]

public ActionResult<List<Employee>> GetStandard()

{

throw new System.Exception("Sample exception for testing.");

}

}

### Step 3: Create Custom Authorization Filter

Create a folder named Filters and add a custom action filter.

// Filters/CustomAuthFilter.cs

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

public class CustomAuthFilter : ActionFilterAttribute

{

public override void OnActionExecuting(ActionExecutingContext context)

{

if (!context.HttpContext.Request.Headers.TryGetValue("Authorization", out var authHeader))

{

context.Result = new BadRequestObjectResult("Invalid request - No Auth token");

return;

}

if (!authHeader.ToString().StartsWith("Bearer "))

{

context.Result = new BadRequestObjectResult("Invalid request - Token present but Bearer unavailable");

}

}

}

### Step 4: Apply the Custom Authorization Filter

Apply the filter to the EmployeeController.

[ServiceFilter(typeof(CustomAuthFilter))]

[ApiController]

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

// Existing code...

}

### Step 5: Create Custom Exception Filter

Create a custom exception filter to handle exceptions globally.

// Filters/CustomExceptionFilter.cs

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

using System.IO;

public class CustomExceptionFilter : IExceptionFilter

{

public void OnException(ExceptionContext context)

{

var exceptionDetail = context.Exception.ToString();

File.AppendAllText("exceptions.log", exceptionDetail); // Log to a file

context.Result = new ObjectResult("An error occurred: " + exceptionDetail)

{

StatusCode = 500

};

}

}

### Step 6: Register Filters in Startup.cs

In Startup.cs, register the filters.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers(options =>

{

options.Filters.Add<CustomExceptionFilter>();

});

services.AddScoped<CustomAuthFilter>();

services.AddSwaggerGen();

}

### Step 7: Test with Swagger

1. Run the application.
2. Access Swagger UI at https://localhost:[port number]/swagger.
3. Test the GET method of the EmployeeController and check the status codes.
4. Test the authorization filter by sending requests without an Authorization header.

### Step 8: Install WebApiCompatShim

dotnet add package WebApiCompatShim

**5.Web API CRUD Operations:**

To implement a complete CRUD operation for updating employee data via a Web API in .NET Core, follow these steps. This will include validating the employee ID, updating the data, and returning the updated employee information.

### Step 1: Update the EmployeeController

Modify the EmployeeController to handle the PUT action method for updating employee data. The method will validate the ID and update the employee information accordingly.

// Controllers/EmployeeController.cs

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

[ApiController]

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

private readonly List<Employee> employees;

public EmployeeController()

{

employees = GetStandardEmployeeList();

}

private List<Employee> GetStandardEmployeeList()

{

return new List<Employee>

{

new Employee { Id = 1, Name = "John Doe", Salary = 50000, Permanent = true,

Department = new Department { Name = "IT" },

Skills = new List<Skill> { new Skill { Name = "C#" }, new Skill { Name = "SQL" } },

DateOfBirth = new DateTime(1990, 1, 1) },

new Employee { Id = 2, Name = "Jane Smith", Salary = 60000, Permanent = false,

Department = new Department { Name = "HR" },

Skills = new List<Skill> { new Skill { Name = "Recruitment" } },

DateOfBirth = new DateTime(1985, 5, 5) }

};

}

[HttpPut("{id}")]

[ProducesResponseType(typeof(Employee), 200)]

[ProducesResponseType(400)]

public ActionResult<Employee> Put(int id, [FromBody] Employee updatedEmployee)

{

if (id <= 0)

{

return BadRequest("Invalid employee id");

}

var existingEmployee = employees.FirstOrDefault(e => e.Id == id);

if (existingEmployee == null)

{

return BadRequest("Invalid employee id");

