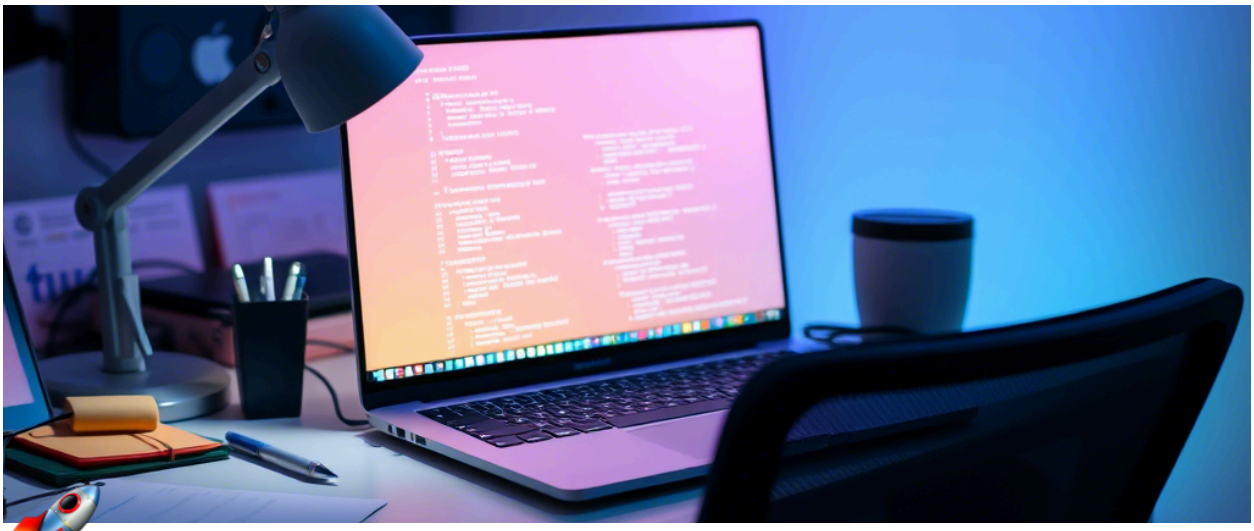


ASP.Net Web API

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ

وَأَنْ لَيْسَ لِلْإِنْسَانِ إِلَّا مَا سَعَىٰ (39) وَأَنَّ سَعْيَهُ سَوْفَ يُرَىٰ (40) ثُمَّ يُجْزَاهُ الْجَزَاءَ الْأَوْفَىٰ (41) وَأَنَّ إِلَىٰ رَبِّكَ الْمُنْتَهَىٰ (42)

📌 My personal accounts links

🔗 LinkedIn	https://www.linkedin.com/in/ahmed-hany-899a9a321?utm_source=share&utm_campaign=share_via&utm_content=profile&utm_medium=android_app
🟢WhatsApp	https://wa.me/qr/7KNUQ7ZI3KO2N1
🟦Facebook	https://www.facebook.com/share/1NFM1PfSjc/

What is a ASP.NET Core ?

is a modern, fast, and cross-platform framework by Microsoft for building web apps and APIs. It works on **Windows, Linux, and macOS** and lets you create:

- Websites
- APIs (for mobile apps or other services)
- Real-time apps (like chat apps)

It's powerful, open-source, and great for building high-performance, scalable apps.

How ASP.NET Core Differs from ASP.NET?

Feature	ASP.NET	ASP.NET Core
Platform	**Windows-only**	**Cross-platform (Windows, macOS, Linux)**
Performance	**Moderate**	**High**
Architecture	**Monolithic**	**Modular**
Open Source	**Partially**	**Fully**
Hosting	**IIS (Internet Information Services)**	**Kestrel, IIS, Nginx, Apache, etc.**
Dependency Injection	**Requires third-party libraries**	**Built-in**
API & MVC Separation	**Separate frameworks**	**Unified framework**

💡 From Where Web API Come ?

In the past, if you wanted to share functionality or logic between applications, you would:

- **Create a Class Library** (DLL) in .NET.
- **Add a Reference** to that DLL in your main application.
- **Directly use the classes and methods** provided by the DLL.

📌 Limitations of the Old Approach:

1. **Local Access Only:** The DLL had to be physically present on the same machine or network.
2. **Platform Dependency:** Typically limited to applications built with the same technology stack (e.g., .NET apps).
3. **Tight Coupling:** Changes to the DLL required redeployment to all consuming applications.

🌐 The Rise of Web API (Modern Approach):

Web API is essentially the same concept as your old Class Library but **exposed over HTTP via a web server**. This allows it to be **accessed from anywhere** using a URL or domain, like:

```
https://myweb.com/api/data
```

✅ Why Web APIs Became Popular:

1. **Platform Independence:** Any application (web, mobile, desktop) can access the API using HTTP.
2. **Global Accessibility:** APIs hosted on a web server can be accessed from anywhere with an internet connection.
3. **Scalability:** Web APIs can be scaled independently of the consuming applications.
4. **Security & Authentication:** APIs can be protected using modern techniques (JWT, OAuth, API Keys).
5. **Standard Communication Protocols:** Uses standard protocols like **HTTP, JSON, and XML**, making it flexible for various clients.

🚀 How It Works:

1. **Create a Web API** using ASP.NET Core.
2. **Deploy the API** to a web server (IIS, Nginx, Azure, etc.).

3. **Access the API** using a URL or domain (e.g., `https://example.com/api/data`).
4. **Consume the API** from any application using HTTP requests (GET, POST, PUT, DELETE).

💡

🌐 What is a Web API?

A **Web API (Application Programming Interface)** is a set of rules and endpoints that allow different applications to communicate with each other **over the internet** using standard protocols like **HTTP/HTTPS**. It provides a way for clients (like web apps, mobile apps, or other servers) to **access and manipulate data** on a server through simple requests.

- Your hospital's website can send a request to a Web API to get a list of doctors.
- A mobile app can send a request to book an appointment through the same API.

It uses **HTTP** (like a website) and usually sends data in **JSON** format. You can interact with it through URLs like:

```
https://example.com/api/doctors
```

This makes it easy for different apps or devices to share data and interact with each other.

💡

What is Kestrel?

Kestrel is a web server that receives and processes **HTTP requests** coming from users. So, if you have a web application, it's the one that listens to requests from the browser and forwards them to your application to handle.

How does Kestrel work?

1. **Receiving Requests**: The first thing Kestrel does is receive a request when a user asks for a page or performs an action on your website.
2. **Passing Through Middleware**: After that, Kestrel passes the **HTTP request** through a chain of **middleware**. Each middleware has a specific job, like **authentication**, **logging**, **routing**, or even modifying the request if there's an issue.
3. **Routing the Request to the Controller**: Once the request goes through the middleware, it reaches the **controller**, where the business logic is located to process the request.
4. **Returning the Response to the User**: Finally, after the controller processes the request, Kestrel sends back an **HTTP response** to the user.

Why is Kestrel important?

- **High Performance**: Kestrel is designed to handle a large number of users simultaneously. If you have **high traffic**, it's the perfect choice.
- **Cross-Platform**: You can run Kestrel on multiple operating systems like **Windows**, **Linux**, and **macOS**, meaning it's not limited to a specific environment.
- **Security**: It supports **HTTPS**, which helps protect the data exchanged between the user and the server.
- **Middleware Integration**: It's very easy to add middleware to include additional functionality, like **authentication**, **logging**, or **error handling**.
- **Scalability**: If you have an application that needs to handle many users, Kestrel helps you scale easily.

A Simple Example to Explain Kestrel

Imagine you're at a restaurant ordering a dish:

- Kestrel is the **waiter** who takes your order.
- The **middleware** are the **chefs** preparing your meal (each chef has a specific task).
- The **controller** is the **restaurant manager** who reviews your order to make sure it's ready.
- Kestrel is the one that brings your **order** back to you once it's ready.

Conclusion

Kestrel is the **default web server** for ASP.NET Core, and it’s essential for receiving and processing **HTTP requests** with high performance and security. It works across major operating systems, handles high traffic well, and is ideal for modern web applications that need scalability and flexibility

What is a RESTful ?

refers to a set of principles and conventions used when designing **Web APIs** based on **REST** (Representational State Transfer) architecture. RESTful APIs are designed to be simple, scalable, and stateless, and they use standard HTTP methods to interact with resources.

✅ Key Principles of RESTful APIs:

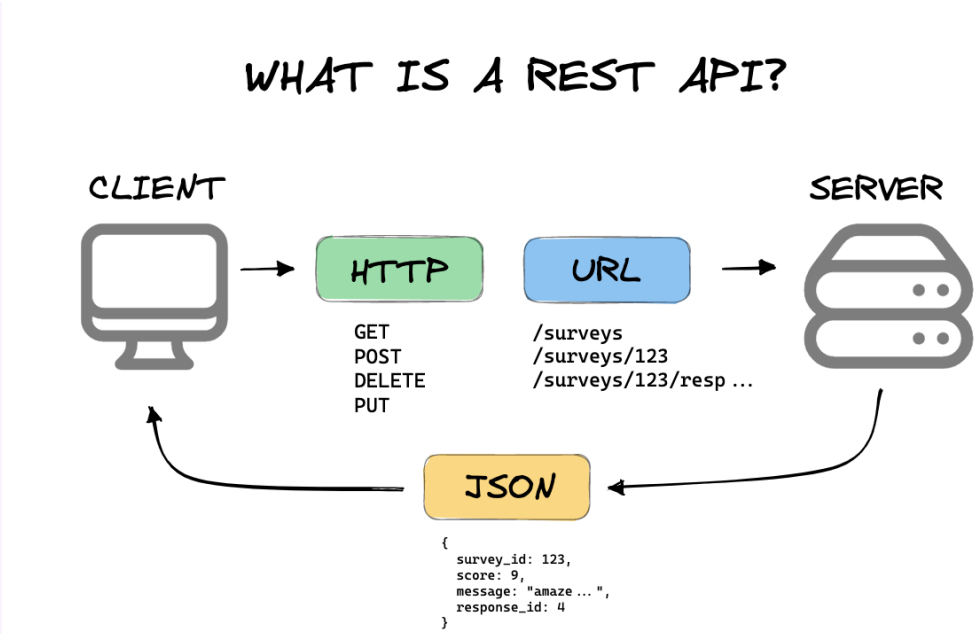
1. **Stateless**: Each API request is independent and contains all the information needed to complete it. The server does not store any information about previous requests ,after response the server forget you
2. **Client-Server Architecture**: The client (such as a web browser or mobile app) and the server (where the data is stored) are separate. They communicate over HTTP but do not rely on each other's implementation.
3. **Uniform Interface**: The API should have consistent and predictable endpoints (URLs) and use standard HTTP methods (GET, POST, PUT, DELETE).
4. **Resources**: RESTful APIs treat data as **resources**. A resource can be anything you want to expose to the client, such as a **doctor, patient, or appointment**. Resources are accessed through URLs, like:

