ASP.NET Core Web API Interview Questions by Karthik M

Q1. What is ASP.NET Core Web API?

Answer:
ASP.NET Core Web API is a lightweight, cross-platform framework for building RESTful services over HTTP.
It enables communication between different systems (like browsers, mobile apps, other servers) using standard HTTP verbs (GET, POST, PUT, DELETE).

Basic Controller:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet]
    public IEnumerable<Product> GetAll()
    {
        // Fetch products from database
        return productService.GetAll();
    }
}
```

Note: ASP.NET Core Web API uses ControllerBase (instead of Controller) because there are no Views, only data responses.

Q2. What is the difference between Controller and ControllerBase in ASP.NET Core?

Answer:

Controller	ControllerBase
Inherits from ControllerBase	Base class
Supports both Views (MVC) and APIs	Supports only APIs (no Views)
Used when building web applications with UI	Used for building Web APIs only

Example:

```
public class HomeController : Controller → For MVC (Views + APIs)
public class ProductController : ControllerBase → For pure APIs
```

Q3. What is the [ApiController] attribute?

Answer:
The [ApiController] attribute tells ASP.NET Core that the controller is used for Web API purposes.

Benefits:

a. Automatic Model Validation
b. Automatic 400 Bad Request response
c. Attribute Routing enforced
d. Binding source inference (from body, route, query)

Example:

```
[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase
```

```
{
    [HttpPost]
    public IActionResult CreateOrder(Order order)
    {
        // ModelState automatically validated
        return Ok();
    }
}
```

Note: Without [ApiController], you would need to manually check ModelState.IsValid.

Q4. What is routing in ASP.NET Core Web API?

Answer:
Routing is the mechanism by which incoming HTTP requests are matched to actions in controllers.

Attribute Routing Example:

```
[Route("api/products")]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}")]
    public IActionResult GetProductById(int id)
    {
        // Fetch product
        return Ok();
    }
}
```

Note: URL like api/products/5 would automatically map to the GetProductById action.

Q5. How do you return different HTTP Status Codes from a Controller?

Answer:
Use built-in helper methods like Ok(), BadRequest(), NotFound(), Created(), etc.

Example:

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

    return Ok(product);
}
```

Note: This ensures the client gets correct status codes (like 200, 404, etc.).

Q6. What is Dependency Injection (DI) and how is it used in Web API?

Answer:
Dependency Injection is a design pattern where dependencies (services) are injected into a class instead of the class creating them.

Register service in Program.cs:
```
builder.Services.AddScoped<IProductService, ProductService>();
```

Use constructor injection:
```
private readonly IProductService _service;
```

```
public ProductsController(IProductService service)
{
    _service = service;
}
```

Note: This makes your code loosely coupled, testable, and maintainable.

Q7. How do you return JSON from an API method?

Answer:
By default, ASP.NET Core Web API serializes your responses into JSON.

Example:

```
[HttpGet]
public IActionResult Get()
{
    var user = new { Name = "Karthik", Age = 30 };
    return Ok(user);
}
```

Note: No need for manual JSON serialization — it uses System.Text.Json (or Newtonsoft.Json optionally).

Q8. What is Model Binding in Web API?

Answer:
Model Binding maps incoming HTTP request data (route data, query string, body) into C# action parameters.

Example:

```
[HttpPost]
public IActionResult CreateProduct([FromBody] Product product)
{
    // product object populated automatically from JSON body
}
```

Binding Sources:

a. [FromRoute]
b. [FromQuery]
c. [FromBody]
d. [FromForm]
e. [FromHeader]

Q9. What is the difference between IActionResult and ActionResult<T>?

Answer:

IActionResult	ActionResult<T>
Can return any type of response	Returns a specific type along with HTTP status
Less type-safe	Type-safe

Example:

```
public IActionResult GetProduct(int id)
```

vs

public ActionResult<Product> GetProduct(int id)

Note: Using ActionResult<Product> allows you to return either a Product (200 OK) or other error codes cleanly.

Q10. How do you validate incoming request data in Web API?