}

// Update the employee information

existingEmployee.Name = updatedEmployee.Name;

existingEmployee.Salary = updatedEmployee.Salary;

existingEmployee.Permanent = updatedEmployee.Permanent;

existingEmployee.Department = updatedEmployee.Department;

existingEmployee.Skills = updatedEmployee.Skills;

existingEmployee.DateOfBirth = updatedEmployee.DateOfBirth;

return Ok(existingEmployee); // Return the updated employee

}

}

### Step 2: Test the PUT Method Using Swagger

1. Run your application.
2. Open the Swagger UI at https://localhost:[port number]/swagger.
3. Navigate to the PUT method for the EmployeeController.
4. Click on the Try it out button.
5. Provide the employee ID in the path and the updated employee JSON in the request body, for example:

{

"Name": "John Doe Updated",

"Salary": 55000,

"Permanent": true,

"Department": {

"Name": "IT"

},

"Skills": [

{

"Name": "C#"

},

{

"Name": "ASP.NET"

}

],

"DateOfBirth": "1990-01-01T00:00:00"

}

1. Click Execute.

### Step 3: Verify the Response

* If you enter an invalid ID (less than or equal to 0), you should receive a 400 Bad Request response with the message "Invalid employee id".
* If you enter a valid ID that does not exist in the list, you should also receive the same 400 Bad Request response.
* If the ID is valid and the update is successful, you should see the updated employee data returned with a 200 OK status.

**5.JSON Web Token:**

To implement JWT authentication in your .NET Core Web API, follow these steps to create an AuthController, set up JWT generation, and configure the EmployeeController for authorization based on roles.

### Step 1: Configure JWT Authentication in Startup.cs

Modify the Startup.cs file to set up JWT authentication.

#### In the ConfigureServices Method

Add the following code to set up JWT authentication:

using Microsoft.IdentityModel.Tokens;

using System.Text;

public void ConfigureServices(IServiceCollection services)

{

string securityKey = "mysuperdupersecret";

var symmetricSecurityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(securityKey));

services.AddAuthentication(x =>

{

x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, x =>

{

x.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = "mySystem",

ValidAudience = "myUsers",

IssuerSigningKey = symmetricSecurityKey

};

});

services.AddControllers();

}

#### In the Configure Method

Enable authentication by adding the following line:

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

// Other middleware...

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

### Step 2: Create AuthController

Create a new controller named AuthController and implement the JWT generation.

using Microsoft.AspNetCore.Mvc;

using Microsoft.IdentityModel.Tokens;

using System.Collections.Generic;

using System.Security.Claims;

using System.Text;

[AllowAnonymous]

[ApiController]

[Route("api/[controller]")]

public class AuthController : ControllerBase

{

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(10),

signingCredentials: credentials);

return new JwtSecurityTokenHandler().WriteToken(token);

}

[HttpGet]

public ActionResult<string> GetToken()

{

var token = GenerateJSONWebToken(1, "Admin");

return Ok(token);

}

}

### Step 3: Update EmployeeController

Add the [Authorize] attribute to the EmployeeController to enforce JWT authentication.

using Microsoft.AspNetCore.Authorization;

[Authorize]

[ApiController]

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

private readonly List<Employee> employees;

public EmployeeController()

{

employees = GetStandardEmployeeList();

}

// Other methods...

[HttpGet]

[ProducesResponseType(200)]

[ProducesResponseType(401)]

public ActionResult<List<Employee>> Get()

{

return Ok(employees);

}

}

### Step 4: Test in Postman

**Generate Token:**

* 1. Send a GET request to http://localhost:[port number]/api/auth to retrieve the JWT.
  2. Copy the token from the response.

**Access Employee API:**

Send a GET request to http://localhost:[port number]/api/employee without the token in the Authorization header. You should receive a 401 Unauthorized response.

Now send the same GET request, but this time include the token in the Authorization header as follows:

Authorization: Bearer <your\_token>

**Test Invalid Token:**

* 1. Modify the token in Postman and send the request again. You should see a 401 Unauthorized status.

### Step 5: Check JWT Expiration

### ****Modify Token Expiration:**** Change the expires attribute in GenerateJSONWebToken to 2 minutes:

expires: DateTime.Now.AddMinutes(2),

**Test Expiration:**

Generate a new token and send a GET request to http://localhost:[port number]/api/employee immediately.