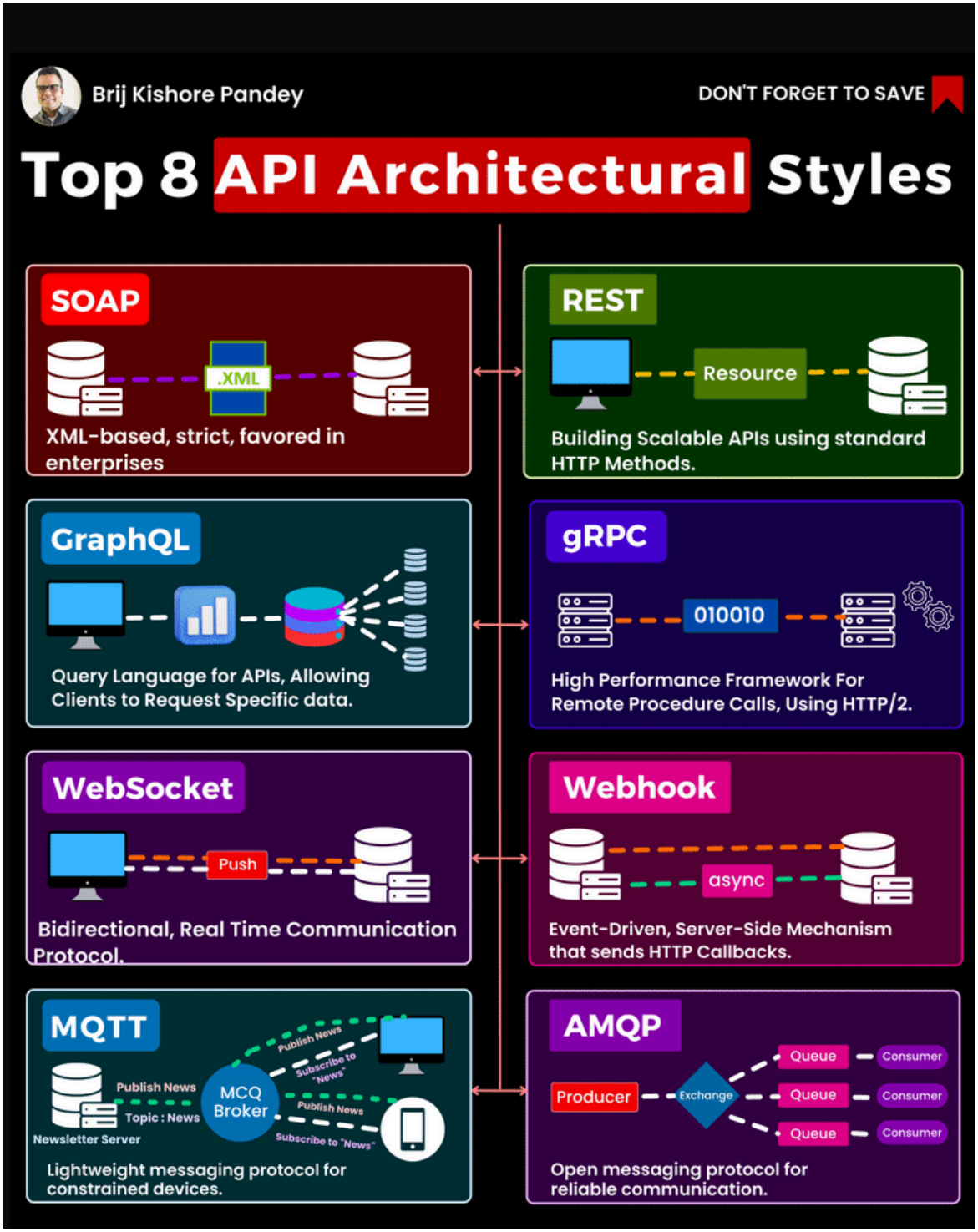
   ```
   https://example.com/api/doctors
   https://example.com/api/patients
   ```

5. **Use of HTTP Methods**:
   - **GET**: Retrieve data (get a list of doctors).
   - **POST**: Create new data (e.g., add a new patient).
   - **PUT**: Update existing data (e.g., update a doctor's information).
   - **DELETE**: Delete data (e.g., remove a patient).
6. **Representation**: Resources can be represented in different formats, but the most common format is **JSON**.

🌟 Why RESTful APIs are popular:

- **Simplicity**: Easy to understand and use.
- **Scalability**: Works well for large, distributed systems.
- **Stateless**: No need to store session information between requests.


WHAT IS A REST API?
CLIENT
SERVER
HTTP
URL
GET
POST
DELETE
PUT
/surveys
/surveys/123
/surveys/123/resp ...
JSON
{
survey_id: 123,
score: 9,
message: "amaze ... ",
response_id: 4
}


Brij Kishore Pandey
DON'T FORGET TO SAVE
Top 8 API Architectural Styles
SOAP
.XML
XML-based, strict, favored in enterprises
REST
Resource
Building Scalable APIs using standard HTTP Methods.
GraphQL
Query Language for APIs, Allowing Clients to Request Specific data.
gRPC
010010
High Performance Framework For Remote Procedure Calls, Using HTTP/2.
WebSocket
Push
Bidirectional, Real Time Communication Protocol.
Webhook
async
Event-Driven, Server-Side Mechanism that sends HTTP Callbacks.
MQTT
Publish News
Subscribe to "News"
Publish News
Topic : News
MCQ Broker
Publish News
Subscribe to "News"
Newsletter Server
Lightweight messaging protocol for constrained devices.
AMQP
Producer
Exchange
Queue
Queue
Queue
Consumer
Consumer
Consumer
Open messaging protocol for reliable communication.

What are the constraints in RESTful APIs?

rules that help make sure the API is easy to use, scalable, and consistent. These rules ensure the API works well across different devices and is easy to interact with.

Main Constraints:

1. **Stateless**: Each request has to include everything needed to complete it. The server doesn't remember anything from previous requests.
2. **Client-Server**: The client (app or website) and server (where the data is stored) are separate and communicate via HTTP.
3. **Cacheable**: Responses can be stored temporarily, so the client doesn’t need to ask the server again for the same information.
4. **Uniform Interface**: The API should be predictable and easy to use, with clear rules for accessing resources (like doctors, patients).
5. **Layered System**: The API can use different layers (like load balancers), and the client doesn’t need to worry about how the server is set up.

These rules help make your API work smoothly and efficiently.

💡 How a Web API works between a Service Consumer (client) and a Service Provider (server)?

1. **Service Consumer (Client):**
   - Sends a request to the service provider using a URL (`RequestURL`).
   - Supports various HTTP request methods (**GET, POST, PUT, DELETE**).
   - Can send data in different formats (**JSON or XML**).
   - Specifies the expected response format using the `accept_type` header (e.g., `json` or `xml`).
2. **Request Format:**
   - Example of sending data in XML:

     ```
     <name>Ahmed</name>
     <age>25</age>
     ```

   - Example of sending data in JSON:

     ```
     { "name": "Ahmed", "Age": 25 }
     ```

3. **Service Provider (Server):**
   - Processes the request and returns a response.
   - The response includes:
     - **Request URL** (confirming the endpoint that was called).
     - **Status Code** indicating the result of the request (e.g., 200, 404, 500).
4. **Common Status Codes:**
   - `200`: OK (Successful request).
   - `202`: Accepted (Request received, but processing is not completed).
   - `204`: No Content (Successful request but no content to return).
   - `404`: Not Found (Resource not found).
   - `402`: Payment Required (Reserved for future use).
   - `500`: Internal Server Error (Server encountered an error).

💡 Why API Controller Inherit from `ControllerBase` not inherit from Controller like MVC?

API controllers in ASP.NET Core inherit from `ControllerBase` instead of `Controller` because they are designed **specifically for building web APIs** rather than full MVC applications with views. Here's why:

🔑 Differences Between `Controller` and `ControllerBase`

Aspect	`ControllerBase` (API Controller)	`Controller` (MVC Controller)
View Support	❌ No support for views.	✅ Supports views (`View()` method).
Focus	🌐 Web APIs (JSON/XML responses).	📄 Web applications (HTML views).
Lightweight	✅ Lighter, no view rendering logic.	❌ Heavier, includes view rendering logic.
Methods Available	✅ `Ok()`, `BadRequest()`, `NotFound()`, etc.	✅ Includes API methods + `View()`, `PartialView()`, etc.
Dependency Injection	✅ Same as `Controller`, but focused on APIs.	✅ Supports DI with views.
Usage	✅ For RESTful APIs.	✅ For MVC web applications.

📌 Why Not Use `Controller` for APIs?

1. **Unnecessary Overhead:** The `Controller` class includes features for rendering views (`View()`, `PartialView()`, etc.) which are not needed for APIs. APIs usually return data in formats like JSON or XML.
2. **Separation of Concerns:** APIs are meant to **provide data**, not **render views**. Inheriting from `ControllerBase` makes it clear that the controller is meant for an API.
3. **Built-In Helpers:** `ControllerBase` provides useful methods like `Ok()`, `BadRequest()`, `NotFound()`, etc., which simplify returning standard HTTP status codes.

