**1.WEBAPI\_HANDSON**

**Objectives:**

* **Explain the concept of RESTful web service, Web API & Microservice**
  + **Features of REST architecture - Representational State Transfer, Stateless, Messages, Concept of Microservice, Difference between WebService & WebAPI, Not restricted to send XML as response**

**Explanation :**

REST (Representational State Transfer):

* An architectural style for designing networked applications.
* Key features:
  + Stateless: Each request contains all the information needed to process it.
  + Representations: Resources are represented using JSON, XML, etc.
  + Messages: Usually HTTP requests/responses.
  + Uses HTTP methods as verbs (GET, POST, PUT, DELETE).

Web API:

* A framework for building HTTP-based services.
* Designed to expose data and services to a broad range of clients.
* Not restricted to XML – commonly uses JSON.

Microservice:

* A small, independently deployable service focused on a single business capability.
* Built using lightweight mechanisms like REST APIs.

Difference between WebService & WebAPI:

| Feature | WebService (SOAP) | Web API (REST) |
| --- | --- | --- |
| Protocol | Uses SOAP | Uses HTTP/HTTPS |
| Message Format | XML only | JSON, XML, etc. |
| Bandwidth | High | Low |
| Platform Dependency | Tightly coupled to WCF | Platform independent |
| Use in Browsers | Not directly supported | Easily consumable |

* **Explain what is HttpRequest & HttpResponse**

**Explanation:**

HttpRequest:

* Represents the incoming request to the server.
* Contains method (GET, POST), headers, body, URL, etc.

HttpResponse:

* Represents the response from the server.
* Includes status code, headers, and body (data or error message).
* **List the types of Action Verbs**
  + **HttpGet, HttpPost, HttpPut, HttpDelete - Meaning of action verbs and how that should be declared as attributes for Web API**

**Explanation:**

Action Verbs (HTTP Methods)

| Verb | Description | Usage in Web API Attribute |
| --- | --- | --- |
| HttpGet | Retrieves data (read) | [HttpGet] |
| HttpPost | Sends data (create) | [HttpPost] |
| HttpPut | Updates existing data | [HttpPut] |
| HttpDelete | Deletes specified data | [HttpDelete] |

* **List the types of HttpStatusCodes used in WebAPI**
  + **Ok, InternalServerError, Unauthorized, BadRequest - All thru the action result types**

**Explanation:**

HTTP Status Codes

| Status Code | Meaning | Usage in Web API |
| --- | --- | --- |
| 200 OK | Successful request | return Ok(data); |
| 400 Bad Request | Invalid input | return BadRequest(); |
| 401 Unauthorized | Authentication failed | return Unauthorized(); |
| 500 Internal Server Error | Server-side issue | return StatusCode(500); |

* **Demonstrate creation of a simple WebAPI - With Read, Write actions**
  + **Structure of a web api - Controller & its inheritance from ApiController, Action verbs, Action method**

Web API:

A Web API is a way for different software systems to communicate over HTTP.  
It exposes endpoints (URLs) that respond to HTTP requests like:

* GET → Read
* POST → Create
* PUT → Update
* DELETE → Delete

Controller:

* The heart of the Web API.
* In ASP.NET Core, the controller is a class that:
  + Inherits from ControllerBase or ApiController
  + Is decorated with the attribute [ApiController]
  + Defines routes using [Route("...")]
  + Contains action methods for handling requests.

Inheritance:

* Your controller class inherits from:
  + ControllerBase (used for Web APIs — no view support)
  + Controller (used for MVC — includes views)

[ApiController]

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

public class ProductController : ControllerBase

{

// Action methods go here

}

Action Verbs and Methods :

| HTTP Verb | Attribute | Purpose | Example |
| --- | --- | --- | --- |
| GET | [HttpGet] | Read data | GetAll(), GetById() |
| POST | [HttpPost] | Create data | Create() |
| PUT | [HttpPut] | Update data | Update() |
| DELETE | [HttpDelete] | Delete data | Delete() |

Model Class (Product.cs) :

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public double Price { get; set; }

Controller Class (ProductController.cs)

[ApiController]

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

public class ProductController : ControllerBase

{

// In-memory list (fake database)

private static List<Product> products = new List<Product>()

{

new Product { Id = 1, Name = "Pen", Price = 10 },

new Product { Id = 2, Name = "Notebook", Price = 50 }

};

// READ all products

[HttpGet]

public ActionResult<IEnumerable<Product>> Get()

{

return Ok(products); // returns 200 OK + data

}

// READ product by ID

[HttpGet("{id}")]

public ActionResult<Product> GetById(int id)

{

var product = products.FirstOrDefault(p => p.Id == id);

if (product == null)

return NotFound(); // returns 404

return Ok(product);

}

// WRITE (add new product)

[HttpPost]

public ActionResult<Product> Post(Product product)

{

product.Id = products.Max(p => p.Id) + 1;

products.Add(product);

return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);

}

}

Output Example

POST /api/product

Request body:

{

"name": "Marker",

"price": 20

}

Response:

201 Created

Location: /api/product/3

Body: { "id": 3, "name": "Marker", "price": 20 }

GET /api/product

Response:

[

{ "id": 1, "name": "Pen", "price": 10 },

{ "id": 2, "name": "Notebook", "price": 50 },

{ "id": 3, "name": "Marker", "price": 20 }

]

* **Explain the types of Configuration files of WebAPI**
  + **Startup.cs with depdency injection, appSettings.json, launchSettings.json, Explain Route.config & WebAPI.config in .Net 4.5 framework**

**Explanation:**

In .NET Core and .NET 6/7/8, configuration is centralized and file-based, with strong support for dependency injection, environment-based settings, and custom configuration.

Startup.cs *(used in .NET Core 3.1 and earlier)*

In .NET 6 and above, it's merged into Program.cs.

Purpose:

* Configures services and middleware.
* Sets up Dependency Injection (DI).
* Defines routing and pipelines.

public class Startup

{

public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

// Configure services (e.g., DI, DBContext)

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddScoped<IMyService, MyService>(); // DI

}

// Configure middleware pipeline

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers(); // Enables attribute routing

});

}

}

appsettings.json

Purpose:

* Stores application configuration like:
  + Connection strings
  + API keys
  + Logging settings
  + Custom config sections

{

"ConnectionStrings": {

"DefaultConnection": "Server=.;Database=MyDb;Trusted\_Connection=True;"

},

"AppSettings": {

"ApiKey": "123456"

},

"Logging": {

"LogLevel": {

"Default": "Information"

}

}

}

Accessing config in code:

public class MyService

{

private readonly string \_apiKey;

public MyService(IConfiguration configuration)

{

\_apiKey = configuration["AppSettings:ApiKey"];

}

}

launchSettings.json

Location: Properties/launchSettings.json

Purpose:

* Defines how the app is launched during development (not production).
* Sets up:
  + Profiles (IIS Express, Project)
  + URLs to launch
  + Environment variables

"profiles": {

"SimpleWebApi": {

"commandName": "Project",

"launchBrowser": true,

"applicationUrl": "https://localhost:5001;http://localhost:5000",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

}

}

Legacy: ASP.NET Web API in .NET Framework 4.5

Before .NET Core, configuration was done differently.

WebApiConfig.cs

Purpose:

* Used to define routing rules for Web API.

Location: App\_Start/WebApiConfig.cs

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Define default route for API

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

config.Formatters.Remove(config.Formatters.XmlFormatter); // Return JSON only

}

}

This file is registered in Global.asax.cs during app startup:

protected void Application\_Start()

{

GlobalConfiguration.Configure(WebApiConfig.Register);

}

RouteConfig.cs

Purpose:

* Handles routing for MVC (not Web API).
* Located in App\_Start/RouteConfig.cs

Sample Code:

public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

}

}

Web.config

Purpose:

* XML-based configuration file for .NET Framework apps.
* Stores:
  + App settings
  + Connection strings
  + Compilation/debug settings

Sample code:

<configuration>

<appSettings>

<add key="ApiKey" value="123456"/>

</appSettings>

<connectionStrings>

<add name="DefaultConnection" connectionString="..." providerName="System.Data.SqlClient"/>

</connectionStrings>

</configuration>

Access:

string key = ConfigurationManager.AppSettings["ApiKey"];

1. **First Web Api using .Net core**

Create a .Net core web application with API template. Use the option to create controller with Read Write permissions. Notice the ValuesController creation with Action methods corresponding to the Action verbs.

On creation of the Web API, execute the application and check if the GET action method result is returned as expected.

