ASP.NETWeb API

.NET Core

Dot Net Full Stack Developer

Interview Questions and Answers

Sai Reddy 8/4/25 Interview Tips

Dot Net Full Stack Developer Interview Q&A

ASP.NET Core Web API Interview Questions

Q1: What is ASP.NET Core? How is it different from ASP.NET MVC?

A: ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, internet-connected applications, including Web APIs, web apps, and microservices.

Key Differences from ASP.NET MVC:

Feature	ASP.NET MVC	ASP.NET Core
Platform	Windows only	Cross-platform (Windows, macOS, Linux)
Web Server	IIS only	Kestrel, IIS, Nginx, Apache
DI	Third-party libraries	Built-in Dependency Injection
Hosting	System.Web	Lightweight self-hosting
Performance	Moderate	Much faster (benchmarked)
Modular	Monolithic	Highly modular via middleware

Q2: What are the main components of ASP.NET Core Web API?

A:

1. **Controllers** – Define API endpoints and business logic
2. **Routing** – Maps URL paths to controller actions
3. **Middleware** – Request pipeline components (e.g., logging, authentication)
4. **Dependency Injection (DI)** – Built-in DI system for service management
5. **Configuration** – appsettings.json, environment variables, etc.
6. **Model Binding and Validation** – Binds request data to .NET objects and validates
7. **Filters** – Logic before/after actions (e.g., Authorization, Logging)
8. **Swagger** – For API documentation
9. **Hosting** – Using Kestrel or IIS

Q3: What is middleware in ASP.NET Core?

A: Middleware is a component in the HTTP request pipeline that handles requests and responses.

Each middleware can:

- Inspect/modify incoming requests
- Pass request to the next middleware (next)
- Inspect/modify outgoing responses

Example:

```
public class LoggingMiddleware
{
   private readonly RequestDelegate _next;
```

```
    public LoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        Console.WriteLine("Request: " + context.Request.Path);
        await _next(context);
        Console.WriteLine("Response: " + context.Response.StatusCode);
    }
}

// In Program.cs
app.UseMiddleware<LoggingMiddleware>();
Built-in middleware includes UseRouting, UseAuthentication, UseAuthorization,
UseEndpoints, etc.
```

Q4: How do you create a simple Web API endpoint?

A: Create a controller that inherits from ControllerBase

1. Add [ApiController] and [Route] attributes
2. Add methods with [HttpGet], [HttpPost], etc.

Example:

```
[ApiController]
[Route("api/[controller]")]
public class HelloController : ControllerBase
{
    [HttpGet("greet")]
    public IActionResult GetGreeting()
    {
        return Ok("Hello from API!");
    }
}
```

Call: GET /api/hello/greet → Output: "Hello from API!"

Q5: What is the use of Startup.cs and Program.cs in .NET 6+?

A: In **.NET 6 and later**, Program.cs uses the **minimal hosting model**, replacing Startup.cs.

In .NET 5 and below:

- Startup.cs → Contains ConfigureServices (DI) and Configure (pipeline setup)
- Program.cs → Entry point using CreateHostBuilder

In .NET 6+:

- Everything is in Program.cs

Example:

```
var builder = WebApplication.CreateBuilder(args);
// Register services
builder.Services.AddControllers();
var app = builder.Build();
// Configure middleware
app.UseRouting();
app.MapControllers();
app.Run();
```

Q6: What is dependency injection and how is it implemented in ASP.NET Core?

A: Dependency Injection (DI) is a design pattern to inject dependencies (services) rather than creating them manually.

ASP.NET Core has built-in support for DI.

Registering:

```
builder.Services.AddScoped<IMyService, MyService>();
```

Injecting:

```
public class HomeController : ControllerBase
{
   private readonly IMyService _service;

   public HomeController(IMyService service)
   {
      _service = service;
   }

   [HttpGet]
   public IActionResult GetData()
   {
      return Ok(_service.GetData());
   }
}
```

Q7: Explain the difference between IActionResult and ActionResult<T>.

A:

Feature	IActionResult	ActionResult
Return Type	Any response result (Ok, NotFound, etc.)	Combines result with a specific type
Type Safety	Not strongly typed	Strongly typed

Example:

```
// IActionResult
[HttpGet]
public IActionResult Get()
{
   return Ok("Hello");
}

// ActionResult<T>
[HttpGet]
public ActionResult<string> GetTyped()
{
   return "Hello";
}
```

Use ActionResult<T> for better type safety and Swagger support.

