Week 4 Assignment Solutions (ASP.NET Core 8.0 Web API)

1. WebApi_Handson

1.1. Concept of RESTful Web Service, Web API & Microservices

RESTful Web Service:

REST (Representational State Transfer) is an architectural style that uses standard HTTP methods for communication between client and server. It is stateless and emphasizes resources (data objects) and their representation.

Features of REST Architecture:

- Representational State Transfer: Every resource is represented by a URI, and the resource can have multiple representations (e.g., JSON, XML).
- Stateless: Each request from client to server must contain all the information needed to understand and process the request.
- Messages: REST uses HTTP messages—requests from the client and responses from the server.
- Not Restricted to XML: Responses can be in multiple formats like JSON, XML, plain text, or HTML.

Web API:

Web API is a framework in .NET used for building HTTP services. It allows interaction through HTTP protocols and is ideal for creating RESTful applications.

Microservices:

Microservices architecture is an approach to developing software systems where complex applications are broken into smaller, independent services that communicate over HTTP/REST.

Difference between WebService & WebAPI:

Feature	WebService (.asmx)	WebAPI
Protocol	SOAP	HTTP/REST
Data Format	XML only	JSON, XML, others
Hosting	IIS only	IIS, Self-hosting, Kestrel
Lightweight	No	Yes

1.2. HTTP Request and HTTP Response

- HttpRequest: A message sent by the client to request data from the server. It contains methods like GET, POST, headers, and parameters.
- HttpResponse: The server's response to the client's request. It includes status codes, response body, headers, etc.

1.3. Types of Action Verbs

HttpGet: Retrieves data from the server.

```
[HttpGet]
public IActionResult GetAllItems() {}
```

HttpPost: Sends new data to the server.

```
[HttpPost]
public IActionResult CreateItem(Item item) {}
```

HttpPut: Updates existing data on the server.

```
[HttpPut]
public IActionResult UpdateItem(int id, Item item) {}
```

HttpDelete: Deletes data on the server.

```
[HttpDelete]
public IActionResult DeleteItem(int id) {}
```

1.4. HTTP Status Codes in WebAPI

200 OK: Request was successful.

400 BadRequest: Invalid input from client.

401 Unauthorized: Authentication failed.

500 InternalServerError: Server error occurred.

Example:

```
return Ok(data);              // 200 OK
return BadRequest("Error");   // 400 Bad Request
return Unauthorized();        // 401 Unauthorized
return StatusCode(500);       // 500 Internal Server Error
```

1.5. Creation of Simple WebAPI with Read/Write Actions

Steps - {Create a new .NET Core Web API project, Add a controller with CRUD actions.}

ValuesController :

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    private static List<string> values = new List<string> { "Value1", "Value2" };

    [HttpGet]
    public IActionResult Get() => Ok(values);

    [HttpPost]
    public IActionResult Post([FromBody] string value)
    {
        values.Add(value);
        return Ok(values);
    }
}
```

1.6. Configuration Files in WebAPI

Startup.cs

- Configures services and middleware.
- Dependency Injection is registered here.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Conclusion:
This document provided an overview of RESTful architecture, Web API, microservices, HTTP verbs and status codes, basic API creation, and .NET configuration files. These are foundational concepts for building scalable and maintainable APIs in .NET Core.

2. WebApi_Handson

2.1. Setting Up Swagger in a .NET Core Web API Project

Step-by-Step:

- Create or open an existing .NET Core Web API project.
- Install the Swashbuckle.AspNetCore NuGet package:

```
Install-Package Swashbuckle.AspNetCore
```

Modify `Startup.cs`:

In `ConfigureServices` method:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo
    {
        Title = "Swagger Demo",
        Version = "v1",
        Description = "TBD",
        TermsOfService = new Uri("https://example.com/terms"),
        Contact = new OpenApiContact
        {
            Name = "John Doe",
            Email = "john@xyzmail.com",
            Url = new Uri("https://www.example.com")
        },
        License = new OpenApiLicense
        {
            Name = "License Terms",
            Url = new Uri("https://www.example.com")
        }
    });
});
```

In `Configure` method:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Swagger Demo");
});
```

Execution:

Run the application.

Navigate to: `https://localhost:[port]/swagger`

Observe the title, version, contact, and controller methods listed.

Use "Try it out" and "Execute" features for testing.

2.2. Using Postman to Test WebAPI

Basic Features:

Headers: Include `Authorization`, `Content-Type`, etc.

Body: Choose raw and JSON format for sending POST/PUT data.

Request Type: Choose GET, POST, PUT, DELETE, etc.

Request Collection: Group requests for reuse.

Tabs: Center pane shows editable request and response interface.

Example:

Launch Postman.

Create a new request to `https://localhost:[port]/api/employee`.

Select method: `GET`.

Hit Send.

Observe: `Body` tab shows list of employees, `Status` in top right shows success (e.g., 200 OK).

Modify Route:

Change controller attribute in `EmployeeController.cs`:

```
[Route("api/Emp")]
```

```
public class EmployeeController : ControllerBase { }
```

Test again in Postman with new endpoint: https://localhost:[port]/api/Emp

2.3. Using Route, Name Attribute, and ActionName

Route Attribute:

Allows customization of endpoint paths.

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

Change it to:

```
[Route("api/Emp")]
```

Name Attribute:

Gives meaningful identifiers to routes.