**Code :**

Values Controller:

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

namespace FirstWebApiDemo.Controllers

{

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

[ApiController]

public class ValuesController : ControllerBase

{

private static List<string> \_values = new List<string> { "value1", "value2" };

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

return \_values;

}

[HttpGet("{id}")]

public ActionResult<string> Get(int id)

{

if (id < 0 || id >= \_values.Count)

return NotFound();

return \_values[id];

}

[HttpPost]

public IActionResult Post([FromBody] string value)

{

\_values.Add(value);

return Ok();

}

[HttpPut("{id}")]

public IActionResult Put(int id, [FromBody] string value)

{

if (id < 0 || id >= \_values.Count)

return NotFound();

\_values[id] = value;

return NoContent();

}

[HttpDelete("{id}")]

public IActionResult Delete(int id)

{

if (id < 0 || id >= \_values.Count)

return NotFound();

\_values.RemoveAt(id);

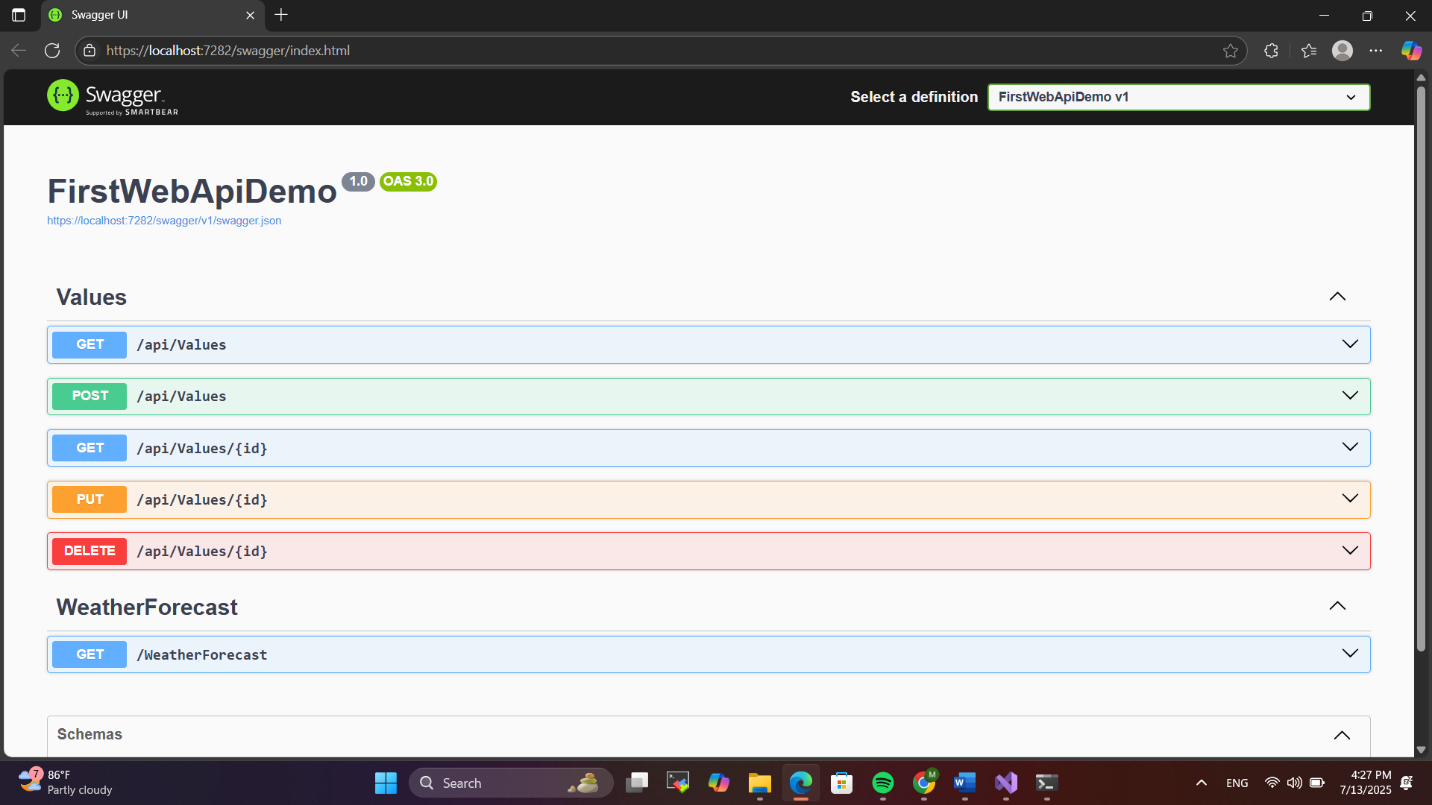
return NoContent();

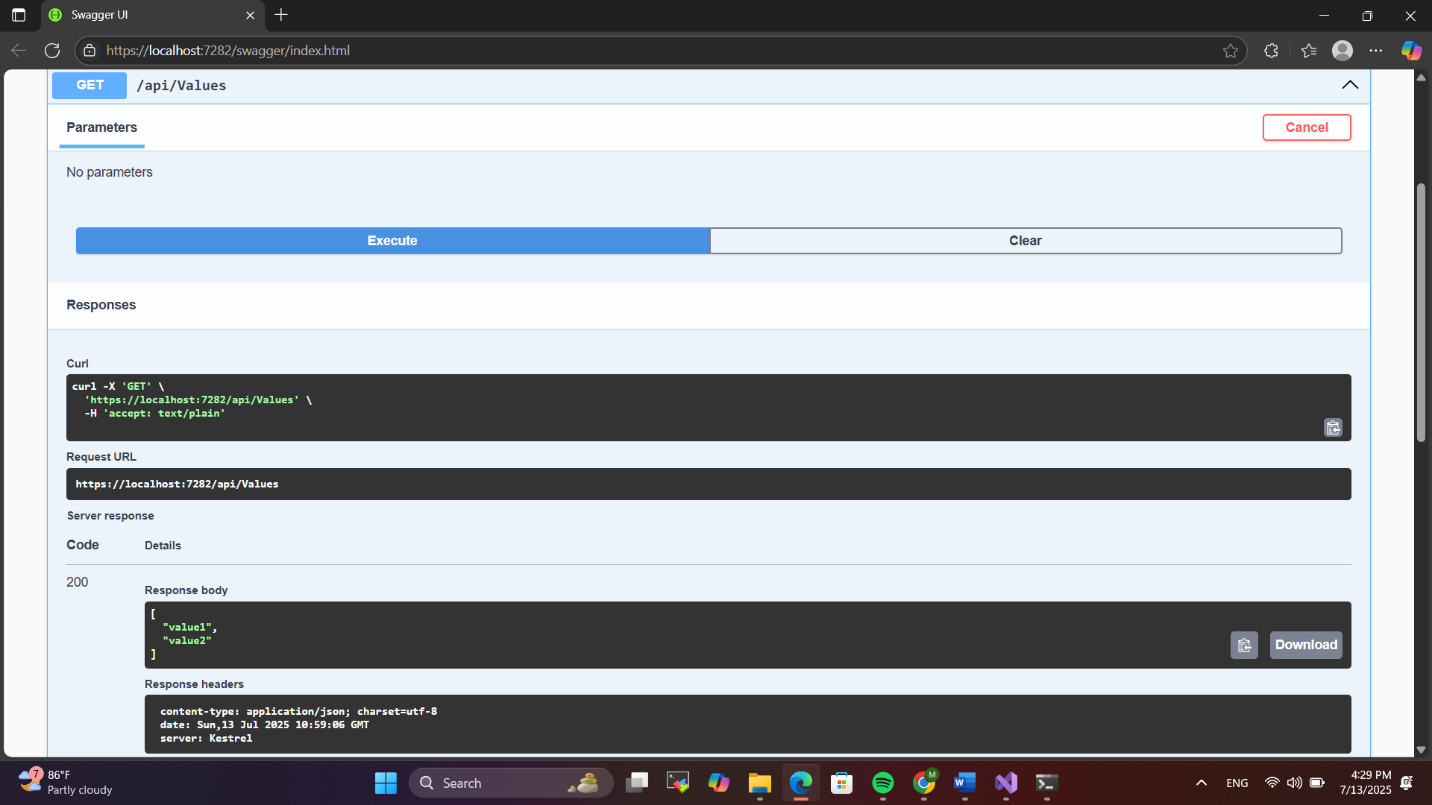
}

}

}

**Output :**

****

****

**2.WEBAPI\_HANDSON**

**Objectives:**

* **Demonstrate Swagger installation to WebAPI and WebAPI listing on browser**
  + **Nuget package to download Swashbuckle.AspNetCore, Usage of ProducesResponseType to Web API method, AddSwaggerGen, UseSwaggerUI**

To demonstrate Swagger installation in an ASP.NET Core Web API, you first install the Swashbuckle.AspNetCore NuGet package. This package automatically generates interactive API documentation. After installing it, you configure Swagger in Program.cs by calling builder.Services.AddSwaggerGen() to enable Swagger generation and then add app.UseSwagger() and app.UseSwaggerUI() in the middleware pipeline to serve the Swagger UI. Each Web API method can be documented using the ProducesResponseType attribute to specify possible response codes, such as 200 OK or 500 Internal Server Error. Once everything is configured, running the application allows you to navigate to the /swagger endpoint in the browser to see the list of all available Web API endpoints in a user-friendly format, where you can test them directly. This improves both documentation and ease of testing.

* **Demonstrate the usage of Postman tool to hit WebAPI methods**
  + **Structure in Postman tool, Headers with Authorization, Body as JSON, Option to choose the type of request, Request collection and how to add a new request in the collection, Tabs in the center pane that corresponds to the request**

Postman is a widely used tool for testing WebAPI methods. It provides an easy-to-use interface where users can select the type of HTTP request (GET, POST, PUT, DELETE) from a dropdown. In the center pane, users can view tabs such as Params, Authorization, Headers, Body, and Tests. To send authenticated requests, developers can add an Authorization header with a Bearer token. For POST or PUT requests, the body section is set to raw and the data is formatted as JSON. Users can create collections to organize multiple related requests, and new requests can be added to these collections by clicking the "New" button or right-clicking a collection. Postman helps developers inspect response status codes, response body, and headers, making it a powerful tool for debugging and testing Web APIs.

* **Demonstrate the usage of Route and Explain Name attribute in Http requests**
  + **Importance of user friendly name to action method, Explain the usage of ActionName to have more than 1 method with the same Action verbDemonstrate creation of a simple WebAPI - With Read, Write actions**

In ASP.NET Core WebAPI, the Route attribute is used to define the URL path through which an action method or controller can be accessed. It plays a crucial role in defining clean and user-friendly URLs. Additionally, the Name attribute in HTTP request attributes like [HttpGet(Name = "GetEmployee")] allows developers to assign a user-friendly and descriptive name to an action method, which can be useful when generating links or for documentation. The ActionName attribute is especially useful when there are multiple methods with the same HTTP verb, as it lets you differentiate between them by assigning unique names. For example, you can have two [HttpGet] methods where one is marked as [ActionName("GetById")] and the other as [ActionName("GetAll")], enabling route mapping without conflict. Demonstrating a simple WebAPI with Read and Write actions involves creating a controller with actions decorated with [HttpGet] and [HttpPost], where [HttpGet] is used to fetch data and [HttpPost] to add new data using a custom model class.