Answer:

a. Use Data Annotations in Models (like [Required], [MaxLength], [Range]).
b. Use [ApiController] attribute to auto-check ModelState.
c. Handle invalid data gracefully.

Model:

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

    [Range(1, 10000)]
    public decimal Price { get; set; }
}
```

Controller:

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

    return Ok();
}
```

Note: In most cases, with [ApiController], you don't need to write ModelState.IsValid manually — it auto returns 400!

Q11. How do you secure a Web API using JWT (JSON Web Token)?

Answer:

1. Install NuGet package:
Microsoft.AspNetCore.Authentication.JwtBearer

2. Configure authentication in Program.cs:

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKey"))
        };
    });
```

3. Protect API endpoints:

```
[Authorize]
[HttpGet]
```

```
public IActionResult GetSecretData()
{
    return Ok("This is protected");
}
```

Note: The client must pass a valid JWT token in the Authorization header.

Q12. How can you enable CORS in Web API?

Answer:
CORS (Cross-Origin Resource Sharing) allows your Web API to be accessed from different domains.

Register CORS in Program.cs:

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

Add it to the HTTP pipeline:
```
app.UseCors("AllowAll");
```

Now your API can accept requests from any domain.
Important: For production, restrict origins instead of AllowAnyOrigin().

Q13. How do you implement API Versioning in ASP.NET Core?

Answer:

Install package:
Microsoft.AspNetCore.Mvc.Versioning

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

Use attributes:

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/products")]
public class ProductsV1Controller : ControllerBase
```

Now, you can manage multiple versions of your APIs cleanly.

Q14. What is IActionFilter and how to create a custom one?

Answer:
Action Filters run before and after an action method executes.

Custom filter:

```
public class LogActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("Action Starting: " + context.ActionDescriptor.DisplayName);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        Console.WriteLine("Action Ended: " + context.ActionDescriptor.DisplayName);
    }
}
```

Apply it:

```
[ServiceFilter(typeof(LogActionFilter))]
public class ProductsController : ControllerBase
```

Or globally in Program.cs.

Q15. How do you return custom error responses in Web API?

Answer:
You can use ProblemDetails:

```
return Problem(detail: "Invalid Request", statusCode: 400);
```

Or create a custom response:

```
return BadRequest(new { ErrorCode = 1001, ErrorMessage = "Product not found" });
```

Note: Customizing error responses improves client debugging and UX.

Q16. What is the difference between FromBody and FromQuery?

Answer:

Attribute	Purpose
[FromBody]	Bind data from HTTP body (JSON)
[FromQuery]	Bind data from URL query string

Example:

```
[HttpPost]
public IActionResult CreateProduct([FromBody] Product product)

[HttpGet]
public IActionResult GetProduct([FromQuery] int id)
```

Q17. How can you log information in ASP.NET Core Web API?

Answer:
Use built-in ILogger<T> service.

Inject logger:

```
private readonly ILogger<ProductsController> _logger;
public ProductsController(ILogger<ProductsController> logger)
{
    _logger = logger;
}

Use it:
_logger.LogInformation("Fetching all products");

Note: Logging supports multiple sinks like Console, File, Azure Monitor, etc.



Q18. How do you implement file upload in Web API?

Answer:
Accept files as IFormFile.

Controller:

[HttpPost("upload")]
public async Task<IActionResult> UploadFile(IFormFile file)
{
    if (file == null || file.Length == 0)
        return BadRequest("No file selected");

    var path = Path.Combine("uploads", file.FileName);

    using (var stream = new FileStream(path, FileMode.Create))
    {
        await file.CopyToAsync(stream);
    }

    return Ok("Uploaded Successfully");
}

Important: Ensure you configure large request body size if needed.



Q19. How do you enable Swagger (OpenAPI) in Web API?

Answer:

Install package:
Swashbuckle.AspNetCore

Configure in Program.cs:
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

Add middleware:
app.UseSwagger();
app.UseSwaggerUI();

Note: Run the app and open /swagger endpoint to see your API documentation
auto-generated!



Q20. What is the difference between 204 No Content and 200 OK?

Answer:

Status Code         Meaning
```

```
200 OK              Success, with content in response body
204 No Content      Success, but no content returned

Example:

return Ok(product); → Returns 200 OK with product data.
return NoContent(); → Returns 204 No Content (useful after Delete operations).
```

```
Q21. How do you implement global exception handling in ASP.NET Core Web API?

Answer:
Use a custom middleware to catch unhandled exceptions.

Example:
```

```
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 InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Unhandled Exception");
            context.Response.StatusCode = 500;
            await context.Response.WriteAsync("Internal Server Error");
        }
    }
}
```

```
Register it:
app.UseMiddleware<ExceptionMiddleware>();

Note: This way, your entire API has centralized error handling!
```

```
Q22. What is the use of [FromRoute], [FromQuery], [FromBody] attributes?

Answer:
These attributes tell ASP.NET Core explicitly where to bind the parameter data from.

Examples:
```

```
[HttpGet("{id}")]
public IActionResult GetProduct([FromRoute] int id)

[HttpGet("search")]
public IActionResult SearchProduct([FromQuery] string name)

[HttpPost]
public IActionResult AddProduct([FromBody] Product product)
```