Example of API Controller

```
[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
    [HttpGet("{id}")]
    public IActionResult GetUser(int id)
    {
        var user = new { Id = id, Name = "John Doe" };
```

```
        if (user == null)
        {
            return NotFound();
        }
        return Ok(user); // Returns JSON response
    }
}
```

Example of MVC Controller

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        var model = new { Message = "Welcome to my website!" };
        return View(model); // Returns an HTML view
    }
}
```

📢 `[Route("api/[controller]")]` is a **convention** used to clearly separate API endpoints from MVC endpoints. By prefixing the route with `"api/"`, you:

- 🔀 **Separate APIs from MVC routes:** Keeps URLs organized and makes it clear these are API endpoints (`/api/Users` instead of just `/Users`).
- 📂 **Prevent Conflicts:** Avoids URL conflicts with MVC controllers or Razor Pages.
- 📚 **Improves Clarity:** Makes it easier for developers to identify API routes in large applications.
- **if you remove “api” no problem**

:

✅ What `[ApiController]` Does

1. **Automatic Model Validation:**
   - Automatically returns `400 Bad Request` if model validation fails.
   - You don't have to manually check `ModelState.IsValid`.

```
[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
    [HttpPost]
    public IActionResult CreateUser(UserDto user)
    {
        // If user is invalid, ASP.NET Core will return 400 automatically.
        return Ok(user);
    }
}
```

2. **Binding Source Inference:**
   - Automatically understands where to get data from (`[FromBody]`, `[FromRoute]`, `[FromQuery]`, etc.).
   - Example: For POST methods, it assumes the data comes from the request body.
3. **Problem Details Responses:**
   - Returns standardized error responses with `ProblemDetails` objects (following RFC 7807).
4. **Route Requirement:**
   - Requires that **all routes are explicitly defined** (e.g., `[Route("api/[controller]")]`).
5. **ProducesResponseType Inference:**
   - Automatically infers response types, making it easier to document your API with Swagger.

📌 Without `[ApiController]`

- You would have to manually validate models and specify binding sources like `[FromBody]`.
- Error handling and response formatting would be more manual.

How can apply HTTP Methods ?

To apply **HTTP methods** in ASP.NET Core, you use specific **action attributes** to indicate the type of HTTP request that an action method should respond to. Here's how you can apply each HTTP method using the appropriate attribute, along with some examples:

1. GET Method

- The **GET** method is used to retrieve data from the server. It doesn't modify the resource, so it's safe to call it multiple times without causing side effects.

Example:

```
[HttpGet("{id:int}")]
public IActionResult GetDepartment(int id)
{
    var department = _context.Departments.FirstOrDefault(d ⇒ d.Id == id);
    if (department == null)
    {
        return NotFound();
    }
    return Ok(department);
}
```

- Here, `GET /api/departments/{id}` fetches the department with the provided `id` from the database.

2. POST Method

- The **POST** method is used to create a new resource on the server. It typically sends data in the **body** of the request.

Example:

```
[HttpPost]
public IActionResult CreateDepartment( DepartmentDto department)
//here get department from body by defailt
{
    if (department == null)
    {
        return BadRequest();
    }

    var newDept = new Department
    {
```

```
        Name = department.DepartmentName,
        ManagerName = department.MangerName
    };

    _context.Departments.Add(newDept);
    _context.SaveChanges();

    return CreatedAtAction(nameof(GetDepartment),
    new { id = newDept.Id }, newDept);
}
```

- Here, `POST /api/departments` creates a new department. The `department` object is passed in the **body** of the request.

3. PUT Method

- The **PUT** method is used to update an existing resource on the server. It usually requires sending a **complex object** in the body with all the properties of the resource to be updated.

Example:

```
[HttpPut("{id:int}")]
public IActionResult UpdateDepartment(int id, DepartmentDto department)
{
    var dept = _context.Departments.FirstOrDefault(d ⇒ d.Id == id);
    if (dept == null)
    {
        return NotFound();
    }

    dept.Name = department.DepartmentName ?? dept.Name;
    dept.ManagerName = department.MangerName ?? dept.ManagerName;

    _context.SaveChanges();
    return NoContent();
}
```

- Here, `PUT /api/departments/{id}` updates the department with the provided `id`. Only the fields provided in the body will be updated.

4. DELETE Method

- The **DELETE** method is used to delete a resource on the server.

Example:

```
[HttpDelete("{id:int}")]
public IActionResult DeleteDepartment(int id)
{
    var dept = _context.Departments.FirstOrDefault(d ⇒ d.Id == id);
    if (dept == null)
    {
        return NotFound();
    }

    _context.Departments.Remove(dept);
    _context.SaveChanges();

    return NoContent();
}
```

- Here, `DELETE /api/departments/{id}` deletes the department with the provided `id` from the database.

Key Points:

1. **HTTP GET**: Used to retrieve resources. Parameters should be passed in the URL (e.g., query parameters or URL parameters).
2. **HTTP POST**: Used to create new resources. Send complex objects (JSON, XML) in the **body**.
3. **HTTP PUT**: Used to update existing resources. Send the complete resource to be updated in the **body**.
4. **HTTP DELETE**: Used to delete a resource. The resource to delete is typically specified in the **URL**.

Best Practices:

- **Primitives in URL**: When sending simple parameters like `id`, they should be passed in the URL.
  Example:
  `GET /api/departments/{id}`
- **Complex Objects in Body**: For operations that require sending more complex data (like creating or updating a department), send the data in the request body as JSON or another supported format.

Summary:

- **GET**: Retrieve data.
- **POST**: Create data.

- **PUT**: Update data.
- **DELETE**: Remove data.

Each HTTP method has its own specific purpose, and the attributes like `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, and `[HttpDelete]` are used to map controller actions to those methods.

What is a Model binding in an API (or any web application framework) ?

refers to the process of automatically mapping data from an HTTP request (such as query strings, form data, or JSON payload) to parameters in an action method in a controller.

In the context of an ASP.NET Core API, model binding takes place when a request is sent to an API endpoint, and the framework tries to match the incoming data to the action method's parameters. This is done in various ways, depending on the content type (like JSON or form data) and the structure of the data being sent.

For example:

- **Query string parameters**: If the URL has parameters like `/api/products?id=1&name=toy`, the API can bind these to a model or individual method parameters.
- **Form data**: If you're sending data from a form (for instance, in a POST request), the data in the form is bound to a model class.
- **JSON**: If the request body contains JSON, the API can deserialize this JSON into an object, automatically binding it to the method parameter.

Here's an example in an ASP.NET Core API:

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

[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpPost]
    public IActionResult CreateProduct([FromBody] Product product)
    {
        // 'product' is automatically populated with data from the request body
        return Ok(product);
    }
}
```

How model binding works:

1. **FromBody**: If you're sending a JSON object (like `{"id": 1, "name": "Toy"}`), the `[FromBody]` attribute tells the framework to deserialize the body content into the `Product` model.

2. **FromQuery**: If you're passing data through query strings, you can bind it using the `[FromQuery]` attribute.
3. **FromRoute**: If you pass data as part of the URL (like `/api/products/{id}`), you can use `[FromRoute]` to bind the route parameter to your method parameter.
4. **FromForm**: If the data comes from a form submission, you can use `[FromForm]` to bind it.

in API by default :

1. **Simple Parameters** (like `int`, `string`, etc.) are automatically taken from the **URL**. For example, if the URL is `/api/department?id=1`, the `id` parameter will be set to `1`.
2. **Complex Objects** (like classes or models) are automatically taken from the **request body**, usually when sending a POST or PUT request. For example, if you send a JSON like:

   ```
   {
      "name": "IT",
      "managerName": "John"
   }
   ```

   This data will be used to populate a `Department` object in your controller.

Example:

- **Simple parameter**:

  ```
  [HttpGet]
  public IActionResult GetDepartmentById(int id)
  {
     return Ok(id);  // 'id' comes from the URL, e.g. /api/department?id=1
  }
  ```

- **Complex object**:

  ```
  [HttpPost]
  public IActionResult CreateDepartment([FromBody] Department department)
  {
     return Ok(department);  // 'department' comes from the body, e.g. JSON data
  }
  ```

Key Rule:

- **Simple types** (like `int`, `string`): From the **URL**.
- **Complex objects** (like `Department`): From the **body** of the request.

What is A DTO?

(Data Transfer Object) is an object that is used to transfer data between layers or systems in an application. It typically contains only the data (no business logic) that needs to be sent across boundaries, such as between the client and server or between different layers of an application.

DTOs are useful for:

1. **Reducing the amount of data being transferred**: You can include only the relevant fields that are needed, instead of sending the entire entity, which may contain unnecessary data.
2. **Decoupling layers**: They help separate the internal representation of an object (such as an entity in a database) from the data that is exposed to the user or the API.
3. **Improving performance**: By sending only the data required, DTOs can reduce the number of database queries or the size of the payload sent over the network.

For example, in a hospital system, a `PatientDTO` could be used to transfer only a patient's name, age, and medical history to the frontend, without exposing sensitive data like their ID or billing information.