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

Create a .Net core web application with API template. (Use existing application if created). Install Swashbuckle.AspNetCore Nuget package. Post this do the following steps in Startup.cs

* In ConfigureServices method, add the code provided below.

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new Info

{

Title = "Swagger Demo",

Version = "v1",

Description = "TBD",

TermsOfService = "None",

Contact = new Contact() { Name = "John Doe", Email = "john@xyzmail.com", Url = "www.example.com" },

License = new License() { Name = "License Terms", Url = "www.example.com" }

});

});

* In Configure method, add the code provided below.

app.UseSwagger();

app.UseSwaggerUI(c =>

{

// specifying the Swagger JSON endpoint.

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

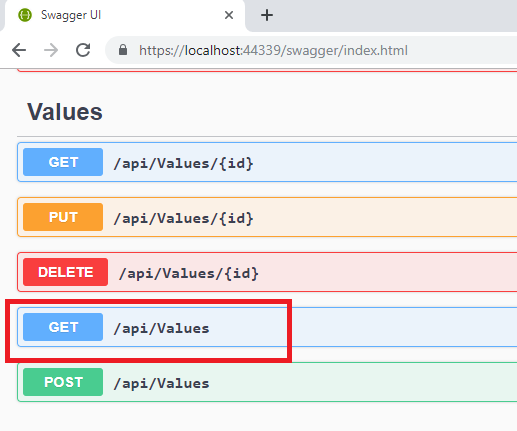
});

Execute the application which will load the default ‘Values’ controller(Settings as per launchSettings.json) GET action method. Change the url to <https://localhost:[port> number]/swagger

Notice the Title, Version, Contact detail provided shown on the top of the page

Notice the Values controller HttpVerb action methods getting listed.

Click the ‘GET’ action verb method(Without the parameter).



It opens a panel which has ‘Try it out’ button. Click that and Click ‘Execute’ button.

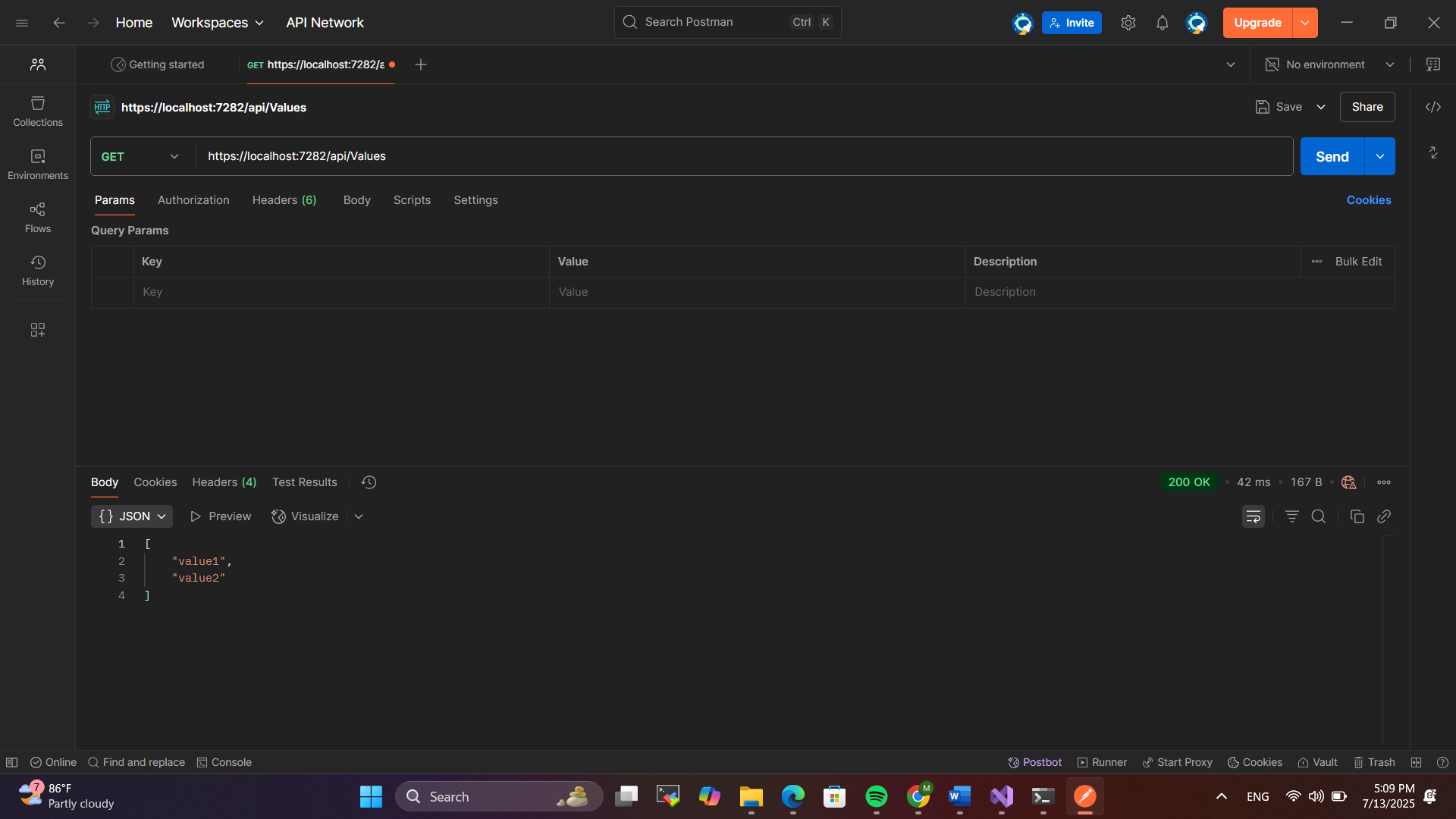
1. Use POSTMAN tool, to point to the local Web API that was created with Employee controller. Test the GET action method using POSTMAN.

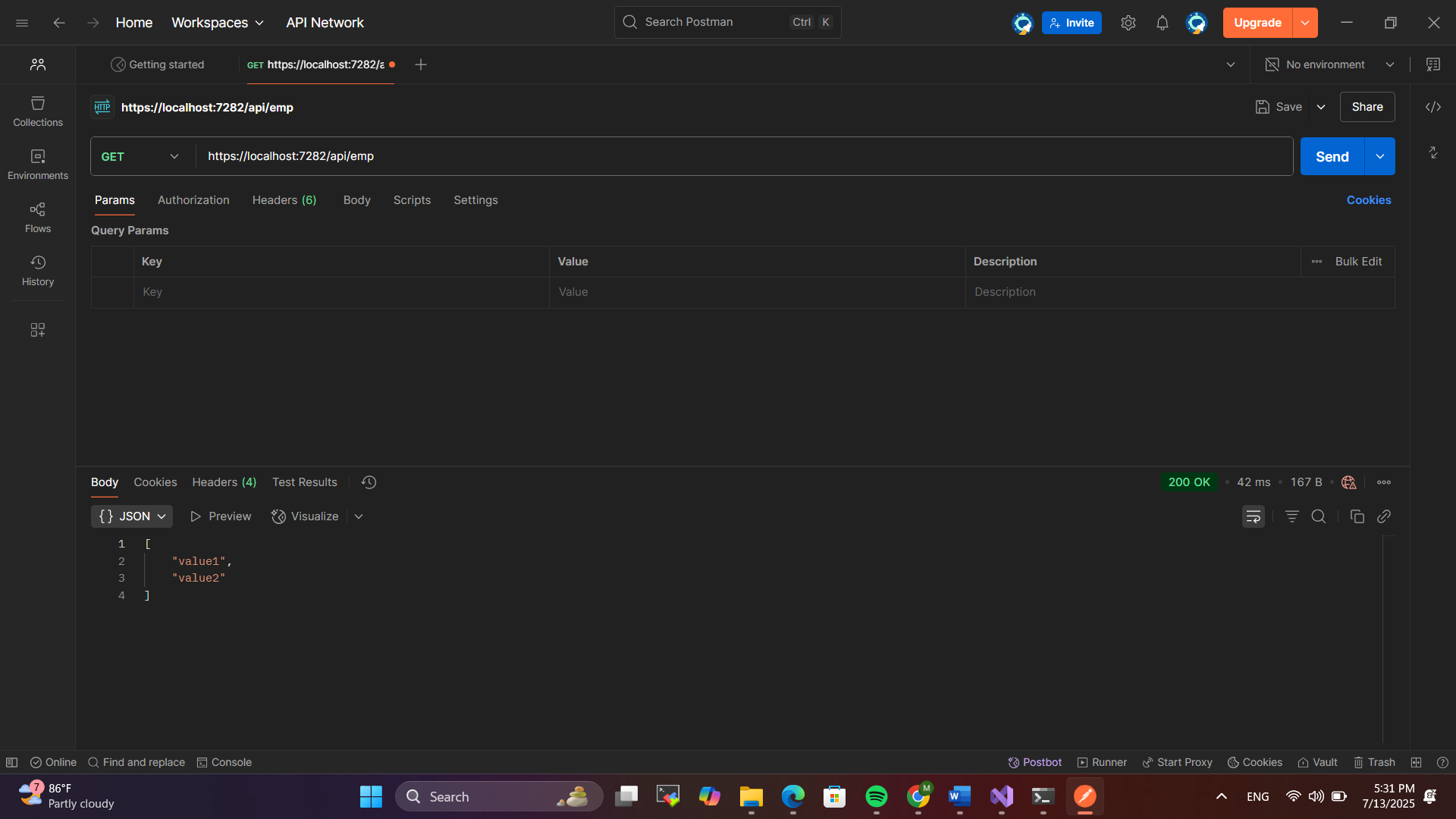
Verify the output if the List of employees are listed in the ‘Body’ part of the GET window on POSTMAN tool.