Q8: How does routing work in ASP.NET Core?

A: Routing maps incoming HTTP requests to controller actions.

- Uses [Route], [HttpGet("path")], or conventional routing.
- MapControllers() enables attribute-based routing.

Example:

```
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
   [HttpGet("{id}")]
   public IActionResult GetById(int id)
   {
      return Ok($"Product with ID = {id}");
   }
}
```

- URL GET /api/products/5 → Calls GetById(5)

Q9: What are the HTTP verbs and how are they used in Web API?

A:

Verb	Purpose	ASP.NET Core Attribute	Typical Use Case
GET	Retrieve a resource	[HttpGet]	Read data (e.g., GET /products/1)
POST	Create a new resource	[HttpPost]	Add data (e.g., POST /products)
PUT	Update/replace a resource	[HttpPut]	Replace full resource (e.g., PUT /products/1)

Verb	Purpose	ASP.NET Core Attribute	Typical Use Case
PATCH	Partial update of a resource	[HttpPatch]	Update part of a resource (e.g., name only)
DELETE	Delete a resource	[HttpDelete]	Remove data (e.g., DELETE /products/1)
OPTIONS	Describe communication options	[HttpOptions]	Used in CORS preflight or client introspection
HEAD	Retrieve headers only (no body)	[HttpHead]	Check if a resource exists
TRACE	Echo the received request	[HttpTrace] (rarely used)	Debugging client-server communication
CONNECT	Establish a tunnel (e.g., HTTPS)	Not handled by ASP.NET Core	Used by proxies (not in APIs directly)

Example:

```
[HttpPost]
public IActionResult Create(Product product)
{
   // Save to DB
   return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

Q10: What is model binding and model validation in ASP.NET Core?

A:

- Model Binding – Automatically maps request data (JSON, query, form) to .NET objects.
- Model Validation – Validates model using [Required], [StringLength], etc.

Example:

```
public class UserDto
{
   [Required]
   public string Name { get; set; }

   [Range(18, 99)]
   public int Age { get; set; }
}
```

```
[HttpPost]
public IActionResult Register(UserDto user)
{
   if (!ModelState.IsValid)
      return BadRequest(ModelState);

   return Ok("User created");
}
If request body is:
{
 "Name": "",
 "Age": 10
}
It will return:
{
 "errors": {
   "Name": ["The Name field is required."],
   "Age": ["The field Age must be between 18 and 99."]
 }
}
```

Q11: What is the role of appsettings.json and how do you read configuration values?

A:

appsettings.json is a configuration file in ASP.NET Core used to store settings like connection strings, API keys, logging levels, etc. It supports hierarchical and environment-specific settings via files like appsettings.Development.json.

Example (appsettings.json):

```
{
 "AppSettings": {
   "AppName": "My API",
   "MaxItems": 100
 },
 "ConnectionStrings": {
   "DefaultConnection": "Server=.;Database=MyDb;Trusted_Connection=True;"
 }
}
```

Reading values in code (from DI):

```
public class MySettings
{
   public string AppName { get; set; }
```

```
    public int MaxItems { get; set; }
}

// Register in Program.cs
builder.Services.Configure<MySettings>(builder.Configuration.GetSection("AppSettings"));

// Inject and use
public class MyService
{
    private readonly MySettings _settings;

    public MyService(IOptions<MySettings> options)
    {
        _settings = options.Value;
    }

    public string GetAppName() => _settings.AppName;
}
```

Q12: How do you implement logging in ASP.NET Core?

A:

ASP.NET Core has built-in support for structured logging through ILogger<T>, with providers like Console, Debug, Azure, Serilog, etc.

Example:

```
public class MyController : ControllerBase
{
    private readonly ILogger<MyController> _logger;
    public MyController(ILogger<MyController> logger)
    {
        _logger = logger;
    }