Example :

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string Password { get; set; } // sensitive data
}


public class UserDTO
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }  // only non-sensitive data
}
```

use A **record** in C# is a special type of object that is **immutable** (cannot be changed once created) and compares objects based on their data, not their memory reference. This makes records a good choice for DTOs because they:

- Keep the data safe from accidental changes (since they are immutable).
- Make comparing objects easier since they compare values, not references.

So, **using a record for a DTO** is a good practice because it's simpler, safer, and more efficient in most cases.

Example:

```
public record UserDTO(int Id,string Name,  string Email );
```

This `record` is immutable and will only contain data (no extra logic).

💡 What is a CORS policy ?

A **CORS policy** (Cross-Origin Resource Sharing policy) is a security feature implemented by web browsers that restricts web pages from making requests to a **different domain** than the one that served the web page. This is done to **prevent malicious websites from making unauthorized requests** to your server using your credentials.

🔑 Why is CORS Important?

By default, browsers enforce the **Same-Origin Policy**, which means:

- A webpage can only make requests to the **same domain, protocol, and port** from which it was loaded.
- If you try to fetch resources from another domain, the browser will block the request unless the server explicitly allows it via CORS headers.

🌐 How CORS Works

When a client (usually a browser) tries to access resources from a different origin (calling an API from a frontend app hosted on another domain), the server needs to include **CORS headers** in its response to allow the request.

📜 Common CORS Headers

1. `Access-Control-Allow-Origin` - Specifies which origins are allowed.

   ```
   Access-Control-Allow-Origin: https://example.com
   ```

   To allow all origins (not recommended for sensitive data):

   ```
   Access-Control-Allow-Origin: *
   ```

2. `Access-Control-Allow-Methods` - Specifies allowed HTTP methods (e.g., GET, POST).

   ```
   Access-Control-Allow-Methods: GET, POST, PUT, DELETE
   ```

3. `Access-Control-Allow-Headers` - Specifies which headers can be sent with the request.

   ```
   Access-Control-Allow-Headers: Content-Type, Authorization
   ```

4. `Access-Control-Allow-Credentials` - Allows cookies and credentials to be sent with the request.

   ```
   Access-Control-Allow-Credentials: true
   ```

🔄 Preflight Request

For **non-simple requests** (e.g., requests with custom headers or methods other than `GET` or `POST`), the browser first sends an **OPTIONS request** to the server to check if the actual request is allowed. This is called a **preflight request**.

🚀 Configuring CORS in ASP.NET Core

In your `Program.cs` file, you can configure CORS like this:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options ⇒
{
   options.AddPolicy("MyCorsPolicy", builder ⇒
   {
      builder.WithOrigins("https://example.com")
            .AllowAnyHeader()
            .AllowAnyMethod()
            .AllowCredentials();
   });
});

var app = builder.Build();

app.UseCors("MyCorsPolicy");

app.UseAuthorization();
app.MapControllers();
app.Run();
```

✅ Setting Up CORS Policy to Require a Specific Method and Header (`my-key`)

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options ⇒
{
   options.AddPolicy("MyCorsPolicy", builder ⇒
   {
      builder.WithOrigins("https://example.com") // Put your allowed origin here.
            .WithMethods("POST") // Allow only POST requests.
            .WithHeaders("my-key")
            // Allow only requests that contain the header 'my-key'.
            .AllowCredentials();
            // Allow cookies or authentication tokens if needed.
   });
});

var app = builder.Build();

app.UseCors("MyCorsPolicy");

app.UseAuthorization();
app.MapControllers(); // Or app.MapRazorPages(); if you are using Razor Pages.
app.Run();
```

How Can Stop Default validation and apply your validations?

```
//in main
builder.Services.AddControllers().ConfigureApiBehaviorOptions(options =>
{
    options.SuppressModelStateInvalidFilter = true;
});


//Example for add your validation
    [HttpPost]
    public IActionResult Addepartment(DepartmentDto department)
    {

        if (ModelState.IsValid)
        {
            deptrepo.AddDepartment(department);

            return Created();
        }else
        {
          //here can add spectif error massage like
          //          ModelState.AddModelError("Filde you want", "message ");
            return BadRequest(ModelState);
        }
    }
```

you can add specific error massage like:

`ModelState.AddModelError("Filde you want", "message ");`

🔐 What is Difference Between Authentication and Authorization ?

✅ Authentication = "Who are you?"

- **Purpose**: Confirms the identity of a user.
- **Think of it as**: Login process — checking username and password.
- **Result**: Once authenticated, the system knows *who* the user is.
- **Example**: You enter your email and password on a website. If correct, you are authenticated.

🧠 It answers the question:

> "Is this person really who they say they are?"

🔓 Authorization = "What are you allowed to do?"

- **Purpose**: Controls what an authenticated user is allowed to access.
- **Think of it as**: Checking permissions — like access to certain pages or data.
- **Result**: Grants or denies access based on roles, claims, or policies.
- **Example**: A doctor can see medical records; a patient cannot see other patients' data.

🧠 It answers the question:

> "Now that I know who you are, what can you do?"

🔄 In Simple Terms:

Feature	Authentication	Authorization
Checks	Identity (who you are)	Permissions (what you can do)
Happens First	✅ Yes	🔒 Only after authentication
Example	Login with username/password	Accessing `/admin` page
Used For	Login process	Access control

🛠️ What is Difference Between Claim and Role?

Claim:

- **Definition**: A claim is a key-value pair that provides information about a user. Claims can be used to store any data about the user (like their name, email, age, or even custom attributes like "IsAdmin" or "PreferredLanguage").
- **Granularity**: Claims are more granular and flexible. You can use them to store specific, detailed information about a user that may not necessarily relate to authorization.
- **Use Cases**: Claims are ideal when you need to store arbitrary user data (e.g., user preferences, custom attributes, or metadata).
- **Example**: A claim might be used to store a user's email address, membership date, or custom data like "SubscriptionLevel."

```
var claims = new List<Claim>
{
   new Claim(ClaimTypes.Name, user.Name),
   new Claim("SubscriptionLevel", "Premium")
};
```

Role:

- **Definition**: A role is a predefined category that groups users based on their permissions or responsibilities. It is typically used for assigning access rights or roles like "Admin", "Doctor", "Patient", etc.
- **Granularity**: Roles are less granular than claims and are used to assign users to a broad category. Roles are often used in authorization policies to determine if a user has permission to access a specific resource.
- **Use Cases**: Roles are ideal when you need to classify users into broad categories to control access to resources (e.g., a user being an "Admin" or "User").
- **Example**: You might assign a user the role "Admin" or "Patient" to manage access to certain pages or resources.

```
var roles = new List<string> { "Admin", "Manager" };
```

Key Differences:

1. **Purpose**:
    - **Claim**: Stores user-specific information (attributes or properties).

- **Role**: Groups users based on their function or access level.

2. **Flexibility**:
   - **Claim**: More flexible and detailed; you can define custom claims.
   - **Role**: Less flexible; predefined categories used for broad authorization.
3. **Use in Authorization**:
   - **Claim**: Can be used for fine-grained authorization by checking specific attributes.
   - **Role**: Used for coarse-grained authorization by checking whether a user belongs to a particular group (`User.IsInRole("Admin")`).
4. **Storage**:
   - **Claim**: Can hold any data about the user.
   - **Role**: Typically represents a single, fixed value related to user permissions.

🛠️ What is Difference between `AddScoped`, `AddTransient` and `AddSingIton`?

🔁 `AddTransient` – Always New

- **Lifetime**: A new instance is created **every time** the service is requested.
- **Use Case**: For **lightweight, stateless** services that don't keep any state or data.
- **Example**: A service that generates random numbers or a service that formats a string.

```
services.AddTransient<IMyService, MyService>();
```

➡️ Every time `IMyService` is injected, a **new instance** of `MyService` is created.

📦 `AddScoped` – One Per Request

- **Lifetime**: A new instance is created **once per HTTP request**.
- **Use Case**: For services that should maintain state **only within the current request**, like working with Entity Framework's `DbContext`.
- **Example**: A service that tracks user actions during a single HTTP request.

```
services.AddScoped<IMyService, MyService>();
```

➡️ One instance is created per HTTP request and reused within that request.

🧠 `AddSingleton` – One For All

- **Lifetime**: A single instance is created **once and shared** across the entire application.
- **Use Case**: For **stateless or cached services** that are expensive to create or hold global state.
- **Example**: A configuration service, logger, or a cache manager.

```
services.AddSingleton<IMyService, MyService>();
```

➡️ The same instance is used throughout the app's lifetime, from startup to shutdown.

🧪 Summary Table:

Lifetime	When New Instance is Created	Shared In...	Suitable For

`Transient`	Every time it's requested	Never shared	Lightweight, stateless services
`Scoped`	Once per HTTP request	Within one request	Request-specific operations (e.g. DbContext)
`Singleton`	Once for the whole app	Entire app lifetime	Shared/global services (e.g. config, cache)

🚨 Notes:

- Be **very careful** using `AddSingleton` with classes that depend on `Scoped` services like `DbContext` — it can lead to runtime errors.
- Use `Scoped` most often in web apps, especially when working with databases.

✅ What is a Token?

A **token** is a piece of data used to authenticate and authorize users when interacting with a web server. It is a credential that a client (like a browser or mobile app) uses instead of traditional credentials (like a username and password) to prove their identity. Tokens are typically short-lived and expire after a certain time for security reasons.

✅ What is JWT (JSON Web Token)?

JWT (JSON Web Token) is a popular format for creating tokens that are:

- **Compact**: Easy to pass in URLs, HTTP headers, or even inside cookies.
- **Self-contained**: It includes all the information needed to verify its validity without requiring server-side storage.

📌 Structure of a JWT

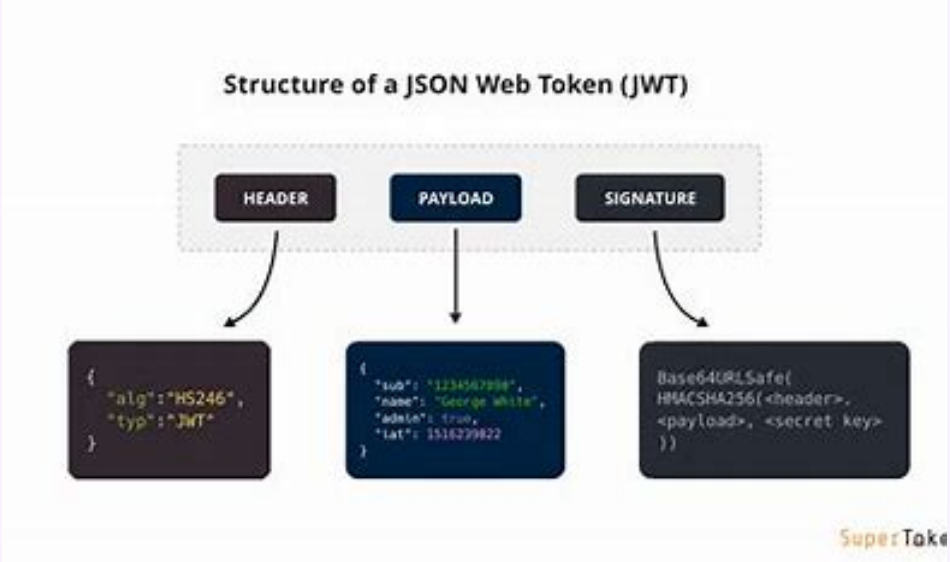
A JWT is made up of three parts, separated by dots (`.`):