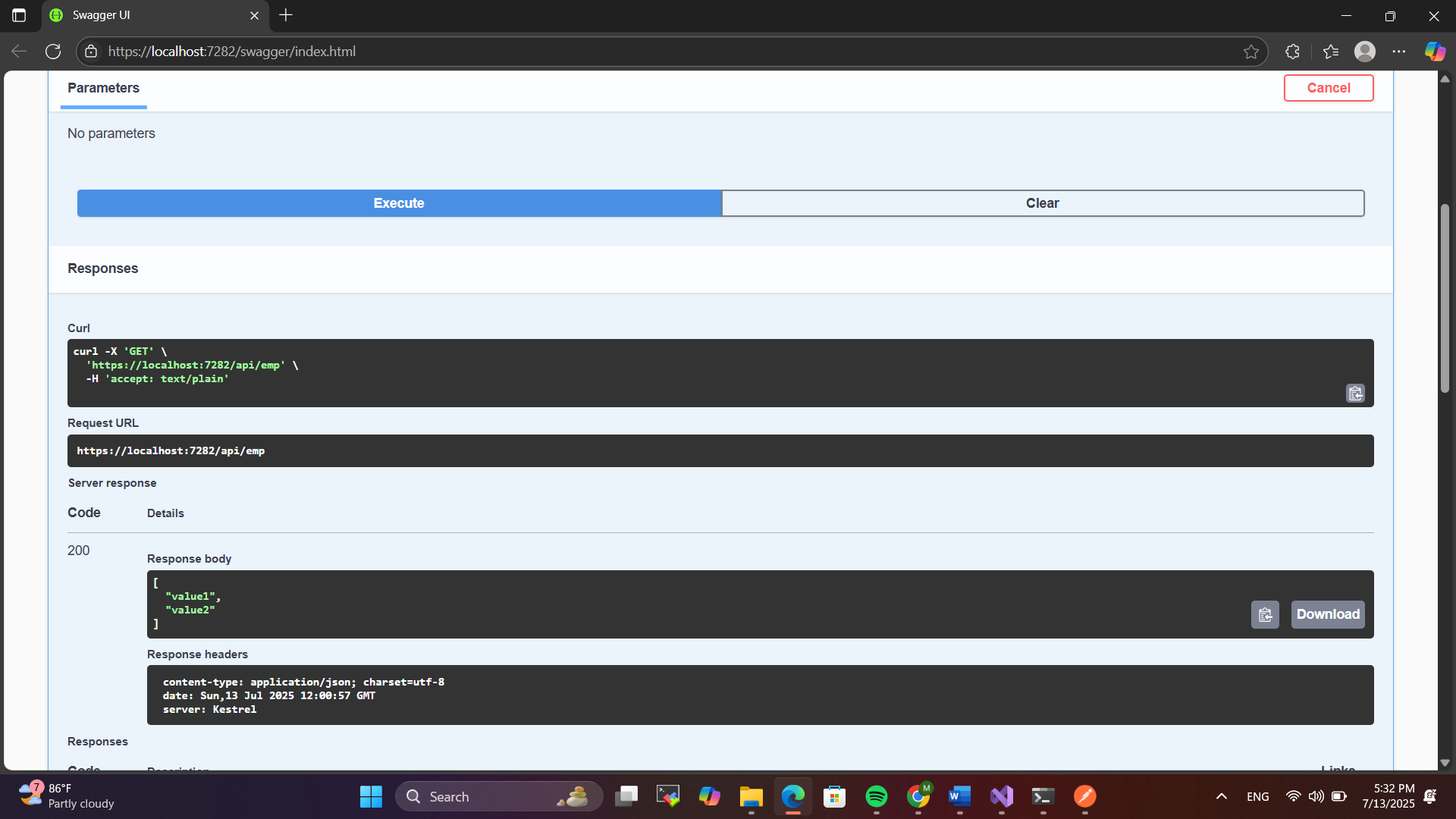
Verify the Status on the right side of the output pane on POSTMAN tool.

1. Modify the Controller name in the Route attribute of the Employee controller to ‘Emp’ and check its access thru POSTMAN

**Output :**







**3.WEBAPI\_HANDSON**

**Objectives:**

* **Demonstrate creation of an Action method to return list of custom class entity**
  + **Model class creation, Use AllowAnonymous attribute, Use HttpGet action method**

To return a list of custom class entities in a Web API, first create a model class—for example, an Employee class with properties like Id, Name, Salary, and other relevant fields. In the controller, define an action method with the [HttpGet] attribute to handle GET requests. Inside this method, create and return a hardcoded list of Employee objects. Use the [AllowAnonymous] attribute on the method or controller to allow access without authentication. The method returns the list wrapped in an ActionResult<List<Employee>> or IEnumerable<Employee>. This setup demonstrates how to expose structured data via an HTTP GET call, which can be tested using tools like Swagger or Postman to verify that the endpoint correctly returns the custom objects in JSON format.

* **Explain the usage of FromBody attribute**
  + **Read the model object from request, other than the query string parameter**

The FromBody attribute in ASP.NET Core Web API is used to bind a complex type parameter, like a custom model object, from the body of an HTTP request rather than from the query string or route. When a client sends data in the request body—typically in JSON format—FromBody helps deserialize that data into the corresponding C# model object. For example, in a POST or PUT action method, you can define a parameter like public IActionResult Post([FromBody] Employee emp) to extract the employee data from the request body. This is essential when dealing with large or structured data that doesn't fit well in query strings.

* **Demonstrate Custom filter** 
  + **Usage of ActionFilterAttribute, OnActionExecuting method to intercept the request, Create filter for Custom exception - Need to install Microsoft.AspNetCore.Mvc.WebApiCompatShim package**

Custom filters in ASP.NET Core Web API allow developers to execute code before or after certain stages in the request pipeline. A custom filter can be created by inheriting from ActionFilterAttribute and overriding the OnActionExecuting method to intercept and validate incoming requests before the action method executes. For example, a custom authentication filter can check if a specific header is present in the request. Similarly, custom exception filters can be created by implementing the IExceptionFilter interface to handle unhandled exceptions globally. These filters enhance security and maintainability by centralizing validation and error handling. To support legacy Web API compatibility when working with exception filters, the Microsoft.AspNetCore.Mvc.WebApiCompatShim NuGet package may be required. This approach helps maintain clean controller logic by offloading cross-cutting concerns like authorization or logging to reusable filters.

1. **Web Api using custom model class**

Create a Custom class ‘Employee’ of the below defined structure

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; }

}

Create a new controller - EmployeeController with Read Write actions

Constructor: Create few records, HTTPGet, HTTPPost/HTTPPut

Create a Private method GetStandardEmployeeList that returns a List of Employee class. Invoke this method in the Get action method of the EmployeeController that was created in the previous step.

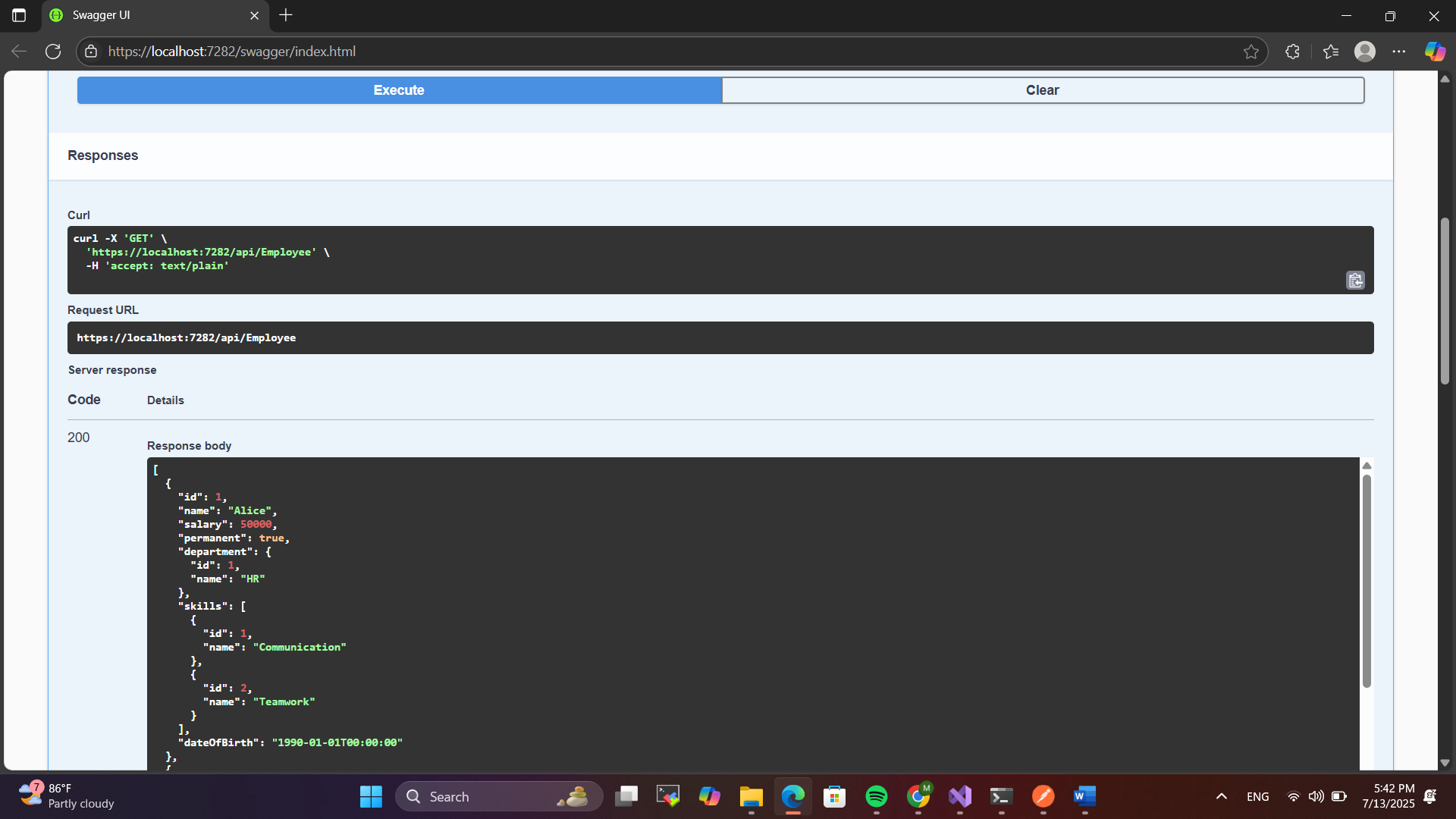
Public ActionResult<Employee> GetStandrad()

Modify the return type of the Get action method(without parameter) to return List of Employee class object

Add ProducesResponseType to the GET action method for Status code 200

Check the Swagger description for the GET method for success status code

**Output:**



1. **Create a Custom action filter for Authorization.**

The requirement is to intercept incoming requests and check if there is a key ‘Authorization’ in the request header or not. If it is there, then to check if it contains a value ‘Bearer’ or not.