Note: It improves clarity and prevents ambiguous binding.

Q23. How do you perform Dependency Injection for multiple services of the same interface?

Answer:
You can inject IEnumerable<T> to get all registered services.

Example:

```
builder.Services.AddTransient<INotificationService, EmailService>();
builder.Services.AddTransient<INotificationService, SmsService>();
```

Inject:

```
public class NotificationManager
{
    private readonly IEnumerable<INotificationService> _services;

    public NotificationManager(IEnumerable<INotificationService> services)
    {
        _services = services;
    }
}
```

Note: You can then iterate through services to call methods.

Q24. How can you restrict Web API to only HTTPS requests?

Answer:
Force HTTPS redirection:

```
builder.Services.AddHttpsRedirection(options =>
{
    options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
    options.HttpsPort = 5001;
});
```

Add middleware:

```
app.UseHttpsRedirection();
```

Note: Now your API will reject any HTTP calls and enforce HTTPS!

Q25. What is ModelState and when should you use it manually?

Answer:
ModelState contains the state of the model binding and validation process.

Normally with [ApiController], model validation errors are automatically handled.
If you are not using [ApiController], you must manually check:

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

Note: ModelState ensures only valid data enters your business logic.

Q26. How do you implement Rate Limiting in ASP.NET Core Web API?

Answer:
Use third-party libraries like AspNetCoreRateLimit.

Example basic rule:
Max 100 requests per 1 minute per client.

You configure it in appsettings.json and register services like:

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

Note: Protects your API from abuse.

Q27. What is Content Negotiation in ASP.NET Core Web API?

Answer:
Content negotiation means the server selects the best format (JSON, XML, etc.) based on what client accepts.

Example:

Request Header: Accept: application/xml

ASP.NET Core Web API will respond with XML if XML formatter is added:

```
builder.Services.AddControllers().AddXmlSerializerFormatters();
```

Note: Otherwise, JSON is the default.

Q28. How do you implement soft delete in Web API?

Answer:
Soft delete means marking the record as deleted without physically removing it.

Entity:

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

Instead of deleting:

```
product.IsDeleted = true;
dbContext.SaveChanges();
```

Query only non-deleted records:

```
dbContext.Products.Where(p => !p.IsDeleted);
```

Q29. How can you handle large file uploads efficiently in Web API?

Answer:

Increase limits:

```
builder.Services.Configure<FormOptions>(options =>
```

```
{
    options.MultipartBodyLengthLimit = 209715200; // 200 MB
});
```

Use streaming:
Instead of loading into memory, read stream directly.

```
using var stream = file.OpenReadStream();
using var destination = File.Create(path);
await stream.CopyToAsync(destination);
```

Note: This avoids memory overflow for large files.

Q30. How do you inject configuration settings into Controllers?

Answer:
Inject IConfiguration.

Example:

```
private readonly IConfiguration _config;
public ProductsController(IConfiguration config)
{
    _config = config;
}
```

Usage:

```
var connectionString = _config.GetConnectionString("DefaultConnection");
```

Note: You can access appsettings.json or environment variables easily!