    [HttpGet]
    public IActionResult Get()
    {
        _logger.LogInformation("Processing GET request");
        return Ok("Logged");
    }
}
```

Configure Logging in Program.cs:

builder.Logging.ClearProviders();

builder.Logging.AddConsole();

Logging Levels: Trace, Debug, Information, Warning, Error, Critical.

Q13: Explain filters in ASP.NET Core: Authorization, Resource, Action, Exception, Result filters.

A:

Filters in ASP.NET Core allow code to run before or after specific stages of request processing.

- **Authorization Filter**: Runs first, checks user's permission. ([Authorize])
- **Resource Filter**: Runs before model binding. Useful for caching, performance.
- **Action Filter**: Runs before and after controller action. (OnActionExecuting)
- **Exception Filter**: Handles unhandled exceptions. (IExceptionFilter)
- **Result Filter**: Runs before and after the response result is executed.

Custom Action Filter:

```
public class LogActionFilter : IActionFilter
{
   public void OnActionExecuting(ActionExecutingContext context)
   {
      Console.WriteLine("Before action");
   }

   public void OnActionExecuted(ActionExecutedContext context)
   {
      Console.WriteLine("After action");
   }
}
```

Register filter globally:

```
builder.Services.AddControllers(options =>
{
   options.Filters.Add<LogActionFilter>();
});
```

Q14: What is Swagger/OpenAPI and how do you configure it in ASP.NET Core?

A:

Swagger (OpenAPI) generates interactive documentation for your API. It helps consumers understand and test the API.

Steps to configure:

1. Install NuGet package: Swashbuckle.AspNetCore
2. In Program.cs:

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var app = builder.Build();
app.UseSwagger();
app.UseSwaggerUI();
```

Run and access: /swagger/index.html

You can add XML comments to methods for better documentation:

```
/// <summary>
/// Gets all items.
/// </summary>
[HttpGet]
public IActionResult GetAll() => Ok();
```

Q15: How do you handle exceptions globally in ASP.NET Core Web API?

A: Use a global exception-handling middleware to capture all unhandled exceptions and return consistent error responses.

Example middleware:

```
public class ExceptionMiddleware
{
   private readonly RequestDelegate _next;
   private readonly ILogger<ExceptionMiddleware> _logger;

   public ExceptionMiddleware(RequestDelegate next, ILogger<ExceptionMiddleware>
logger)
   {
      _next = next;
      _logger = logger;
   }

   public async Task Invoke(HttpContext context)
   {
      try
      {
         await _next(context);
      }
      catch (Exception ex)
      {
         _logger.LogError(ex, "Unhandled Exception");
         context.Response.StatusCode = 500;
```

```
            await context.Response.WriteAsJsonAsync(new { message = "Something went
wrong." });
        }
    }
}
```

Register in Program.cs:

```
app.UseMiddleware<ExceptionMiddleware>();
```

Q16: What are DTOs and why are they important in Web APIs?

A:

DTO (Data Transfer Object) is a simplified object used to transfer data between layers or over the network, often between API and clients.

Why use DTOs:

- Avoid exposing database entities
- Customize data per client need
- Improve security and performance
- Control serialization

Example:

```
public class UserDto
{
    public string Name { get; set; }
    public string Email { get; set; }
}
```

Mapping from entity:

```
var dto = new UserDto
{
    Name = user.Name,
    Email = user.Email
};
```

You can use libraries like AutoMapper to simplify this mapping.

Q17: Explain CORS and how to enable it in ASP.NET Core.

A:

CORS (Cross-Origin Resource Sharing) allows your API to be accessed from a different domain than it's hosted on (e.g., frontend app on localhost:3000 calling API on localhost:5000).

Enable CORS in Program.cs:

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowFrontend", policy =>
    {
```

```
        policy.WithOrigins("http://localhost:3000")
            .AllowAnyMethod()
            .AllowAnyHeader();
    });
});

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

Common Use Case: React or Angular frontend calling .NET Web API.

Q18: How does model validation work with Data Annotations and FluentValidation?

A:

Data Annotations are attributes applied to model properties.

Example:

```
public class Product
{
    [Required]
    public string Name { get; set; }

    [Range(1, 1000)]
    public decimal Price { get; set; }
}
```

Validation on controller:

```
[HttpPost]
public IActionResult Add(Product product)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    return Ok();
}
```

FluentValidation is a powerful external library.