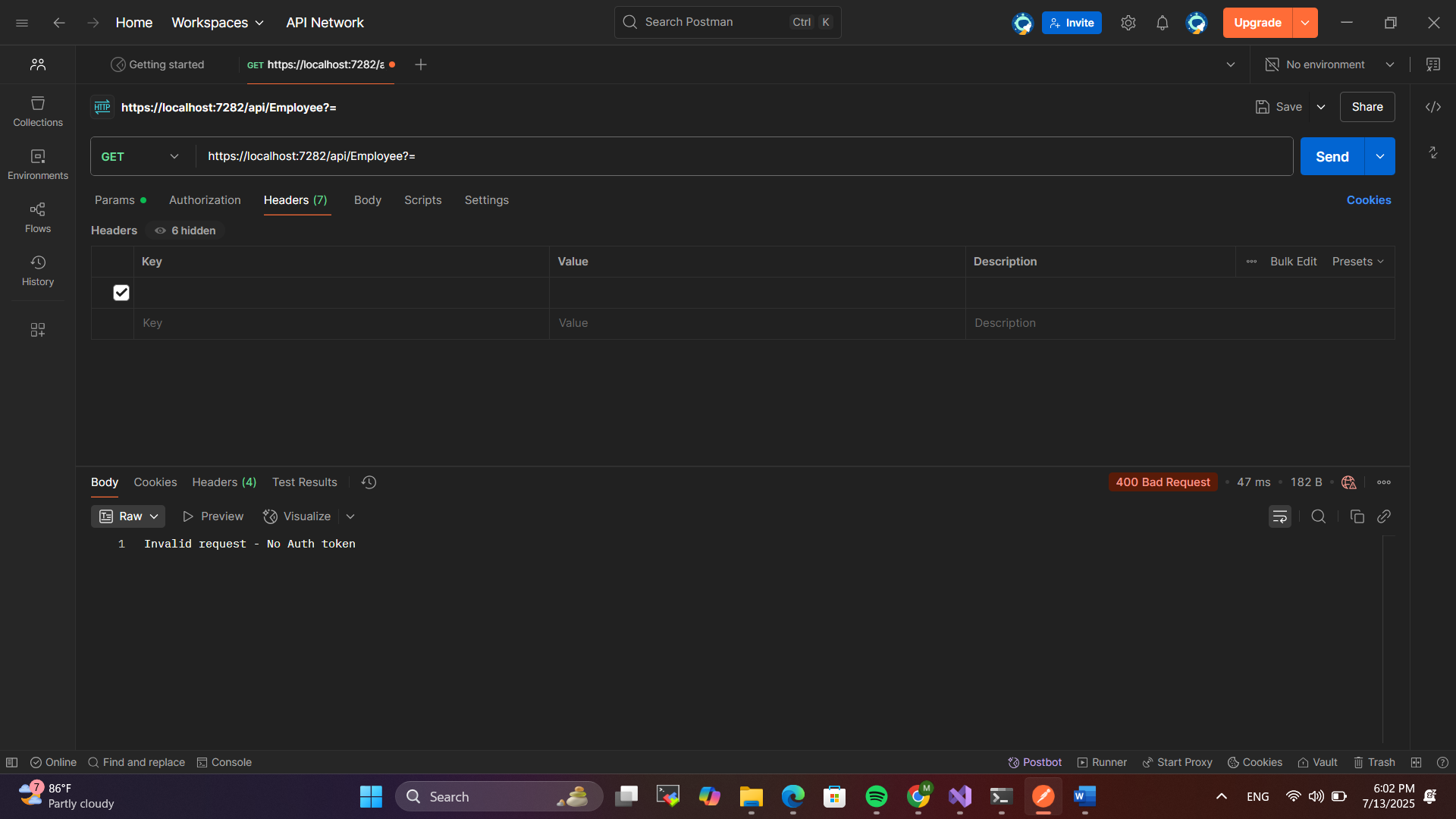
Create a folder ‘Filters’ in the application solution. Create a class ‘**CustomAuthFilter**’ to filter requests. Inherit ActionFilterAttribute. Override OnActionExecuting method to check if the request object has Header ‘Authorization’ or not. If not, throw BadRequestResult with the message

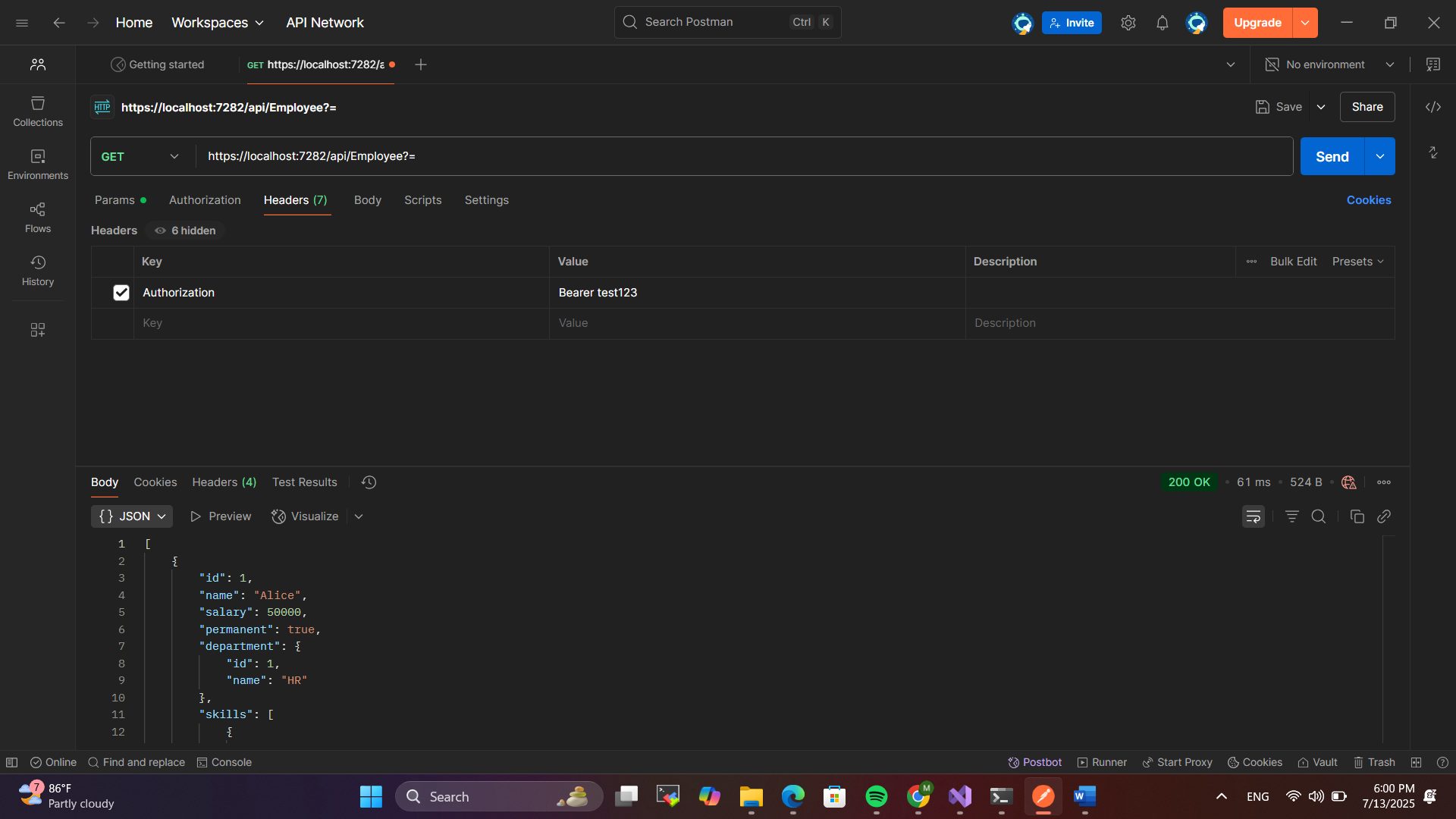
Invalid request - No Auth token

If the header is present, then check if the value contains the word ‘Bearer’. If not, throw BadRequestResult with the message