Q31. How do you send custom headers from a Web API response?

Answer:
You can add custom headers to the HTTP response like this:

```
[HttpGet]
public IActionResult GetProduct()
{
    Response.Headers.Add("X-Custom-Header", "KarthikAPI");
    return Ok(new { Message = "Success" });
}
```

Note: Useful for sending metadata, version info, correlation IDs, etc.

Q32. How can you consume a Web API using HttpClient in .NET?

Answer:
Example client:

```
var client = new HttpClient();
var response = await client.GetAsync("https://localhost:5001/api/products");
if (response.IsSuccessStatusCode)
{
    var content = await response.Content.ReadAsStringAsync();
    Console.WriteLine(content);
}
```

Note: HttpClient is the standard way to call external APIs from your code.

Q33. What is middleware in ASP.NET Core?

Answer:
Middleware is software that's assembled into an app pipeline to handle requests and responses.

Example custom middleware:

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

    public HelloMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        await context.Response.WriteAsync("Hello from Middleware! ");
        await _next(context);
    }
}
```

Register it:

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

Q34. What are the different return types supported by Web API methods?

Answer:
Common return types:

a. IActionResult
b. ActionResult<T>
c. Task<IActionResult>
d. Task<ActionResult<T>>
e. Direct object (e.g., Product)
f. HttpResponse (advanced)

Example:

```
public ActionResult<Product> Get(int id)
```

Asynchronous:

```
public async Task<ActionResult<Product>> GetAsync(int id)
```

Q35. What is the difference between synchronous and asynchronous Web API methods?

Answer:

Synchronous	Asynchronous
Blocks the thread	Frees the thread
Less scalable under load	Highly scalable
Good for quick operations	Best for I/O, DB calls, file access

Example async method:

```
[HttpGet]
```

```
public async Task<IActionResult> GetProducts()
{
    var products = await _service.GetAllAsync();
    return Ok(products);
}
```

Note: Async/await should be preferred in production Web APIs!

Q36. How do you unit test a Web API Controller?

Answer:
Use Mocking frameworks like Moq + Xunit.

Example basic unit test:

```
var mockService = new Mock<IProductService>();
mockService.Setup(x => x.GetAll()).Returns(new List<Product> { new Product { Id = 1, Name = "Sample" } });

var controller = new ProductsController(mockService.Object);
var result = controller.GetAll();

Assert.IsType<OkObjectResult>(result);
```

Note: Always mock dependencies and test controller logic independently.

Q37. How do you use Token-based Authentication in Web API?

Answer:
Generate JWT Token after login:

```
var tokenHandler = new JwtSecurityTokenHandler();
var key = Encoding.ASCII.GetBytes("YourSecretKey");
var tokenDescriptor = new SecurityTokenDescriptor
{
    Subject = new ClaimsIdentity(new Claim[]
    {
        new Claim(ClaimTypes.Name, user.Id.ToString())
    }),
    Expires = DateTime.UtcNow.AddHours(1),
    SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
};
var token = tokenHandler.CreateToken(tokenDescriptor);
return tokenHandler.WriteToken(token);
```

Note: Clients must pass this token in the Authorization: Bearer <token> header for secured API calls.

Q38. What is Kestrel in ASP.NET Core?

Answer:
Kestrel is the built-in cross-platform web server for ASP.NET Core applications.

Features:

a. Lightweight, high-performance
b. Used as an edge server or behind reverse proxy (e.g., IIS, Nginx)
c. Can handle both HTTP/1.x and HTTP/2

Note: ASP.NET Core apps always run inside Kestrel, whether self-hosted or behind a proxy.

Q39. What is the difference between AddSingleton, AddScoped, and AddTransient services?

Answer:

Lifetime	Scope	Usage
Singleton	One instance for entire app	Configurations, stateless services
Scoped	One instance per HTTP request	Database contexts
Transient	New instance every time requested	Lightweight services

Example:

```
builder.Services.AddSingleton<IMyService, MyService>();
builder.Services.AddScoped<IMyDbContext, MyDbContext>();
builder.Services.AddTransient<ILogicHelper, LogicHelper>();
```

Note: Choosing the right lifetime impacts memory and performance.

Q40. How do you use Swagger to test secured APIs (with JWT Token)?

Answer:

Add security definition in Program.cs:

```
builder.Services.AddSwaggerGen(c =>
{
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        In = ParameterLocation.Header,
        Description = "Please enter JWT with Bearer into field",
        Name = "Authorization",
        Type = SecuritySchemeType.ApiKey
    });

    c.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference { Type = ReferenceType.SecurityScheme,
Id = "Bearer" }
            },
            new string[] { }
        }
    });
});
```

Note: Now, Swagger UI will show an Authorize button to paste the token. It Helps easily test secured endpoints during development!

Q41. What are Route Constraints in Web API and how do you use them?

Answer:
Route constraints restrict the format of parameters.

Example:

```
[HttpGet("product/{id:int}")]
public IActionResult GetProduct(int id)
{
    return Ok(id);
}
```

Other constraints: string, bool, datetime, guid, minlength, maxlength, etc.

Note: Helps validate route data before reaching the action method!

Q42. What is the difference between UseAuthorization and UseAuthentication middleware?

Answer:

Middleware	Purpose
UseAuthentication	Validates token or credentials
UseAuthorization	Checks user permissions after authentication

Order matters:

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

Note: Without proper order, authorization might fail unexpectedly.

Q43. How do you create a custom authorization policy in ASP.NET Core?

Answer:

Define policy in Program.cs:

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

Protect action:

```
[Authorize(Policy = "AdminOnly")]
[HttpGet("admin-data")]
public IActionResult GetAdminData()
{
    return Ok("Secret Data");
}
```

Note: You can create powerful fine-grained security controls using policies.

Q44. What are Minimal APIs introduced in .NET 6 and .NET 7?

Answer:
Minimal APIs allow creating Web APIs without Controllers.

Example:

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

```
var app = builder.Build();

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

Note: Great for small, lightweight APIs and microservices.

Q45. How do you bind complex types from the URL in Web API?

Answer:
Bind from query parameters:

```
[HttpGet]
public IActionResult Search([FromQuery] ProductFilter filter)
```

Where ProductFilter is:

```
public class ProductFilter
{
    public string Name { get; set; }
    public int? CategoryId { get; set; }
}
```

Note: Supports advanced filtering, sorting, and searching via query strings.

Q46. How can you handle circular references in Web API responses (like in EF Core)?

Answer:
Configure JSON options to ignore cycles:

```
builder.Services.AddControllers().AddJsonOptions(x =>
{
    x.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
});
```

Note: Prevents serialization errors like Self referencing loop detected when returning entity models.

Q47. What are Filters in ASP.NET Core Web API?

Answer:
Filters allow you to run code before or after key pipeline stages.

Types:

a. Authorization filters
b. Resource filters
c. Action filters
d. Exception filters
e. Result filters

Example Action Filter:

```
public class LogFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("Action is starting...");
    }
}
```

Apply:

```
[ServiceFilter(typeof(LogFilter))]
```

Q48. How do you configure multiple environments (Development, Staging, Production) in ASP.NET Core?

Answer:
Use launchSettings.json or set ASPNETCORE_ENVIRONMENT variable.

Access inside code:

```
if (env.IsDevelopment())
{
    // Enable detailed error page
}
else
{
    // Generic error page
}
```

You can inject IWebHostEnvironment anywhere:

```
private readonly IWebHostEnvironment _env;
public HomeController(IWebHostEnvironment env)
{
    _env = env;
}
```

Note: Enables environment-specific behavior easily.

Q49. How do you send validation error responses in a standard format?

Answer:
Custom middleware:

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

    public ValidationExceptionMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (ValidationException ex)
        {
            context.Response.StatusCode = 400;
            await context.Response.WriteAsJsonAsync(new { Error = ex.Message });
        }
    }
}
```

Note: Makes client-side error handling much easier.

Q50. What is a DTO and why should you use it in Web APIs?

Answer:
DTO = Data Transfer Object

Use DTOs to:

Hide internal domain models

Avoid exposing sensitive data

Customize API response shape

Improve performance

Example DTO:

```
public class ProductDto
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Controller:

```
[HttpGet("{id}")]
public ActionResult<ProductDto> Get(int id)
{
    var product = _context.Products.Find(id);
    if (product == null) return NotFound();

    return new ProductDto
    {
        Name = product.Name,
        Price = product.Price
    };
}
```

Note: DTOs are a best practice for secure, clean, and future-proof Web APIs.