```
Header.Payload.Signature
```

Example:

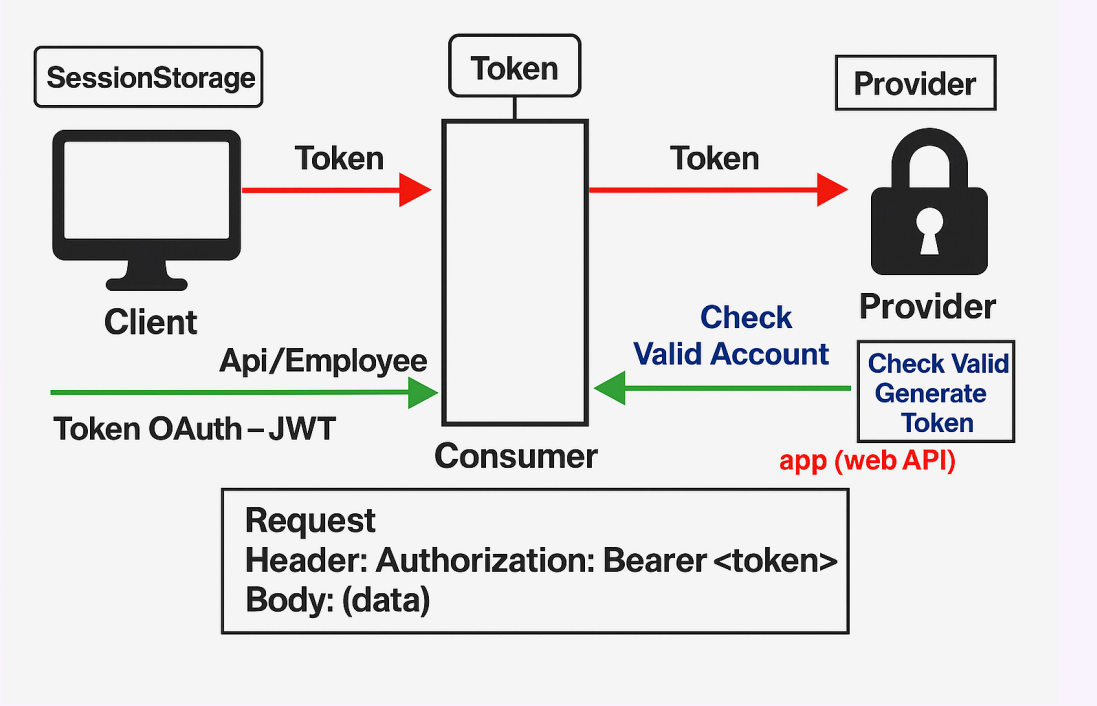
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IkpvaG5Eb2UiLCJyb2xlIjoiRG9jdG9yIiwiZXhwIjoxNjg5ODg5ODg4fQ.4PjR9Kfs7zHJxI4snk6L2DrpXn3_4VBNIddcGGUV9Qk
```

1. **Header**: Specifies the algorithm used (`HS256`) and the type of token (`JWT`).
2. **Payload**: Contains the claims or data, like `username` , `role` ,Init Id, and `expiration` .
3. **Signature**: Created by encoding the header and payload, then signing them using a secret key.



✅How JWT Works (Flow Between Client, Consumer, and Provider)

1. **Client (Browser/Angular App):**
   - Sends login credentials (username and password) to the **Provider (Web API)**.
   - The Provider validates the credentials and generates a **JWT Token**.
   - The token is sent back to the client and stored (usually in `SessionStorage` or `LocalStorage`).
2. **Consumer (Middleware/Service):**
   - When the client makes requests to protected resources (`/Api/Employees`), the token is sent along with the request in the **Header** (Authorization: `Bearer <token>`).
   - The Consumer (like Angular app middleware) forwards the request to the **Provider** for validation.
3. **Provider (Web API):**
   - **Validates the Token** by:
     - Checking the signature to ensure the token is genuine.
     - Checking the token is not expired.
     - Checking claims like `role` if needed.
   - If valid, grants access to the requested resource and returns the response to the client.
   - If invalid, returns an error message (`401 Unauthorized`).



✅ Why Use JWT?

- **Stateless Authentication:** No need to store user sessions on the server.
- **Scalability:** Since tokens are self-contained, they can be verified without accessing a database.
- **Security:** Tokens can be signed and encrypted, making them tamper-proof.

🔑 Summary

- **Token:** A piece of data used for authentication.
- **JWT:** A type of token that is compact, self-contained, and secure.
- **Flow:** Client requests a token, uses it for authentication, and the Provider validates it on each request.

🔑✅ What are the advantages of JWT?

1. **Compact**: JWTs are small in size and can be sent via URLs, HTTP headers, or POST requests. This makes them fast to transmit and easy to use.
2. **Self-Contained**: They include all necessary user information (like username, role, etc.) directly within the token. This means there's no need to query the database every time, making it faster and more efficient.
3. **Digitally Signed**: JWTs are signed using a secret key (HMAC) or a public/private key pair (RSA or ECDSA). This ensures the data is authentic and hasn't been tampered with.
4. **Transfer Information Between Bodies**: JWTs can securely transfer data between two parties, such as a client and server.
5. **Broad Compatibility**: JWTs are supported in many programming languages, making them easy to use across different systems.
6. **Secure Signing Methods**: JWTs can use hashing algorithms (HMAC) or more secure signing methods (RSA, ECDSA) for extra protection.

💡 Why do we get a 404 Not Found instead of 401 Unauthorized when using `[Authorize]` on an API endpoint?

✅ The Problem:

When you apply `[Authorize]` on an API endpoint and try to access it **without being authenticated**, sometimes you get a **404 Not Found** instead of the expected **401 Unauthorized**.

✅ The Reason:

1. By default, **ASP.NET Core** assumes you're using **cookie-based authentication** (used in MVC apps).
2. When you're not authenticated, it tries to **redirect you to the login page**, typically:

   `/Account/Login` the exists end point is `api/Account/Login`
3. But in API projects using **JWT**, there is **no such login page**.
4. If the system tries to redirect you to `/Account/Login` and it doesn't exist, you get a:

   ❌ **404 Not Found**

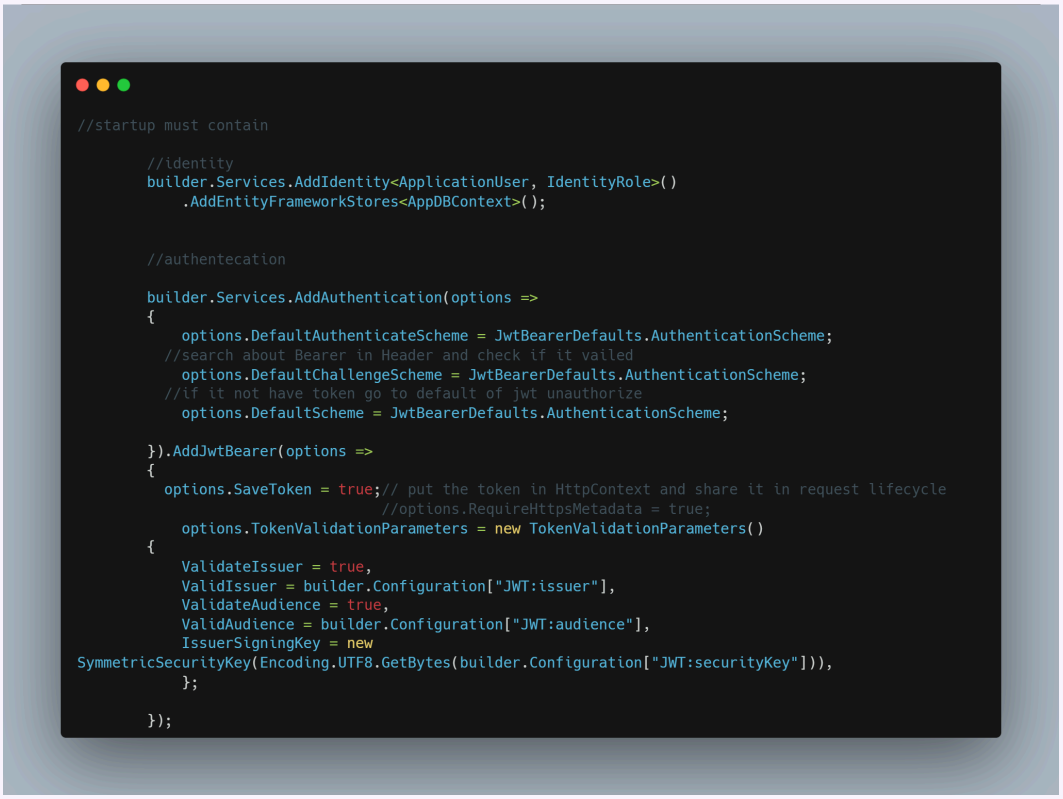
So it's **not really an authorization issue**, it's a redirection issue caused by the default behavior.