Invalid request - Token present but Bearer unavailable

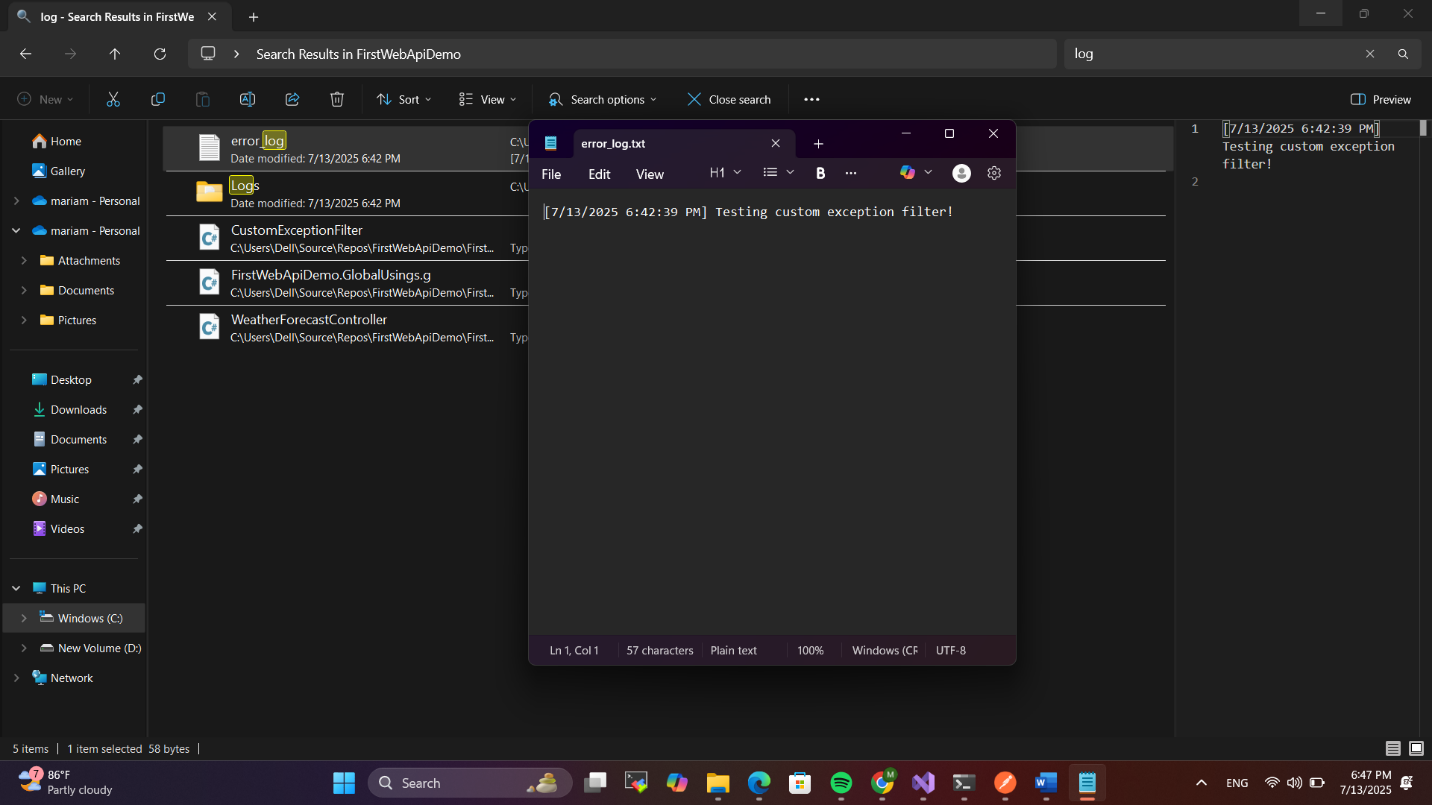
Add an attribute **CustomAuthFilter** to the Employee controller to filter any request to check for the Authorization token in the request header.

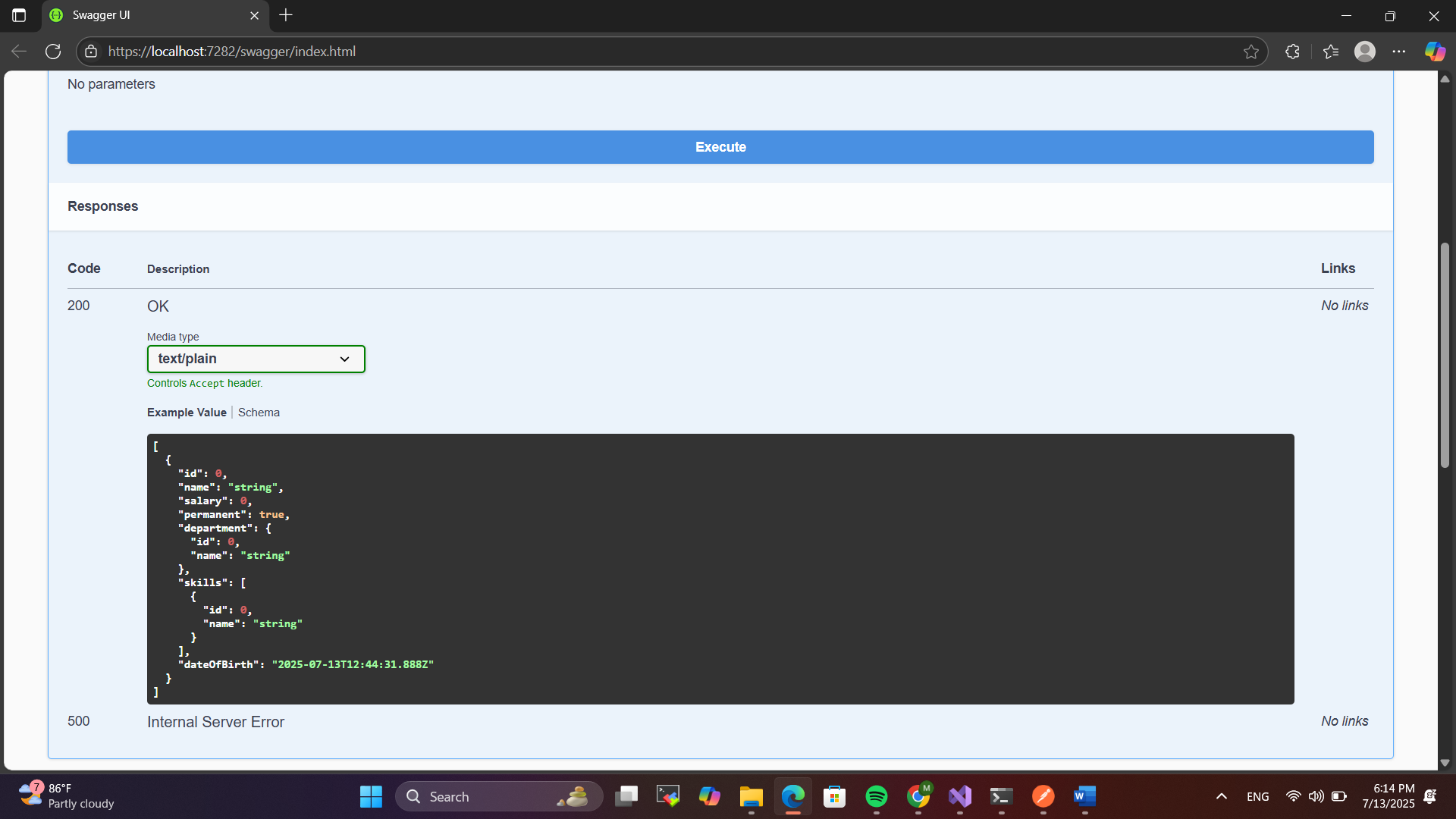




1. **Custom Exception filter**

Create a class ‘CustomExceptionFilter’ to catch the exceptions occuring the application. Implement IExceptionFilter thru the OnException method  
  
Use the exception context to fetch the exception detail. Capture that and write it to a File in the system.  
  
Set the Result property of the exception context to ExceptionResult.  
  
Throw an exception in GET action method.  
Ensure that the GET action method has ProducesResponseType for 500 - Internal server error  
  
Use Swagger to test the exception and message being thrown.  
  
Note: This needs WebApiCompatShim NuGet package installation

**Output:  
  
  
**



**4.WEBAPI\_HANDSON**

**Objectives:**

* **Demonstrate creation of an Action method to perform data create, update & delete operation**
  + **Use FromBody attribute, extract data to custom model class using FromBody attribute, use hardcoded data to update & delete data, Use Swagger and POSTMAN to test**

To demonstrate create, update, and delete operations in a Web API, action methods can be defined within a controller using HTTP verbs like POST, PUT, and DELETE. The [FromBody] attribute is used in method parameters to bind the incoming JSON data to a custom model class (e.g., Employee). For example, in a Post method, new employee data sent in the request body is deserialized into an Employee object and added to a hardcoded list. In the Put method, the list is searched by ID, and the matched object’s properties are updated with the new values. Similarly, in the Delete method, the employee with the specified ID is removed from the list. These methods return appropriate ActionResult types such as Ok, CreatedAtAction, or BadRequest. Swagger UI and Postman can be used to test these methods by sending HTTP requests with JSON bodies for POST and PUT, and URL path parameters for DELETE. This allows for easy validation of data manipulation through a simulated backend.

1. **Web Api CRUD operation**

Update Employee data as per the input thru Web API PUT action method call

Employee information has to be updated based on the user input. Use Swagger tool to invoke the action method mapped with Http PUT action verb to update an employee data.