Q51. How do you return a file (like PDF, Excel) from an API?

Answer:
Use File() helper method:

```
[HttpGet("download")]
public IActionResult DownloadFile()
{
    var bytes = System.IO.File.ReadAllBytes("files/sample.pdf");
    return File(bytes, "application/pdf", "downloaded_sample.pdf");
}
```

Note: Automatically sets the correct content type and download prompt.

Q52. What is the purpose of ProducesResponseType attribute?

Answer:
Documents possible HTTP responses.

Example:

```
[HttpGet("{id}")]
```

```
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IActionResult GetProduct(int id)
{
    ...
}
```

Note: It helps tools like Swagger generate better API documentation.

Q53. What is the difference between IHostedService and BackgroundService?

Answer:

Feature	IHostedService	BackgroundService
What it is	Interface	Abstract class
Control	You implement everything manually	Gives a ready-to-override ExecuteAsync
Use case	Cron jobs, event listeners	Long-running background tasks

Example of BackgroundService:

```
public class Worker : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            Console.WriteLine("Running...");
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

Q54. What is the purpose of UseEndpoints middleware?

Answer:
It maps HTTP routes to corresponding handlers (Controllers, Razor Pages, etc).

Example:

```
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

Note: Always needed to activate routing after authentication and authorization.

Q55. How do you validate an incoming model manually without [ApiController]?

Answer:
You must check ModelState manually:

```
[HttpPost]
public IActionResult AddProduct(Product product)
```

```
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    _context.Products.Add(product);
    _context.SaveChanges();

    return Ok();
}
```

Note: With [ApiController], validation happens automatically.

Q56. What is the purpose of IHttpContextAccessor?

Answer:
It allows you to access HttpContext outside of Controllers, for example inside Services.

Inject and use:

```
private readonly IHttpContextAccessor _httpContextAccessor;
public MyService(IHttpContextAccessor httpContextAccessor)
{
    _httpContextAccessor = httpContextAccessor;
}

var userId =
_httpContextAccessor.HttpContext?.User?.FindFirst(ClaimTypes.NameIdentifier)?.Value;
```

Note: Useful for accessing user claims, headers, session, etc., in deep layers.

Q57. How do you customize model validation error messages?

Answer:
Using Data Annotations:

```
public class Product
{
    [Required(ErrorMessage = "Product name is mandatory")]
    public string Name { get; set; }

    [Range(1, 10000, ErrorMessage = "Price must be between 1 and 10000")]
    public decimal Price { get; set; }
}
```

Note: Custom error messages make APIs much more user-friendly.

Q58. How do you create an API that returns pagination results?

Answer:
Standard pagination:

```
[HttpGet]
public IActionResult GetProducts(int page = 1, int pageSize = 10)
{
    var skip = (page - 1) * pageSize;
    var products = _context.Products.Skip(skip).Take(pageSize).ToList();
    return Ok(products);
}
```

Note: Always return total count and current page info in real APIs!

Q59. How do you implement Caching in ASP.NET Core Web API?

Answer:

1. Response Caching:

Add middleware:

```
app.UseResponseCaching();
```

Controller:

```
[HttpGet]
[ResponseCache(Duration = 60)]
public IActionResult GetProducts()
{
    return Ok(_service.GetProducts());
}
```

Note: Caches full HTTP response.

2. Memory Caching:

Register service:

```
builder.Services.AddMemoryCache();
```

Use:

```
private readonly IMemoryCache _cache;
public MyService(IMemoryCache cache)
{
    _cache = cache;
}

_cache.Set("key", value, TimeSpan.FromMinutes(10));
```

Q60. What is the default maximum request body size in ASP.NET Core?

Answer:

Default maximum size = 30 MB (for Kestrel).