Setup:

1. Install FluentValidation.AspNetCore
2. Define validator:

```
public class ProductValidator : AbstractValidator<Product>
{
    public ProductValidator()
    {
        RuleFor(x => x.Name).NotEmpty();
        RuleFor(x => x.Price).InclusiveBetween(1, 1000);
```

```
    }
}
```

3. Register in Program.cs:

```
builder.Services.AddFluentValidation(fv =>
fv.RegisterValidatorsFromAssemblyContaining<ProductValidator>());
```

Q19: How can you version a Web API?

A:

ASP.NET Core supports multiple API versioning strategies via the Microsoft.AspNetCore.Mvc.Versioning package.

Install and configure:

```
builder.Services.AddApiVersioning(options =>
{
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.DefaultApiVersion = new ApiVersion(1, 0);
    options.ReportApiVersions = true;
});
```

Use version in controller:

```
[ApiController]
[Route("api/v{version:apiVersion}/products")]
[ApiVersion("1.0")]
public class ProductsV1Controller : ControllerBase
{
    [HttpGet]
    public IActionResult Get() => Ok("V1 data");
}
```

You can version via:

- URL path (/api/v1)
- Query string (?api-version=1.0)
- Header (api-version: 1.0)

Q20: How do you secure a Web API using JWT Authentication?

A:

JWT (JSON Web Token) authentication is a stateless way to secure APIs using tokens signed by a secret key.

Steps:

1. Add NuGet: Microsoft.AspNetCore.Authentication.JwtBearer
2. Configure authentication in Program.cs:

```
builder.Services.AddAuthentication("Bearer")
    .AddJwtBearer("Bearer", options =>
    {
```

```
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "myapi",
            ValidAudience = "myclient",
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("your_secret_key"))
        };
    });

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

3. Protect controller:

```
[Authorize]
[HttpGet]
public IActionResult GetSecretData() => Ok("This is protected data");
```

4. Issue JWT token on login:

```
var token = new JwtSecurityToken(
    issuer: "myapi",
    audience: "myclient",
    expires: DateTime.Now.AddHours(1),
    signingCredentials: new SigningCredentials(
        new SymmetricSecurityKey(Encoding.UTF8.GetBytes("your_secret_key")),
        SecurityAlgorithms.HmacSha256)
);
```

You pass the token in requests as:

Authorization: Bearer <your_jwt_token>

Q21: What is the difference between AddScoped, AddSingleton, and AddTransient services?

A:

These methods define the **lifetime** of a service in the built-in DI container:

- AddSingleton: Creates a single instance for the entire application lifetime. Same object for all requests.
- AddScoped: Creates one instance per **HTTP request**. Shared across classes during that request.
- AddTransient: Creates a new instance **every time** it's requested.

Example:

```
builder.Services.AddSingleton<IService, Service>();   // Same for all users
builder.Services.AddScoped<IService, Service>();      // One per request
builder.Services.AddTransient<IService, Service>();   // New each time
```

Q22: How do you implement role-based and policy-based authorization?

A:

Role-based Authorization uses [Authorize(Roles = "Admin")] to restrict access based on user roles.

Policy-based Authorization defines rules (like age or claims) using custom policies.

In Program.cs:

```
builder.Services.AddAuthorization(options =>
{
   options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));
   options.AddPolicy("MinAge", policy => policy.RequireClaim("age", "18"));
});
```

Usage in controllers:

```
[Authorize(Policy = "AdminOnly")]
public IActionResult AdminTask() => Ok("Admin Access");

[Authorize(Roles = "Manager")]
public IActionResult ManagerTask() => Ok("Manager Access");
```

Q23: Explain rate limiting and how to implement it in ASP.NET Core Web API.

A:

Rate limiting prevents abuse by restricting how many requests a client can make in a time window.

Install NuGet: AspNetCoreRateLimit

In Program.cs:

```
builder.Services.AddMemoryCache();
builder.Services.Configure<IpRateLimitOptions>(builder.Configuration.GetSection("IpRateLimiting"));
builder.Services.AddInMemoryRateLimiting();
builder.Services.AddSingleton<IRateLimitConfiguration, RateLimitConfiguration>();
```

In appsettings.json:

```
"IpRateLimiting": {
  "EnableEndpointRateLimiting": true,
  "StackBlockedRequests": false,
  "RealIpHeader": "X-Real-IP",
  "GeneralRules": [
   {
```