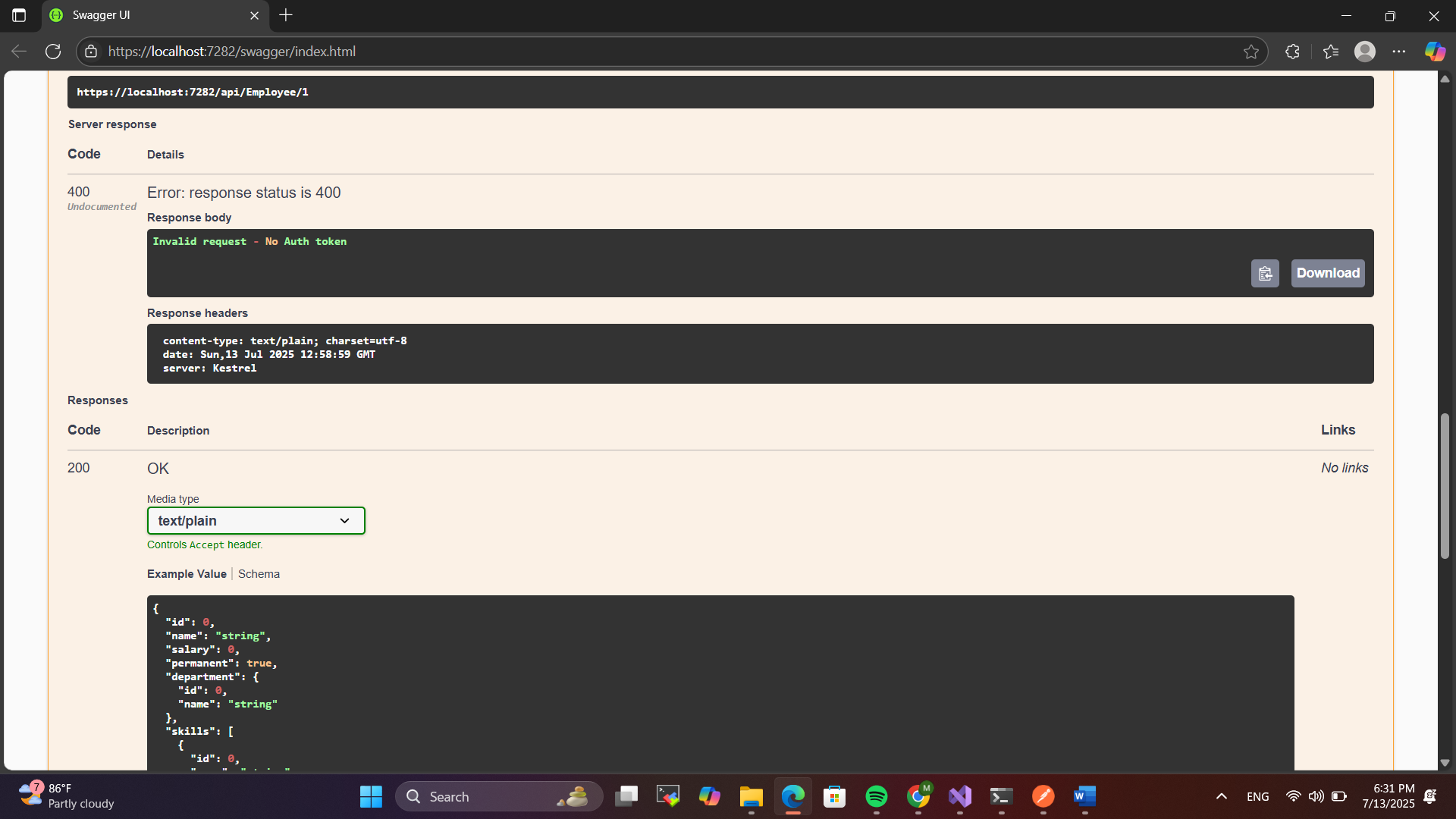
Modify the action method to return Employee data thru ActionResult.

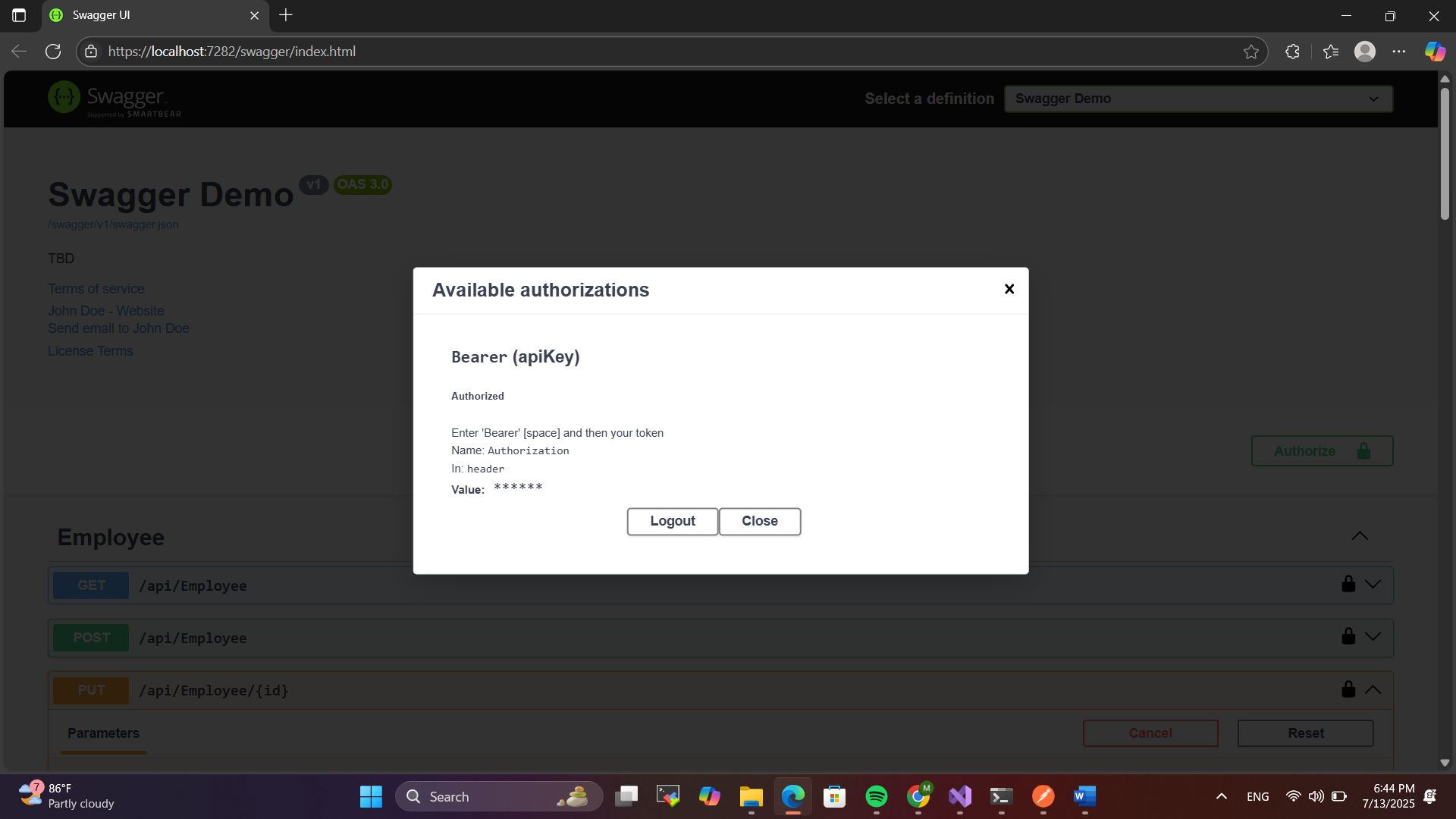
Check if the id value is lesser than or equal to 0. If true, throw BadRequest action result with the message ‘Invalid employee id’

If the value is greater than 0 but not available in the list of employee ids that is there in the hardcoded list of employees, throw BadRequest action result with the same message as stated above.

If the id value is valid, use the JSON data from the input body and update the hardcoded list. Filter the employee list data for the input id and return that as the output.

**Output :**





**5.WEBAPI\_HANDSON**

**Objectives:**

* **Explain CORS enablement for Web API access for local application** o **What is CORS?, How to enable CORS thru Startup.cs, Install Cors nuget package to Web API application**

CORS (Cross-Origin Resource Sharing) is a security feature implemented by browsers to restrict web pages from making requests to a different domain than the one that served the page. In the context of Web API development, enabling CORS is essential when your frontend application (like Angular or React) is hosted on a different domain or port than your backend API. Without CORS, the browser blocks such cross-origin calls. To enable CORS in an ASP.NET Core Web API, you typically configure it in the Program.cs file. This involves adding a CORS policy using builder.Services.AddCors and specifying the allowed origins, methods, and headers. For local development, you can use AllowAnyOrigin, AllowAnyHeader, and AllowAnyMethod to allow unrestricted access. Then, you activate CORS in the middleware pipeline with app.UseCors(), placing it before app.UseAuthorization(). While CORS is included by default in modern ASP.NET Core versions, you may need to install the

Microsoft.AspNetCore.Cors package in older versions. In production environments, it is recommended to restrict CORS access to specific trusted origins for security.

* **Demonstrate security in WebAPI** o **Bearer and Jwt token authentication, Use Authorize attribute & send roles in Jwt token, Setting in Startup.cs for AddAuthentication and AddJwtBearer with validation attributes, UseAuthentication, AllowAnonymous to AuthController to generate token, Claims**

Security in Web API is often implemented using Bearer and JWT (JSON Web Token) authentication to ensure that only authenticated users can access protected endpoints. In this approach, a client first requests a token from a dedicated AuthController, which is decorated with the [AllowAnonymous] attribute to allow unauthenticated access. The token is generated using claims that include the user's identity and roles (e.g., "Admin", "User", or "POC") and is signed with a secure key. The token is then passed in the Authorization header as a Bearer token when accessing secure endpoints. On the server side, JWT authentication is configured in the Program.cs file using AddAuthentication and AddJwtBearer, where token validation parameters such as issuer, audience, and signing key are set. The [Authorize] attribute is applied to controllers or actions to restrict access, and roles can be specified using [Authorize(Roles = "Admin")] to allow role-based access control. This ensures that only users with valid tokens and proper roles can interact with sensitive parts of the API, enhancing security by validating identity and access rights.

**1. JsonWebToken**

There are various modes of authenticating a request. Json Web Token(JWT) authentication is one among them. It is a methodology of passing a token in the Authorization header value in the request so that it can be checked at the WebAPI and validated. If not there, then ‘**Unauthorized**’ status message with status code 401 should be thrown.