For IIS, it’s controlled by maxRequestLength and maxAllowedContentLength.

You can increase it in Kestrel:

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxRequestBodySize = 104857600; // 100 MB
});
```

Note: Important when uploading files, images, large JSON payloads.

Q61. How do you implement Global Exception Handling in Web API?

Answer:
Use a custom middleware:

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

    public ExceptionMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            context.Response.StatusCode = 500;
            await context.Response.WriteAsJsonAsync(new { Error = ex.Message });
        }
    }
}

Register it:
app.UseMiddleware<ExceptionMiddleware>();

Note:  Centralizes error handling for your entire API.




Q62. How can you upload files using ASP.NET Core Web API?

Answer:
Controller method:

[HttpPost("upload")]
public async Task<IActionResult> UploadFile(IFormFile file)
{
    if (file == null || file.Length == 0)
        return BadRequest("No file selected");

    var path = Path.Combine("uploads", file.FileName);

    using (var stream = new FileStream(path, FileMode.Create))
    {
        await file.CopyToAsync(stream);
    }

    return Ok("File uploaded successfully");
}

Note: Don't forget to increase MaxRequestBodySize if needed!




Q63. How can you implement Soft Delete in a Web API with Entity Framework Core?

Answer:
Entity:

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

Soft delete method:

```
public IActionResult Delete(int id)
{
    var product = _context.Products.Find(id);
    if (product == null) return NotFound();

    product.IsDeleted = true;
    _context.SaveChanges();

    return NoContent();
}
```

Note: This way, you don't physically remove records — good for audit trails!

Q64. What is the difference between FromQuery, FromBody, and FromRoute attributes?

Answer:

Attribute	Source of data
[FromQuery]	URL query parameters
[FromBody]	HTTP request body
[FromRoute]	URL route parameters

Example:

```
[HttpPost("product/{id}")]
public IActionResult UpdateProduct([FromRoute] int id, [FromBody] Product product,
[FromQuery] string source)
```

Note: Understanding these helps handle multi-source data correctly!

Q65. How can you return a 201 Created response with a location header?

Answer:
Use CreatedAtAction().

Example:

```
[HttpPost]
public IActionResult Create(Product product)
{
    _context.Products.Add(product);
    _context.SaveChanges();

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

Note: Correct way to indicate resource creation according to REST standards!

Q66. What is the purpose of [BindProperty] attribute in ASP.NET Core?

Answer:
Primarily used in Razor Pages but can apply to APIs too.

It binds request values automatically:

```
[BindProperty]
public Product Product { get; set; }
```

In Razor Pages:

```
public async Task<IActionResult> OnPostAsync()
{
    _context.Products.Add(Product);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

Note: Simplifies data binding in page models.

Q67. What are Action Results and Non-Action Results in Web API?

Answer:

Action Results: Returns HTTP-specific responses (e.g., Ok(), NotFound(), BadRequest()).
Non-Action Results: Directly return object without wrapping (e.g., string, int).

Examples:

Action Result:

```
public ActionResult<Product> Get(int id)
```

Non-Action Result:

```
public Product GetProduct(int id)
```

Note: Action Results are more flexible and recommended for APIs!

Q68. How do you expose multiple versions of a Web API?

Answer:
Use API versioning.

Install package:

```
Microsoft.AspNetCore.Mvc.Versioning
```

Setup:

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

Controller:

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
```

Note: Supports version via URL, query string, headers!

Q69. How can you enable CORS (Cross-Origin Resource Sharing) globally?

Answer:

1. Configure CORS:

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

2. Apply:

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

Note: CORS is mandatory when APIs are called from different domains.

Q70. How do you pass complex objects in query strings safely?

Answer:
Serialize the object to JSON, then URL encode it.

Client-side:

```
let obj = { name: "karthik", age: 25 };
let queryString = encodeURIComponent(JSON.stringify(obj));
fetch(/api/users?data=${queryString});
```

Server-side:

```
[HttpGet]
public IActionResult GetUser([FromQuery] string data)
{
    var user = JsonSerializer.Deserialize<User>(data);
    return Ok(user);
}
```

Note: Not best practice for very large objects — prefer POST with body instead!