🔑 How can Implement JWT Security ?

```
namespace API_1.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AcountController : ControllerBase
    {
        private readonly UserManager<ApplicationUser> userManager;
        private readonly IConfiguration configuration;

        public AcountController(UserManager<ApplicationUser> userManager, IConfiguration configuration)
        {
            this.userManager = userManager;
            this.configuration = configuration;
        }


        [HttpPost("Register")]
        public async Task<IActionResult> Register(RegisterDto UserFromRegister)
        {
            if (ModelState.IsValid)
            {
                ApplicationUser user = new ApplicationUser
                {
                    UserName = UserFromRegister.UserName,
                    Email = UserFromRegister.Email
                };

                IdentityResult result = await userManager.CreateAsync(user, UserFromRegister.Password);


                if (result.Succeeded)
                {
                    return Created();

                }

                foreach (var error in result.Errors)
                {
                    if(error.Code.Contains("Password"))
                    {
                        ModelState.AddModelError("Password", error.Description);
                    }
                    else if(error.Code.Contains("UserName"))
                    {
                        ModelState.AddModelError("UserName", error.Description);

                    }
                    else if (error.Code.Contains("Email"))
                    {
                        ModelState.AddModelError("Email", error.Description);

                    }
                    else
                    {
                        ModelState.AddModelError(string.Empty, error.Description);

                    }
                }


                return ValidationProblem(ModelState);
            }


            return BadRequest(ModelState);

        }
    }
```

```
        [HttpPost("Login")]
        public async Task<IActionResult> Login(LoginDto requestfromlogin)
        {
            if(ModelState.IsValid)
            {
                ApplicationUser user =await userManager.FindByEmailAsync(requestfromlogin.Email);

                if (user != null) {

                    var isfound = await userManager.CheckPasswordAsync(user, requestfromlogin.Password);

                    if (isfound)
                    {

                        List<Claim> claims = new List<Claim>();
                        claims.Add(new Claim(ClaimTypes.NameIdentifier, user.Id));
                        claims.Add(new Claim(ClaimTypes.Name, user.UserName));


                        claims.Add(new Claim("aud", "http://my-react-app.com/"));//to add new audience

                        IList<string> roles = await userManager.GetRolesAsync(user);

                        foreach (var role in roles)
                        {
                            claims.Add(new Claim(ClaimTypes.Role, role));
                        }

                        claims.Add(new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()));


                        //security key
                        SymmetricSecurityKey securityKey = new SymmetricSecurityKey(Encoding.UTF8
                            .GetBytes(configuration["JWT:securityKey"]!));



                        //signature
                        SigningCredentials signingCredentials =
                            new SigningCredentials(securityKey, algorithm :
SecurityAlgorithms.HmacSha256);




                        //full design of  token
                        JwtSecurityToken token = new JwtSecurityToken(

                            issuer: configuration["JWT:issuer"],
                            audience: configuration["JWT:audience"],
                            expires: DateTime.Now.AddHours(2),
                             claims: claims,
                             signingCredentials: signingCredentials

                        );

                        var NewToken = new JwtSecurityTokenHandler().WriteToken(token);

                        var expire = DateTime.Now.AddHours(2);




                        var mytoken = new Generate_Token(NewToken, expire);
                        return Ok(mytoken);


                    }

                    ModelState.AddModelError("UserName","UserName Or Password Invalid");
                }

            }

            return BadRequest(ModelState);
        }
    }
}
```

```
//appsettingjson contain
 "JWT": {
   "issuer": "http://localhost:5030/",
   "audience": "http://localhost:4200/",
   "securityKey": "HcRq327^zvxga575hqgF16DKJshdvcaqe4efef"

 }
```



```
//startup must contain

        //identity
        builder.Services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<AppDBContext>();


        //authentecation

        builder.Services.AddAuthentication(options =>
        {
            options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
          //search about Bearer in Header and check if it vailed
            options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
          //if it not have token go to default of jwt unauthorize
            options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

        }).AddJwtBearer(options =>
        {
          options.SaveToken = true;// put the token in HttpContext and share it in request lifecycle
                                   //options.RequireHttpsMetadata = true;
            options.TokenValidationParameters = new TokenValidationParameters()
        {
            ValidateIssuer = true,
            ValidIssuer = builder.Configuration["JWT:issuer"],
            ValidateAudience = true,
            ValidAudience = builder.Configuration["JWT:audience"],
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["JWT:securityKey"])),
            };

        });
```

Show this post to understand Mapper

💡 Mapper (Manual ,Auto ,Mapster ,Mapperly):

https://www.linkedin.com/posts/ahmed-hany-899a9a321_%D8%A7%D9%84%D8%B3%D9%84%D8%A7%D9%85-%D8%B9%D9%84%D9%8A%D9%83%D9%85-%D9%87%D9%86%D8%AA%D9%83%D9%84%D9%85-%D8%B9%D9%86-%D9%85%D9%88%D8%B6%D9%88%D8%B9-%D9%83%D9%84%D9%86%D8%A7-%D8%A8%D9%86%D8%AD%D8%AA%D8%A7%D8%AC%D9%88-activity-7324131852556926976-awBG?utm_source=share&utm_medium=member_android&rcm=ACoAAFF_KXUB_4sXiwxlEmO-rsGzGT4Ol4j2cLU

Logging

🛠️

🔍 What is Logging?

✅ Definition:

Logging is the process of **recording events, actions, or errors** that occur while your application is running.

Think of it as your application's **diary**. Every time something important happens, it writes it down.

💡 Why Do We Use Logging?

Logging is **crucial** for several reasons:

1. Error Diagnosis

When something breaks in production, logs tell you:

- What happened?
- When?
- Where?
- Why?

2. Performance Monitoring

Track how long operations take, and spot performance issues.

3. User Behavior Analysis

Understand how users are using your app by logging actions like:

- Page visits
- Form submissions
- API requests

4. Security

Track login attempts, suspicious actions, and unauthorized access.

5. Maintenance & Debugging

Developers and support teams can understand how the system behaves without stepping through the code manually.

🌟 Introducing Serilog

What is Serilog?

Serilog is a **powerful, flexible, and structured logging library** for .NET applications.

It allows:

- Writing logs to **files, databases, cloud services**
- **Structured Logging** (logs are stored as data, not just text)
- Easy integration with **third-party tools** like Seq, Kibana

---

✅ When Should You Use Serilog?

Scenario	Use Serilog?
You want to log to a file	✅
You need logs in a structured format	✅
You want to analyze logs using tools	✅
You're building a large/production app	✅
Small/basic app with minimal logging	❌ ILogger is fine

🧪 Step-by-Step: Using Serilog in ASP.NET Core

Packages we will use

```
dotnet add package Serilog
dotnet add package Serilog.Sinks.Console
dotnet add package Serilog.Sinks.File
dotnet add package Serilog.Enrichers.Environment
dotnet add package Serilog.Enrichers.Thread
```

Tools to user for logging

1. full information in file with message and location of Error
2. Message only in Console (Warning)
3. in Debug in Development Environment

```
public static class LogExeptions
{
    public static void LogEx(Exception ex)
    {
        LogFile($"Error: {ex.Message}\nStackTrace: {ex.StackTrace}");
        LogConsole($"Error: {ex.Message}");
        LogDebug($"Full Error: {ex}");
    }

    private static void LogConsole(string message)  =>  Log.Warning(message);
    private static void LogFile(string message) => Log.Information(message);
    private static void LogDebug(string message) => Log.Debug(message);

}
```

Configurations

💡

```
public static void AddConfigurationLog(string filename, string environmentName)
{
    // Set up log directory
    var logDirectory = Path.Combine(Directory.GetCurrentDirectory(), "Logs");
    Directory.CreateDirectory(logDirectory); // Ensure directory exists

    // Configure logger
    Log.Logger = new LoggerConfiguration()
        .MinimumLevel.Debug() // Base level
        .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
        .MinimumLevel.Override("System", LogEventLevel.Warning)
        .MinimumLevel.Override("Microsoft.AspNetCore", LogEventLevel.Warning)
        .Enrich.FromLogContext()
        .Enrich.WithProperty("Environment", environmentName)
        .Enrich.WithMachineName()
        .Enrich.WithProcessId()
        .Enrich.WithThreadId()
        .WriteTo.Console(
            outputTemplate: "[{Timestamp:HH:mm:ss} {Level:u3}] " +
                            "(ReqId: {RequestId}, User: {Username}, " +
                            "Env: {Environment}) {Message:lj}{NewLine}{Exception}",
            theme: AnsiConsoleTheme.Code,
            restrictedToMinimumLevel: environmentName == "Development"
                ? LogEventLevel.Debug
                : LogEventLevel.Information
        )
        .WriteTo.File(
            path: Path.Combine(logDirectory, ${filename}-.log"),
            outputTemplate: "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} " +
                            "[{Level:u3}] (ReqId: {RequestId}, " +
                            "User: {UserId}|{Username}, Role: {Role}, " +
                            "Machine: {MachineName}, Thread: {ThreadId}) " +
                            "{Message:lj}{NewLine}{Exception}",
            rollingInterval: RollingInterval.Day,
            retainedFileCountLimit: 30, // Keep logs for 30 days
            shared: true,
            restrictedToMinimumLevel: LogEventLevel.Information
        )
        .WriteTo.Debug(restrictedToMinimumLevel: LogEventLevel.Debug)
        .CreateLogger();

    Log.Information("Logger initialized in {Environment} environment", environmentName);
}
```

Let’s Analyses :

1. Minimum Level Configuration in Debug

```
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Debug() // Base level
    .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
    .MinimumLevel.Override("System", LogEventLevel.Warning)
    .MinimumLevel.Override("Microsoft.AspNetCore", LogEventLevel.Warning)
```

> It logs everything from your app but only warnings and errors from Microsoft, System, and ASP.NET Core to reduce noise in the logs.

2. Important Things we want

```
.Enrich.FromLogContext()
.Enrich.WithProperty("Environment", environmentName)
.Enrich.WithMachineName()
.Enrich.WithProcessId()
.Enrich.WithThreadId()
```

🔑 → Adds values (like UserId, RequestId...) from the current HTTP context to each log automatically.

→ Adds the current environment name (Development or Production) to every log line.

→ Logs the name of the machine running the app.

→ Adds the process ID to help trace which app process wrote the log.

→ Adds the thread ID to help debug multi-threaded issues.

3. Logs in Console

```
.WriteTo.Console(
    outputTemplate: "[{Timestamp:HH:mm:ss} {Level:u3}] " +
                    "(ReqId: {RequestId}, User: {Username}, " +
                    "Env: {Environment}) {Message:lj}{NewLine}{Exception}",
    theme: AnsiConsoleTheme.Code,
    restrictedToMinimumLevel: environmentName == "Development"
        ? LogEventLevel.Debug
        : LogEventLevel.Information
)
```

🔑 → Writes logs to the console (Terminal or Output window), and uses colors for better visibility.