```
    "Endpoint": "*",
    "Period": "1m",
    "Limit": 5
   }
 ]
}
```

Use middleware:

app.UseIpRateLimiting();

Q24: What is the purpose of the HttpClientFactory and how is it used?

A:

HttpClientFactory helps manage the lifetime of HttpClient instances and avoids socket exhaustion.

Benefits:

- Reuses handlers
- Manages DNS changes
- Supports Polly for retries, circuit breakers

Register in Program.cs:

```
builder.Services.AddHttpClient("external", client =>
{
   client.BaseAddress = new Uri("https://api.example.com/");
});
```

Inject and use:

```
public class MyService
{
   private readonly HttpClient _httpClient;
   public MyService(IHttpClientFactory factory)
   {
      _httpClient = factory.CreateClient("external");
   }

   public async Task<string> GetData() =>
      await _httpClient.GetStringAsync("endpoint");
}
```

Q25: How can you improve the performance of your ASP.NET Core API?

A:

- **Use Asynchronous Code**: Prefer async/await with I/O operations
- **Response Caching**: Cache frequent GET responses
- **Output Compression**: Enable Gzip compression using app.UseResponseCompression()

- **Minimize Database Calls**: Use efficient queries and indexing
- **Use Paging**: Avoid returning large datasets
- **Caching**: Use memory or distributed caching
- **Use HttpClientFactory**: For efficient external API calls
- **Connection Pooling**: With Dapper or EF Core
- **Minimize Middleware**: Keep request pipeline light
- **Profiling**: Use tools like Application Insights or MiniProfiler

Q26: What are background services (IHostedService / BackgroundService) and how do you use them?

A:

IHostedService and BackgroundService allow running background tasks independently of HTTP requests.

Example using BackgroundService:

```
public class MyWorker : BackgroundService
{
   protected override async Task ExecuteAsync(CancellationToken stoppingToken)
   {
      while (!stoppingToken.IsCancellationRequested)
      {
         Console.WriteLine("Doing background work...");
         await Task.Delay(5000);
      }
   }
}
```

Register in Program.cs:

```
builder.Services.AddHostedService<MyWorker>();
```

Q27: What is gRPC and how is it different from REST APIs?

A:

gRPC is a high-performance, contract-first, binary protocol built on HTTP/2, using **Protobuf** for serialization.

Differences:

Feature	REST API	gRPC
Protocol	HTTP/1.1	HTTP/2
Data Format	JSON	Protobuf (binary)
Performance	Slower	Faster
Streaming	Limited	Supports bi-directional streaming
Contract	OpenAPI	.proto files

Use Cases: Internal microservice communication, real-time apps, low-latency scenarios

Q28: How do you unit test controllers and services in ASP.NET Core Web API?

A:

Use **xUnit/NUnit**, **Moq**, and **TestServer** to test.

Example Controller:

```
public class ProductsController : ControllerBase
{
   private readonly IProductService _service;
   public ProductsController(IProductService service)
   {
      _service = service;
   }

   [HttpGet("{id}")]
   public IActionResult Get(int id)
   {
      var product = _service.Get(id);
      if (product == null) return NotFound();
      return Ok(product);
   }
}
```

Unit Test using Moq:

```
[Fact]
public void Get_ReturnsProduct_WhenExists()
{
   var mock = new Mock<IProductService>();
   mock.Setup(s => s.Get(1)).Returns(new Product { Id = 1 });

   var controller = new ProductsController(mock.Object);
   var result = controller.Get(1) as OkObjectResult;

   Assert.NotNull(result);
   Assert.Equal(200, result.StatusCode);
}
```

Q29: How do you implement caching in ASP.NET Core?

A:

ASP.NET Core supports:

- **In-memory caching**: IMemoryCache
- **Distributed caching**: Redis, SQL Server

In-memory example:

```
public class MyService
{
   private readonly IMemoryCache _cache;

   public MyService(IMemoryCache cache)
   {
      _cache = cache;
   }

   public string GetData()
   {
      return _cache.GetOrCreate("key", entry =>
      {
         entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10);
         return "Cached data";
      });
   }
}
```

Register:

builder.Services.AddMemoryCache();

Q30: What are minimal APIs in .NET 6/7/8 and how are they different from traditional Web APIs?

A:

Minimal APIs allow defining endpoints with **less boilerplate**, ideal for small or microservices apps.

Differences:

Traditional Web API	Minimal API
Uses Controllers	Uses route handlers directly
Attributes like [HttpGet]	Uses MapGet, MapPost, etc.
More ceremony	Lightweight and fast to build

Example Minimal API:

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

app.MapGet("/hello", () => "Hello World");

app.Run();
```

Ideal for: lightweight services, fast development, fewer dependencies.