```
[HttpGet(Name = "GetAllEmployees")]
public IEnumerable<Employee> Get() { ... }
```

ActionName:

Enables multiple methods with same verb by renaming method identity.

```
[HttpGet]
[ActionName("GetById")]
public IActionResult GetById(int id) { ... }

[HttpGet]
[ActionName("GetByDept")]
public IActionResult GetByDept(string dept) { ... }
```

2.4. Creating Simple Web API with Read & Write Actions

Controller Example:

```
[Route("api/[controller]")]
[ApiController]
public class EmployeeController : ControllerBase
{
   private static List<string> employees = new List<string> { "Raj", "Shalu" };

   [HttpGet]
   public IActionResult Get() => Ok(employees);

   [HttpPost]
   public IActionResult Post([FromBody] string emp)
   {
      employees.Add(emp);
      return Ok(employees);
   }
}
```

Test with Swagger and Postman

Swagger: Use "Try it out" on GET/POST.

Postman: Set method, URL, and JSON body (for POST), then hit Send.

Conclusion:
This hands-on document helps understand how to integrate Swagger into Web API projects, test with Postman, use routing techniques, and build simple APIs for CRUD operations. These are essential practices in modern API development using .NET Core.

3. WebApi_Handson

3.1. Custom Model Class: Employee

Model: Employee

```
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; }
}
```

Additional Supporting Models:

```
public class Department
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Skill
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

3.2. Controller: EmployeeController

Sample Constructor + GET/POST Actions:

```
[Route("api/[controller]")]
[ApiController]
[CustomAuthFilter]
public class EmployeeController : ControllerBase
{
    private List<Employee> _employees;

    public EmployeeController()
    {
        _employees = GetStandardEmployeeList();
    }

    private List<Employee> GetStandardEmployeeList() { /* Return mock list */ }
```

```
  [HttpGet]
  [ProducesResponseType(typeof(List<Employee>), 200)]
  public ActionResult<List<Employee>> GetStandard()
  {
    throw new Exception("Custom error for testing");
    return _employees;
  }

  [HttpPost]
  [AllowAnonymous]
  public IActionResult AddEmployee([FromBody] Employee emp)
  {
    _employees.Add(emp);
    return Ok(_employees);
  }
}
```

3.3. Explanation of Attributes

AllowAnonymous: Skips authorization for specific endpoints.

HttpGet/HttpPost/HttpPut: Maps action methods to HTTP verbs.

FromBody: Reads object from request body (e.g., POST JSON data).

ProducesResponseType: Describes expected HTTP status codes for Swagger documentation.

3.4. Creating Custom Authorization Filter

Filters/CustomAuthFilter

```
public class CustomAuthFilter : ActionFilterAttribute
{
  public override void OnActionExecuting(ActionExecutingContext context)
  {
    var headers = context.HttpContext.Request.Headers;

    if (!headers.ContainsKey("Authorization"))
    {
      context.Result = new BadRequestObjectResult("Invalid request - No Auth token");
      return;
    }

    if (!headers["Authorization"].ToString().Contains("Bearer"))
    {
      context.Result = new BadRequestObjectResult("Invalid request - Token present but Bearer unavailable");
      return;
    }

    base.OnActionExecuting(context);
  }
}
```

Apply `[CustomAuthFilter]` on controllers to protect all actions.

3.5. Custom Exception Filter

Class: CustomExceptionFilter

```
public class CustomExceptionFilter : IExceptionFilter
{
   public void OnException(ExceptionContext context)
   {
      string filePath = Path.Combine(Directory.GetCurrentDirectory(), "Logs", "errors.txt");
      Directory.CreateDirectory(Path.GetDirectoryName(filePath));

      File.AppendAllText(filePath, $"{DateTime.Now}: {context.Exception.Message}\n");

      context.Result = new ObjectResult("Internal server error")
      {
         StatusCode = 500
      };
   }
}
```

Register this globally in Startup.cs under services.AddControllers():

```
services.AddControllers(options =>
{
   options.Filters.Add(typeof(CustomExceptionFilter));
});
```

NuGet Package Required:

```
Install-Package Microsoft.AspNetCore.Mvc.WebApiCompatShim
```

3.6. Swagger Testing

Run the pplication and navigate to /swagger.

Check that GET method on EmployeeController returns 500 (Internal Server Error) due to the exception.

ProducesResponseType for 200 and 500 is reflected in Swagger documentation.

Conclusion:
This documentation demonstrated how to:

Create and return custom class data using Web API

Apply and understand filter attributes (AllowAnonymous, FromBody, ProducesResponseType)

Build custom filters for authorization and exception handling

Integrate with Swagger for meaningful API testing and documentation

These components form a robust foundation for enterprise-grade .NET Core WebAPI solutions.