→ If you are in **Development**, show **Debug+ logs**.

→ If in **Production**, show only **Information level and above** to reduce noise.

4. Logs in File

```
.WriteTo.File(
    path: Path.Combine(logDirectory, $"{filename}-.log"),
    outputTemplate: "{Timestamp:yyyy-MM-dd HH:mm:ss.fff zzz} " +
                    "[{Level:u3}] (ReqId: {RequestId}, " +
                    "User: {UserId}|{Username}, Role: {Role}, " +
                    "Machine: {MachineName}, Thread: {ThreadId}) " +
                    "{Message:lj}{NewLine}{Exception}",
    rollingInterval: RollingInterval.Day,
    retainedFileCountLimit: 30, // Keep logs for 30 days
    shared: true,
    restrictedToMinimumLevel: LogEventLevel.Information
)
```

Output
2025-05-16 22:40:17.123 +02:00 [INF]
(ReqId: abc123, User: 1|Ali, Role: Doctor, Machine: LAPTOP-XYZ, Thread: 4)
Something happened!

🔑

🔷 `rollingInterval: RollingInterval.Day`

- Creates a **new log file every day**.

🔷 `retainedFileCountLimit: 30`

- Keeps the **last 30 log files** (older files will be deleted automatically).

🔷 `shared: true`

- Allows **multiple processes** to write to the same log file **safely**.

🔷 `restrictedToMinimumLevel: LogEventLevel.Information`

- Only logs with level **Information or higher** (Warning, Error) will be saved.
- **Debug logs will be ignored** to keep the file clean in production.

Middleware

```
using Microsoft.AspNetCore.Http;
using Serilog;
using Serilog.Context;
using System;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;

public class RequestLoggingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger _logger;

    public RequestLoggingMiddleware(RequestDelegate next, ILogger logger)
    {
        _next = next;
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var startTime = Stopwatch.GetTimestamp();
        var requestId = context.TraceIdentifier;

        // Extract user information
        var userId = context.User?.Claims?
            .FirstOrDefault(c => c.Type == "sub" || c.Type == "UserId")?
            .Value ?? "Anonymous";

        var username = context.User?.Identity?.IsAuthenticated == true
            ? context.User.Identity.Name
            : "Anonymous";

        var role = context.User?.Claims?
            .FirstOrDefault(c => c.Type == "role")?
            .Value ?? "None";

        // Push properties to log context
        using (LogContext.PushProperty("RequestId", requestId))
        using (LogContext.PushProperty("UserId", userId))
        using (LogContext.PushProperty("Username", username))
        using (LogContext.PushProperty("Role", role))
        {
            try
            {
                // Log request start
                _logger.Information(
                    "Starting request {Method} {Path} from {RemoteIpAddress}",
                    context.Request.Method,
                    context.Request.Path,
                    context.Connection.RemoteIpAddress);

                await _next(context);

                // Calculate elapsed time
                var elapsedMs = GetElapsedMilliseconds(startTime, Stopwatch.GetTimestamp());

                // Log successful completion
                _logger.Information(
                    "Completed request {Method} {Path} with {StatusCode} in {ElapsedMs}ms",
                    context.Request.Method,
                    context.Request.Path,
                    context.Response.StatusCode,
                    elapsedMs);
            }
            catch (Exception ex)
            {
                var elapsedMs = GetElapsedMilliseconds(startTime, Stopwatch.GetTimestamp());

                // Log the error
                _logger.Error(ex,
                    "Request {Method} {Path} failed after {ElapsedMs}ms with error: {ErrorMessage}",
                    context.Request.Method,
                    context.Request.Path,
                    elapsedMs,
                    ex.Message);

                throw; // Re-throw to let the error handling middleware handle it
            }
        }
    }

    private static double GetElapsedMilliseconds(long start, long stop)
    {
        return (stop - start) * 1000 / (double)Stopwatch.Frequency;
    }
}
```

🔑 Sure! Here's a concise breakdown of the important parts:

1. **Constructor**

   Initializes middleware with the next delegate and Serilog logger.

2. **InvokeAsync(HttpContext context)**

   Main method called on each HTTP request.

3. **Start Time & Request ID**

   Records the request start time and gets a unique request ID.

4. **Extract User Info**

   Gets user ID, username, and role from the current user claims; defaults if anonymous.

5. **Push Properties to Log Context**

   Adds request and user info to the Serilog context to include in all logs during this request.

6. **Log Request Start**

   Logs an information message when the request starts (method, path, IP).

7. **Call Next Middleware**

   Passes control to the next middleware in the pipeline (handles the actual request).

8. **Calculate Elapsed Time**

   Measures how long the request took.

9. **Log Request Completion**

   Logs success info including status code and elapsed time.

10. **Catch Exceptions**

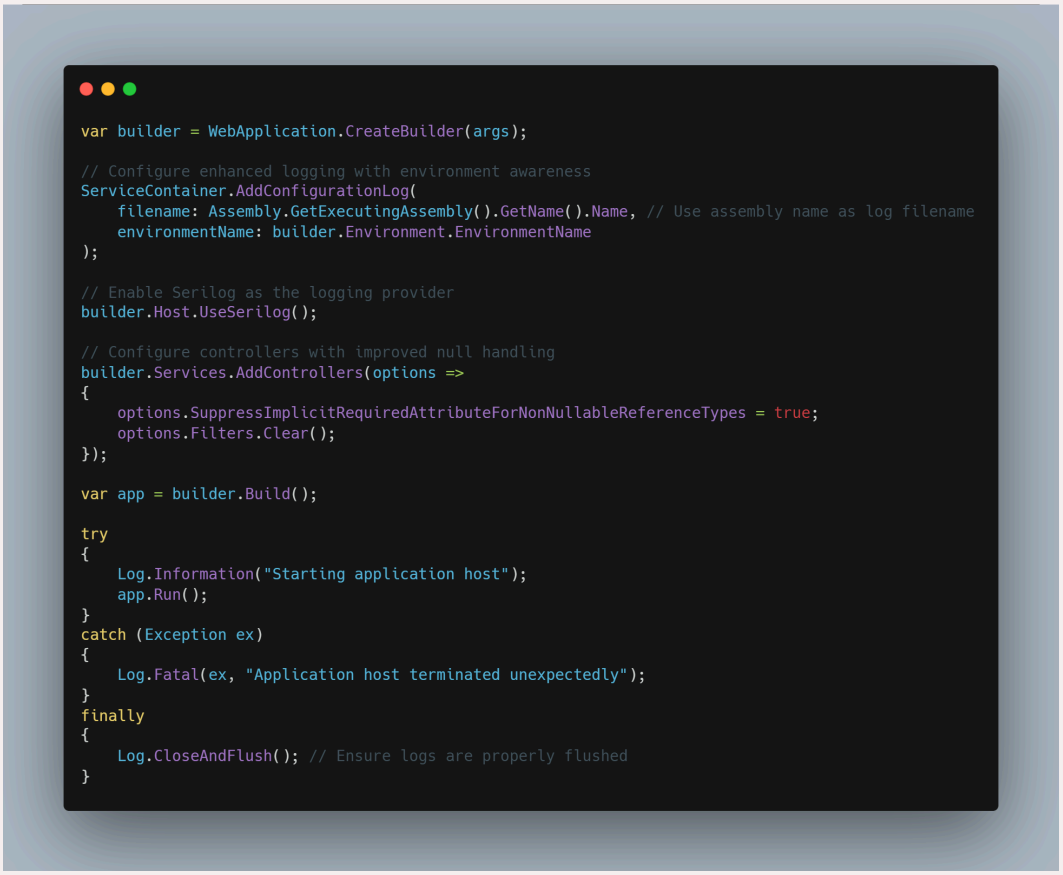
    If an error occurs, logs the error with details and rethrows to let other middleware handle it.

11. **GetElapsedMilliseconds**

    Helper method to calculate the request duration in milliseconds.

That covers all key steps without extra details!

Setting of Start Up



```
var builder = WebApplication.CreateBuilder(args);

// Configure enhanced logging with environment awareness
ServiceContainer.AddConfigurationLog(
    filename: Assembly.GetExecutingAssembly().GetName().Name, // Use assembly name as log filename
    environmentName: builder.Environment.EnvironmentName
);

// Enable Serilog as the logging provider
builder.Host.UseSerilog();

// Configure controllers with improved null handling
builder.Services.AddControllers(options =>
{
    options.SuppressImplicitRequiredAttributeForNonNullableReferenceTypes = true;
    options.Filters.Clear();
});

var app = builder.Build();

try
{
    Log.Information("Starting application host");
    app.Run();
}
catch (Exception ex)
{
    Log.Fatal(ex, "Application host terminated unexpectedly");
}
finally
{
    Log.CloseAndFlush(); // Ensure logs are properly flushed
}
```

MediatR

💡

What is MediatR?

MediatR is a popular open-source .NET library that implements the **Mediator pattern**, providing an in-process messaging mechanism. It was created by Jimmy Bogard and helps to decouple components in an application by acting as a "middleman" for communication between objects.

Key Features of MediatR:

1. **Implements the Mediator pattern**: Objects communicate through the mediator rather than directly with each other
2. **Supports two main messaging patterns**:
   - Request/Response (commands/queries)
   - Notifications (events)
3. **Minimal dependencies**: Simple abstraction over messaging
4. **Pipeline behaviors**: Allows adding cross-cutting concerns

Benefits of Using MediatR:

1. **Reduced Coupling**:
   - Components don't need references to each other
   - Only need to know about the mediator and message contracts
2. **Simplified Code**:
   - Removes the need for service locators or dependency injection of many services
   - Each handler has a single responsibility
3. **Improved Testability**:
   - Handlers can be tested in isolation
   - Easy to mock the mediator in unit tests
4. **Pipeline Behaviors**:
   - Add cross-cutting concerns like logging, validation, caching in one place
   - Similar to middleware in ASP.NET Core
5. **Clear Architecture**:
   - Enforces CQRS (Command Query Responsibility Segregation) pattern
   - Makes commands and queries explicit
6. **Scalability**:

- Easy to transition from in-process to out-of-process messaging later
- Good foundation for more complex architectures

Common Use Cases:

- Implementing CQRS pattern in applications
- Handling domain events in domain-driven design
- Reducing dependencies between controllers and business logic
- Adding cross-cutting concerns uniformly

MediatR is particularly popular in ASP.NET Core applications and works well with other patterns like Clean Architecture or Vertical Slice Architecture.