Q31: How do you design a highly scalable and resilient Web API architecture in ASP.NET Core?

A:

To design a scalable and resilient API:

- **Use Load Balancers**: Distribute traffic across instances (e.g., Azure Front Door, AWS ALB)
- **Horizontal Scaling**: Deploy APIs behind auto-scaling containers or VMs (Kubernetes, App Service)
- **Stateless Services**: Avoid session state on server, use JWT for auth
- **Use Caching**: In-memory and distributed cache (Redis) to reduce DB hits
- **Async Programming**: Use async/await to free threads for more requests
- **Retry & Timeout Policies**: Use Polly for transient fault handling
- **Circuit Breakers**: Prevent cascading failures
- **Health Checks**: Use AddHealthChecks() and expose /health endpoint
- **Database Scalability**: Read replicas, partitioning, connection pooling
- **Monitoring**: Use App Insights, Prometheus, ELK for observability

Q32: What are best practices for designing RESTful APIs?

A:

- **Use Nouns in URLs**: /api/products not /getProducts
- **Use HTTP Verbs Properly**: GET for fetch, POST for create, PUT for update
- **Version Your API**: Via URL, header, or query (e.g., /api/v1/products)
- **Return Proper Status Codes**: 200 OK, 201 Created, 400 BadRequest, 404 NotFound
- **Use DTOs**: Never expose EF entities directly
- **Paginate Large Results**: Include page, pageSize in query
- **Filter/Sort**: Use query params /products?sort=name&filter=active
- **Idempotency**: Ensure PUT/DELETE operations are idempotent
- **Include HATEOAS (if needed)**: Hypermedia links for discoverability
- **Secure Endpoints**: Use HTTPS, JWT, Role/Policy-based auth

Q33: How do you handle multi-tenancy in ASP.NET Core applications?

A:

Multi-tenancy allows a single app to serve multiple customers (tenants). Approaches:

- **Single DB, Shared Schema**: Add TenantId to each table
- **Single DB, Separate Schemas**: Each tenant has own schema
- **Separate DB per Tenant**: Strong isolation, higher cost

Implementation:

- **Tenant Resolution**: From subdomain, header, token, or request
- **Middleware**: Identify tenant and store in scoped context

```
public class TenantMiddleware
{
```

```
    public async Task Invoke(HttpContext context, ITenantService tenantService)
    {
        var tenantId = context.Request.Headers["X-Tenant-ID"];
        tenantService.SetTenant(tenantId);
        await _next(context);
    }
}
```

- **DbContext Customization**: Use OnModelCreating to filter data per tenant
- **Separate Config/Logging** per tenant if needed

Q34: Explain how to implement CQRS and MediatR in an ASP.NET Core Web API.

A:

CQRS (Command Query Responsibility Segregation) separates read and write operations:

- **Commands**: Mutate data (CreateUserCommand)
- **Queries**: Fetch data (GetUserByIdQuery)

Install MediatR NuGet

```
builder.Services.AddMediatR(typeof(Startup));
```

Define Command:

```
public record CreateUserCommand(string Name) : IRequest<Guid>;

public class CreateUserHandler : IRequestHandler<CreateUserCommand, Guid>
{
    public Task<Guid> Handle(CreateUserCommand request, CancellationToken ct)
    {
        var id = Guid.NewGuid(); // Save user logic
        return Task.FromResult(id);
    }
}
```

Controller:

```
[HttpPost]
public async Task<IActionResult> Create(CreateUserCommand cmd)
{
    var id = await _mediator.Send(cmd);
    return Ok(id);
}
```

Benefits: Clean separation of logic, testability, scalability

Q35: What are custom middleware and when should you write one?

A:

Custom middleware lets you execute logic during request/response pipeline.