Wait for 2 minutes and send the same request again. You should receive a 401 Unauthorized response.

### Step 6: Implement Role-Based Authorization

**Modify the** EmployeeController **Authorization:**  
Adjust the Authorize attribute in EmployeeController to check for roles:

[Authorize(Roles = "POC")]

[ApiController]

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

// Other methods...

}

**Test Unauthorized Role:**

Generate a token with the role of Admin and try to access the EmployeeController. You should receive a 401 Unauthorized response.

**Allow Multiple Roles:**  
Modify the Authorize attribute to allow both Admin and POC:

[Authorize(Roles = "Admin,POC")]

**Final Testing:**

Test accessing the EmployeeController with a token that has the Admin role and verify you receive a 200 OK response.

**ANALYSIS:**

# Analysis of JWT Authentication Implementation in .NET Core Web API

## Introduction

The implementation of JSON Web Token (JWT) authentication in a .NET Core Web API is a robust solution for securing API endpoints. This analysis outlines the steps taken to set up JWT authentication, create an authorization mechanism, and handle role-based access control.

## Overview of JWT Authentication

JWT is a compact, URL-safe means of representing claims to be transferred between two parties. It allows for secure communication and ensures that data can be verified and trusted. By passing a token in the Authorization header, the server can validate the token to authenticate requests.

### Key Components

1. **Security Key**: A symmetric key is used to sign the tokens, ensuring that they cannot be tampered with.
2. **Claims**: These are pieces of information about the user, such as roles and user IDs, included in the token.
3. **Token Expiration**: Tokens are issued with an expiration time to enhance security.

## Implementation Steps

### 1. Configuration in Startup.cs

The first step involved configuring the JWT authentication in the Startup.cs file. This included setting up the authentication scheme, validation parameters, and ensuring that the application uses authentication middleware.

* **Security Key**: A symmetric security key is defined and used for signing tokens.
* **Token Validation Parameters**: These parameters dictate how the token should be validated, including issuer, audience, and signing key.

### 2. Creating AuthController

A new AuthController was created to handle the generation of JWTs.

* **Token Generation**: The method GenerateJSONWebToken was implemented to create a token containing user-specific claims and set for a defined expiration time.
* **Anonymous Access**: The AllowAnonymous attribute allows unauthenticated users to request a token.

### 3. Authorization in EmployeeController

The EmployeeController was updated to require JWT authentication.

* **Authorize Attribute**: The controller uses the [Authorize] attribute to enforce token validation for its endpoints.
* **Role-Based Access**: By specifying roles within the Authorize attribute, access can be controlled based on user roles defined in claims.

### 4. Testing with Postman

Testing was conducted using Postman to ensure the functionality of the JWT authentication.

* **Token Retrieval**: A GET request was sent to the AuthController to retrieve a token.
* **Protected Endpoint Access**: Requests to the EmployeeController were tested both with valid and invalid tokens to confirm the expected responses (200 OK for valid tokens and 401 Unauthorized for invalid ones).
* **Expiration Handling**: The expiration time was set to 2 minutes, and tests confirmed that access was denied after the token expired.

### 5. Role-Based Authorization

The implementation included testing for role-based access control.

* **Unauthorized Access**: Attempts to access the EmployeeController with insufficient roles resulted in 401 Unauthorized responses.
* **Multi-Role Authorization**: The ability to specify multiple roles in the Authorize attribute was demonstrated, allowing access to users with either role.

## Conclusion

The implementation of JWT authentication in the .NET Core Web API demonstrated a comprehensive approach to securing API endpoints through token-based authentication. The steps taken not only provided a secure means of validating user requests but also allowed for granular control over access based on user roles.

This analysis highlights the importance of secure authentication methods in modern web applications, emphasizing JWT's effectiveness in ensuring both security and usability in API interactions. The successful integration of these features sets a strong foundation for further enhancements in API security and user management.