Would you like me to explain any specific aspect of MediatR in more detail?

Let’s see an Example :

💡 Here's a complete example demonstrating how to implement CRUD operations using:

- Repository pattern
- Interface abstraction
- MediatR for command/query handling

1. Setup the Project Structure

First, ensure you have MediatR installed:

```
dotnet add package MediatR
dotnet add package MediatR.Extensions.Microsoft.DependencyInjection
```

2. Define the Domain Model

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
}
```

3. Create the Repository Interface

```
public interface IProductRepository
{
    Task<Product> GetByIdAsync(int id);
    Task<IEnumerable<Product>> GetAllAsync();
    Task AddAsync(Product product);
    Task UpdateAsync(Product product);
    Task DeleteAsync(int id);
}
```

4. Implement the Repository (Entity Framework Core Example)

```
public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _context;
```

```
    public ProductRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task<Product> GetByIdAsync(int id)
    {
        return await _context.Products.FindAsync(id);
    }

    public async Task<IEnumerable<Product>> GetAllAsync()
    {
        return await _context.Products.ToListAsync();
    }

    public async Task AddAsync(Product product)
    {
        await _context.Products.AddAsync(product);
        await _context.SaveChangesAsync();
    }

    public async Task UpdateAsync(Product product)
    {
        _context.Products.Update(product);
        await _context.SaveChangesAsync();
    }

    public async Task DeleteAsync(int id)
    {
        var product = await GetByIdAsync(id);
        if (product != null)
        {
            _context.Products.Remove(product);
            await _context.SaveChangesAsync();
        }
    }
}
```

5. Create MediatR Commands and Queries

Queries:

```
// Get all products
public class GetAllProductsQuery : IRequest<IEnumerable<Product>> { }

public class GetAllProductsQueryHandler : IRequestHandler<GetAllProductsQuery, IEnumerable<Product>>
{
    private readonly IProductRepository _repository;

    public GetAllProductsQueryHandler(IProductRepository repository)
    {
        _repository = repository;
    }

    public async Task<IEnumerable<Product>> Handle(GetAllProductsQuery request, CancellationToken cancellationToken)
    {
        return await _repository.GetAllAsync();
    }
}

// Get product by ID
public class GetProductByIdQuery : IRequest<Product>
{
    public int Id { get; set; }
}

public class GetProductByIdQueryHandler : IRequestHandler<GetProductByIdQuery, Product>
{
    private readonly IProductRepository _repository;

    public GetProductByIdQueryHandler(IProductRepository repository)
    {
        _repository = repository;
    }

    public async Task<Product> Handle(GetProductByIdQuery request, CancellationToken cancellationToken)
    {
        return await _repository.GetByIdAsync(request.Id);
    }
}
```

Commands:

```
// Create product
public class CreateProductCommand : IRequest<int>
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
}

public class CreateProductCommandHandler : IRequestHandler<CreateProductComma
nd, int>
{
    private readonly IProductRepository _repository;

    public CreateProductCommandHandler(IProductRepository repository)
    {
        _repository = repository;
    }

    public async Task<int> Handle(CreateProductCommand request, CancellationToken
cancellationToken)
    {
        var product = new Product
        {
            Name = request.Name,
            Price = request.Price,
            Description = request.Description
        };

        await _repository.AddAsync(product);
        return product.Id;
    }
}

// Update product
public class UpdateProductCommand : IRequest
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
}
```

```
public class UpdateProductCommandHandler : IRequestHandler<UpdateProductComm
and>
{
    private readonly IProductRepository _repository;

    public UpdateProductCommandHandler(IProductRepository repository)
    {
        _repository = repository;
    }

    public async Task Handle(UpdateProductCommand request, CancellationToken can
cellationToken)
    {
        var product = await _repository.GetByIdAsync(request.Id);
        if (product != null)
        {
            product.Name = request.Name;
            product.Price = request.Price;
            product.Description = request.Description;

            await _repository.UpdateAsync(product);
        }
    }
}

// Delete product
public class DeleteProductCommand : IRequest
{
    public int Id { get; set; }
}

public class DeleteProductCommandHandler : IRequestHandler<DeleteProductComma
nd>
{
    private readonly IProductRepository _repository;

    public DeleteProductCommandHandler(IProductRepository repository)
    {
        _repository = repository;
    }

    public async Task Handle(DeleteProductCommand request, CancellationToken canc
```

```
ellationToken)
    {
        await _repository.DeleteAsync(request.Id);
    }
}
```

6. Configure Dependency Injection (in Program.cs)

```
builder.Services.AddScoped<IProductRepository, ProductRepository>();
builder.Services.AddMediatR(cfg ⇒ cfg.RegisterServicesFromAssembly(typeof(Progra
m).Assembly));
```

7. Use in API Controller

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly IMediator _mediator;

    public ProductsController(IMediator mediator)
    {
        _mediator = mediator;
    }

    // GET: api/products
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Product>>> GetAll()
    {
        var products = await _mediator.Send(new GetAllProductsQuery());
        return Ok(products);
    }

    // GET: api/products/5
    [HttpGet("{id}")]
    public async Task<ActionResult<Product>> GetById(int id)
    {
        var product = await _mediator.Send(new GetProductByIdQuery { Id = id });
        if (product == null) return NotFound();
        return Ok(product);
    }
```

```
    // POST: api/products
    [HttpPost]
    public async Task<ActionResult<int>> Create(CreateProductCommand command)
    {
        var productId = await _mediator.Send(command);
        return CreatedAtAction(nameof(GetById), new { id = productId }, productId);
    }

    // PUT: api/products/5
    [HttpPut("{id}")]
    public async Task<IActionResult> Update(int id, UpdateProductCommand comman
d)
    {
        if (id != command.Id) return BadRequest();
        await _mediator.Send(command);
        return NoContent();
    }

    // DELETE: api/products/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(int id)
    {
        await _mediator.Send(new DeleteProductCommand { Id = id });
        return NoContent();
    }
}
```

Key Benefits in This Implementation:

1. **Separation of Concerns**: Each handler has a single responsibility
2. **Testability**: Easy to mock repositories and test handlers independently
3. **Clean Controllers**: Controllers only deal with HTTP concerns, not business logic
4. **Flexibility**: Can easily add pipeline behaviors (logging, validation, caching)
5. **Maintainability**: Commands and queries are explicit and discoverable

Polly

💡

What is Polly?

Polly is a .NET **resilience and transient-fault-handling library** that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner. It's designed to help applications handle transient failures when calling external services, databases, or other dependencies.

Key Features of Polly:

1. **Resilience Policies**:
   - Retry (with various strategies)
   - Circuit Breaker
   - Timeout
   - Bulkhead Isolation
   - Cache
   - Fallback
2. **Policy Combinations**:
   - Wrap multiple policies together (e.g., Retry inside a Circuit Breaker)
3. **Async Support**:
   - Full support for asynchronous operations
4. **DI Integration**:
   - Works well with .NET's dependency injection system

Benefits of Using Polly:

1. Improved Application Resilience

- Automatically handles transient faults (network issues, timeouts)
- Prevents cascading failures in distributed systems

2. Simplified Error Handling

- Declarative policy definition (what to do) separate from execution (how to do it)
- Reduces boilerplate try-catch code

3. Protects Against Common Cloud/Network Issues

- Handles temporary network blips

- Manages throttling from cloud services

4. Prevents System Overload

- Circuit Breaker stops calling failing services
- Bulkhead limits concurrent calls to prevent resource exhaustion

5. Better User Experience

- Fallback policies can provide graceful degradation
- Automatic retries can make temporary issues invisible to users

6. Performance Benefits

- Cache policy avoids repeated expensive calls
- Smart retry strategies minimize wait times

Example Use Cases:

1. Retry failed HTTP requests to external APIs
2. Break the circuit when a microservice is down
3. Timeout long-running database queries
4. Limit concurrent calls to a legacy system
5. Provide fallback content when a recommendation service is unavailable

Example:

1. If All resources have the same need

```
services.AddDbContext<MyDbContext>(options ⇒
    options.UseSqlServer(connectionString, sqlOptions ⇒
        sqlOptions.EnableRetryOnFailure(
            maxRetryCount: 3,
            maxRetryDelay: TimeSpan.FromSeconds(5),
            errorNumbersToAdd: null) // ممكن تحدد اكواد الخطأ اللي عايز retry عليها
    ));
```

2. If you want to apply a specific Configuration

```
var retryPolicy = Policy
    .Handle<SqlException>(ex ⇒ IsTransient(ex)) // شرط إن الخطأ مؤقت (مثلاً Deadlock)
```

```
        .WaitAndRetry(3, retryAttempt => TimeSpan.FromSeconds(2));

    public async Task<T> ExecuteWithRetryAsync<T>(Func<Task<T>> action)
    {
        return await retryPolicy.ExecuteAsync(action);
    }




    public async Task<Patient> GetPatientAsync(int id)
    {
        return await ExecuteWithRetryAsync(async () =>
        {
            return await _dbContext.Patients.FindAsync(id);
        });
    }
```

When to Use Polly:

- Calling external HTTP APIs
- Database operations
- Microservice communications
- Any I/O operation that might fail temporarily
- Cloud service integrations (AWS, Azure, etc.)

Polly is particularly valuable in:

- Microservices architectures
- Cloud-native applications
- Systems with many external dependencies
- Applications where high availability is important

💡 متنسانيش بدعوة حلوة يا صديقي 💫