Use the below code in Startup.cs

• In ConfigureServices method

string securityKey = "mysuperdupersecret"; var symmetricSecurityKey = new

SymmetricSecurityKey(Encoding.UTF8.GetBytes(securityKey)); services.AddAuthentication(x =>

{

x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

x.DefaultSignInScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, x =>

{

x.TokenValidationParameters = new TokenValidationParameters

{

//what to validate

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

//setup validate data

ValidIssuer = "mySystem",

ValidAudience = "myUsers",

IssuerSigningKey = symmetricSecurityKey

};

});

• In Configure method app.UseAuthentication();

This is to enable the JWT authentication in .Net core

Create a new controller ‘AuthController’ in the Web API application. Add **AllowAnonymous** attribute to the controller. Create a private method GenerateJSONWebToken as shown thru the code below.

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);

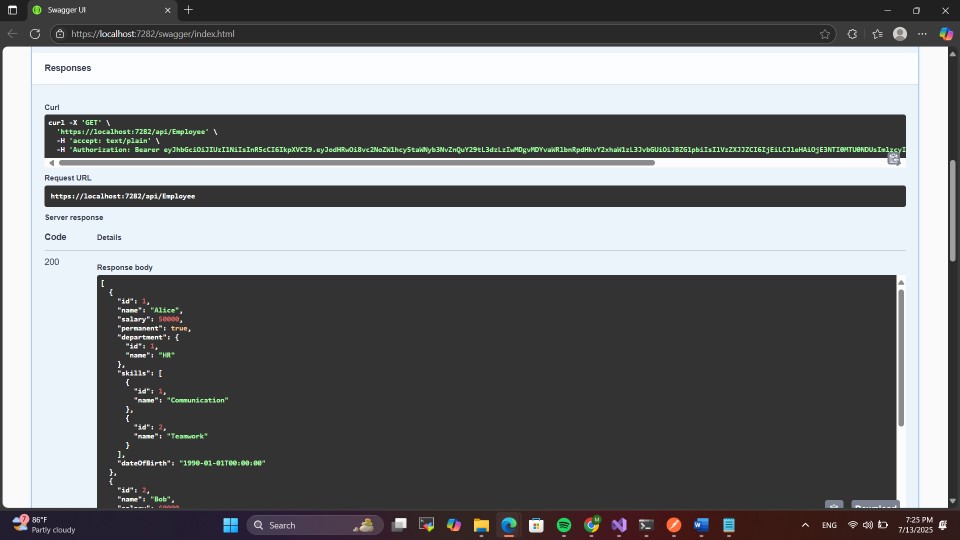
return new JwtSecurityTokenHandler().WriteToken(token);

}

Note that the issuer, audience and the securitykey defined in the Startup.cs code and method code shown above is the same and should match.

Invoke the GenerateJSONWebToken in the GET action method by sending some value for user id and ‘Admin’ for the user role. This is to set Claims information to check the user role

**Output**



**2. Use the JWT generated thru the AuthController to be used in POSTMAN request.**

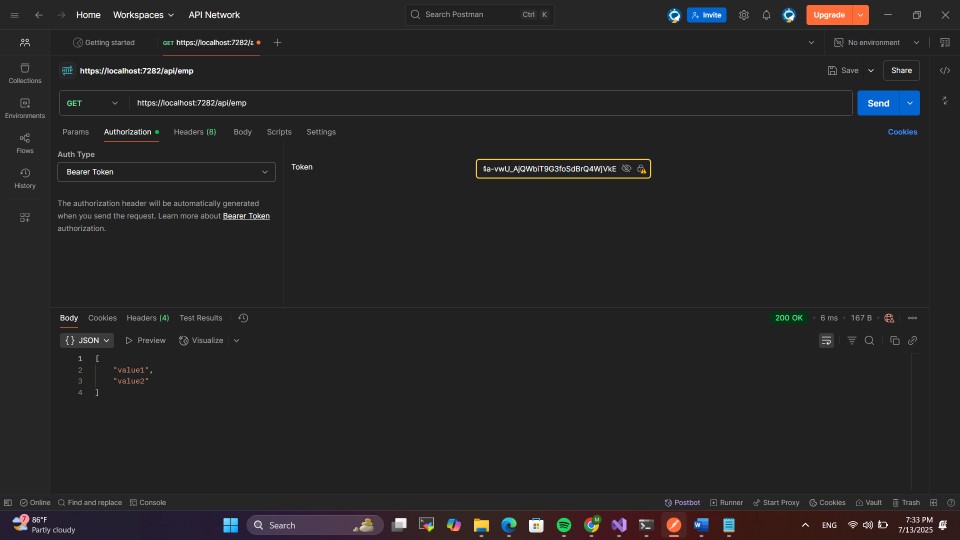
Remove the ‘CustomAuthFilter’ that is currently mapped to the Employee controller(if already done)

Use **Authorize** attribute in the Employee controller to check if the request header contains ‘Authorization’ key with ‘Bearer’ token or not. If the token is unavailable, then ‘Unauthorized’ status message whose code 401 will be thrown. Use POSTMAN to hit a GET action method call. Note the ‘Status’ attribute in the ‘Headers’ section in the output window.

Use the AuthController to generate the JWT. Use that in the GET action method call thru POSTMAN if the request is authenticated or not.

Modify the token value in the POSTMAN tool and check if ‘Unauthorized’ status message is thrown. Note the ‘Status’ attribute in the ‘Headers’ section in the output window.

**Output:**



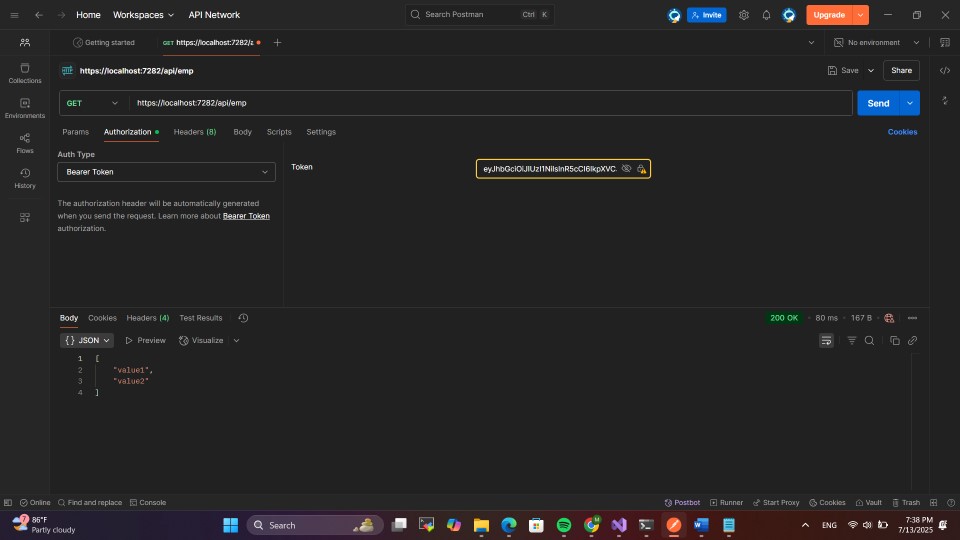
1. **Check for JWT expiration**

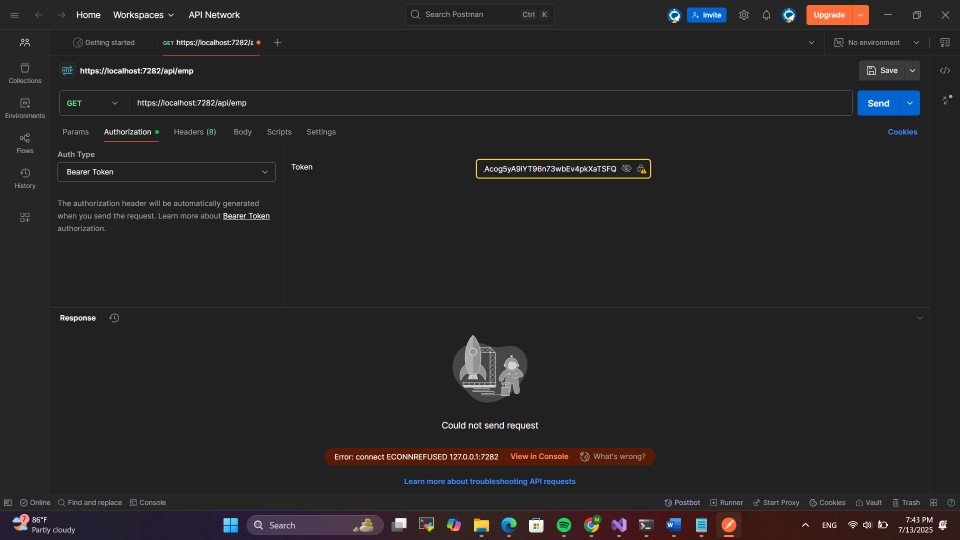
A JWT token has an attribute which can be set to determine how long the token is valid to be used.

In the GenerateJSONWebToken method in AuthController, the ‘expires’ attribute of the JwtSecurityToken object denotes the time in minutes for which the token would be valid.

Modify the duration for ‘expires’ attribute to 2 minutes. Check the POSTMAN request for GET call **AFTER** 2 minutes of generation of the JWT, which should yield ‘Unauthorized’ message with Http status code 401.

**Output:**





1. **Add the roles to be authorized in the Authorize attribute.**

The **Authorize** attribute supports the roles to be used to filter the controller action method access.

The GenerateJSONWebToken method of AuthController uses the role ‘Admin’ set in the claims.

Include the role ‘**POC**’ in the Authorize attribute in the Employee controller. Hit the GET action method of the Employee controller thru POSTMAN. Verify if the response status is ‘Unauthorized’ with status code 401

Include the role ‘**Admin**’ along with ‘POC’ in the Authorize attribute in the Employee controller. Hit the GET action method of the Employee controller thru POSTMAN. Verify if the response status is OK with status code 200

**Output:**