4. WebApi_Handson

4.1. Model Class: Employee

```
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 int Id { get; set; }
    public string Name { get; set; }
}

public class Skill
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

4.2. EmployeeController with PUT (Update) Action

Controller Setup:

```
[Route("api/[controller]")]
[ApiController]
public class EmployeeController : ControllerBase
{
    private static List<Employee> employees = new List<Employee>
    {
        new Employee { Id = 1, Name = "Raj", Salary = 50000 },
        new Employee { Id = 2, Name = "Shalu", Salary = 60000 }
    };

    [HttpPut("{id}")]
    public ActionResult<Employee> UpdateEmployee(int id, [FromBody] Employee updatedEmp)
    {
        if (id <= 0)
        {
            return BadRequest("Invalid employee id");
        }

        var emp = employees.FirstOrDefault(e => e.Id == id);
        if (emp == null)
        {
            return BadRequest("Invalid employee id");
        }

        emp.Name = updatedEmp.Name;
        emp.Salary = updatedEmp.Salary;
        emp.Permanent = updatedEmp.Permanent;
        emp.Department = updatedEmp.Department;
        emp.Skills = updatedEmp.Skills;
        emp.DateOfBirth = updatedEmp.DateOfBirth;

        return Ok(emp);
    }
}
```

4.3. Explanation of Key Concepts

FromBody Attribute:

Binds data from the request body (typically in JSON format) to the custom model class.

Used for POST, PUT operations where complete object data is expected from the client.

ActionResult<Employee>:

Ensures flexible return types like BadRequest, Ok, or even custom results.

Useful in real-world APIs for returning proper status and payload.

4.4. Testing With Postman

Steps:

Open Postman and create a new request.

Set Method = PUT, URL = https://localhost:[port]/api/employee/1

Under Headers, add:

Key = Content-Type, Value = application/json

Under Body, select raw and JSON, then provide the updated employee object:

```
{
  "id": 1,
  "name": "Raj Vardhan",
  "salary": 75000,
  "permanent": true,
  "department": { "id": 1, "name": "Development" },
  "skills": [
    { "id": 1, "name": "C#" },
    { "id": 2, "name": "ASP.NET" }
  ],
  "dateOfBirth": "1998-05-12T00:00:00"
}
```

Conclusion:
This document outlined how to create a WebAPI with update (PUT) capability for Employee data. Using FromBody, ActionResult, and hardcoded data lists, we handled validation and error responses. Testing through Swagger and Postman ensures reliability and usability of the API endpoints.

5. WebApi_Handson

5.1. What is CORS?

CORS (Cross-Origin Resource Sharing) is a mechanism that allows restricted resources on a web server to be accessed from another domain.

Why it's needed:

By default, browsers block cross-origin requests for security. APIs must explicitly allow them.

How to Enable CORS in Startup.cs:

Install NuGet package:

```
Install-Package Microsoft.AspNetCore.Cors
```

In ConfigureServices:

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

In Configure:

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

5.2. JWT Authentication Setup

Install JWT Dependencies:

```
Install-Package Microsoft.AspNetCore.Authentication.JwtBearer
```

In Startup.cs → ConfigureServices:

```
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(x =>
{
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "mySystem",
        ValidAudience = "myUsers",
        IssuerSigningKey = symmetricSecurityKey
    };
});
```

In Startup.cs → Configure:

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

5.3. AuthController for Token Generation

```
[AllowAnonymous]
[Route("api/[controller]")]
[ApiController]
public class AuthController : ControllerBase
{
   [HttpGet]
   public IActionResult GetToken()
   {
      var token = GenerateJSONWebToken(101, "Admin");
      return Ok(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(10),
         signingCredentials: credentials);

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

5.4. EmployeeController with JWT Authorization

Basic Setup:

```
[Authorize(Roles = "Admin,POC")]
[Route("api/[controller]")]
[ApiController]
public class EmployeeController : ControllerBase
{
   [HttpGet]
   public IActionResult GetEmployees()
   {
      return Ok("Employee list returned.");
   }
}
```

5. Testing in Postman

Steps to Generate and Use Token:

- Generate Token:
  - Method: GET
  - URL: https://localhost:[port]/api/auth
- Use Token for API Call:
  - Method: GET
  - URL: https://localhost:[port]/api/employee
  - Header:
    - Key: Authorization
    - Value: Bearer <token>

5.6. Testing Token Expiry and Role Access

Set Token Expiry:

```
expires: DateTime.Now.AddMinutes(2),
```

- After 2 minutes, the token becomes invalid → POSTMAN returns 401 Unauthorized.

Role Restriction Tests:

- Use `Authorize(Roles = "POC")` alone → Admin token fails → 401 Unauthorized
- Use `Authorize(Roles = "Admin,POC")` → Admin token works → 200 OK

Conclusion:
This guide covers enabling CORS for frontend-backend interaction and securing WebAPI using JWT. You learned to generate tokens, protect endpoints with `Authorize`, assign roles via `Claims`, and validate token expiry and usage through Postman or Swagger.

This is the complete solution of all modules till 5,
Done by : Raj Vardhan
SupersetID : 6363514