Write one when you need:

- Request/response logging
- Multi-tenancy resolution
- Exception handling
- Request transformations
- Feature toggles

Example:

```
public class TimingMiddleware
{
   private readonly RequestDelegate _next;
   public async Task Invoke(HttpContext context)
   {
      var sw = Stopwatch.StartNew();
      await _next(context);
      sw.Stop();
      Console.WriteLine($"Request took {sw.ElapsedMilliseconds} ms");
   }
}
```

Register in Program.cs:

app.UseMiddleware<TimingMiddleware>();

Q36: How do you handle database transactions in ASP.NET Core with EF Core or Dapper?

A:

With EF Core:

```
using var transaction = await _db.Database.BeginTransactionAsync();
try
{
   _db.Users.Add(user);
   _db.Orders.Add(order);
   await _db.SaveChangesAsync();
   await transaction.CommitAsync();
}
catch
{
   await transaction.RollbackAsync();
   throw;
}
```

With Dapper and ADO.NET:

```
using var conn = new SqlConnection(_connectionString);
conn.Open();
using var tx = conn.BeginTransaction();
```

```
try {
   conn.Execute("INSERT INTO Users...", transaction: tx);
   conn.Execute("INSERT INTO Orders...", transaction: tx);
   tx.Commit();
} catch {
   tx.Rollback();
   throw;
}
```

Always wrap multi-step DB operations in transactions to ensure atomicity.

Q37: How do you design a microservices-based system with ASP.NET Core APIs?

A: Design considerations:

- **Bounded Context**: Each microservice owns its domain and data
- **Independent Deployment**: Services deploy and scale separately
- **Communication**: Use REST for sync, messaging (RabbitMQ, Kafka) for async
- **Database per Service**: Avoid shared DBs
- **Service Discovery**: Use Consul or Kubernetes DNS
- **API Gateway**: Central entry point with routing, auth, caching
- **Health Checks**: Expose /health for monitoring
- **Security**: JWT, OAuth2, IdentityServer
- **Observability**: Logs, metrics, tracing per service

Example services: AuthService, OrderService, InventoryService, ProductService

Q38: Explain API Gateway and BFF patterns in enterprise systems.

A: **API Gateway** is a single entry point for client requests to microservices.

Responsibilities:

- Request routing
- Authentication
- Rate limiting
- Aggregation

Popular Gateways: Ocelot, YARP, Azure API Management, Kong

BFF (Backend for Frontend) is a gateway tailored for each frontend (web, mobile)

- Optimizes data for that UI
- Avoids over-fetching/under-fetching
- Reduces round-trips

Example:

- /mobile/orders (returns compact view)
- /web/orders (returns detailed view with filters)

Use API Gateway + BFF for clean, scalable frontend/backend integration.

Q39: How do you perform end-to-end security hardening of a public Web API?

A: **Use HTTPS only**: Redirect HTTP to HTTPS

- **Authentication**: Use JWT/OAuth2 with IdentityServer/Azure AD
- **Authorization**: Role/Policy-based access control
- **Input Validation**: Prevent SQL injection, XSS
- **Rate Limiting**: Prevent brute-force and DoS
- **CORS Configuration**: Only allow trusted origins
- **Logging and Alerts**: Log auth failures, block repeated attempts
- **Exception Handling**: Avoid leaking stack traces
- **Secrets Management**: Use environment variables or Azure Key Vault
- **Headers**: Add X-Content-Type-Options, Strict-Transport-Security, etc.
- **Content Security Policy**: Restrict resource loading

Pen-test APIs regularly and follow OWASP API Security Top 10.

Q40: What's your strategy for monitoring and observability (logs, metrics, tracing) in ASP.NET Core APIs?

A:

A good observability strategy includes:

- **Logging**: Use ILogger, Serilog, or NLog for structured logs
- **Metrics**: Track request count, duration, CPU/memory (Prometheus + Grafana)
- **Tracing**: Use OpenTelemetry or Application Insights for distributed tracing
- **Health Checks**: Add /health endpoint

```
builder.Services.AddHealthChecks().AddSqlServer(connStr);
app.MapHealthChecks("/health");
```

- **Exception Tracking**: Use Sentry, Raygun, or App Insights
- **Dashboards**: Grafana or Azure Monitor for real-time visualization
- **Alerts**: Auto-alert on 5xx errors, slow requests, memory spikes

Combine logs + metrics + traces for full-stack observability.