Q71. What is Dependency Injection in ASP.NET Core?

Answer:
Dependency Injection (DI) means giving an object what it needs (dependencies) instead of letting it create them.

Example:

```
public interface IMessageService
{
    string GetMessage();
}

public class EmailService : IMessageService
{
    public string GetMessage() => "Email sent!";
}
```

Controller:

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

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

    [HttpGet]
    public string Index() => _service.GetMessage();
}

Register service:
builder.Services.AddScoped<IMessageService, EmailService>();

Note: Promotes loose coupling and easy testing.



Q72. What are the different lifetimes for services in ASP.NET Core?

Answer:


Lifetime    Description
==========================================
Singleton   One instance for the entire app
Scoped      One instance per request
Transient   New instance every time injected


Examples:

builder.Services.AddSingleton<MyService>();
builder.Services.AddScoped<MyService>();
builder.Services.AddTransient<MyService>();

Note: Choose wisely — wrong lifetime can lead to bugs or memory leaks!



Q73. How can you protect APIs with API Key Authentication?

Answer:

1. Create a Middleware:

public class ApiKeyMiddleware
{
    private readonly RequestDelegate _next;
    private const string APIKEY = "X-API-KEY";

    public ApiKeyMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        if (!context.Request.Headers.TryGetValue(APIKEY, out var extractedApiKey))
        {
            context.Response.StatusCode = 401;
            await context.Response.WriteAsync("API Key missing");
            return;
```

```
        }

        if (extractedApiKey != "MySuperSecretApiKey")
        {
            context.Response.StatusCode = 403;
            await context.Response.WriteAsync("Invalid API Key");
            return;
        }

        await _next(context);
    }
}

2. Register:
app.UseMiddleware<ApiKeyMiddleware>();

Note: Light-weight security mechanism for private APIs.
```

Q74. How do you limit request rates (Throttling) in Web APIs?

Answer:
a. Use third-party middleware like:
b. AspNetCoreRateLimit
c. Custom Middleware

Basic example:

```
app.UseRateLimiter(new RateLimiterOptions
{
    PermitLimit = 5,
    Window = TimeSpan.FromMinutes(1)
});
```

Note: Prevents abuse and protects server resources.

Q75. What is Content Negotiation in ASP.NET Core Web API?

Answer:
It decides whether the client wants:

a. JSON
b. XML
c. or some other format

Example:
If the client sends Accept: application/xml, Web API can automatically return XML.

To enable XML:
```
builder.Services.AddControllers().AddXmlSerializerFormatters();
```

Note: Smart APIs support both JSON and XML based on client preferences.

Q76. How do you send an email inside ASP.NET Core Web API?

Answer:
Using SmtpClient:

```
var client = new SmtpClient("smtp.gmail.com")
{
    Port = 587,
    Credentials = new NetworkCredential("yourEmail@gmail.com", "password"),
    EnableSsl = true,
};

await client.SendMailAsync("from@gmail.com", "to@gmail.com", "subject", "body");
```

Note: For production, always use secured, environment-based configurations.

Q77. How do you secure sensitive data like connection strings in ASP.NET Core?

Answer:
Use Secret Manager during development:

```
dotnet user-secrets init
dotnet user-secrets set "ConnectionStrings:Default" "Server=mydb..."
```

Or Environment Variables in production.

Note: Never hardcode sensitive data inside appsettings.json!

Q78. How can you trigger code before the server shuts down?

Answer:
Implement IHostedService or use ApplicationLifetime events.

Example:

```
app.Lifetime.ApplicationStopping.Register(() =>
{
    Console.WriteLine("Application is shutting down...");
});
```

Note: Useful for graceful shutdown tasks like closing connections.

Q79. How do you bind nested complex objects from JSON?

Answer:
Model classes:

```
public class Order
{
    public int Id { get; set; }
    public Customer Customer { get; set; }
}

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

POST JSON:

```
{
```

```
  "id": 1,
  "customer": { "name": "Karthik", "email": "kar@example.com" }
}
```

Note: Web API automatically binds deeply nested structures!

Q80. What are some common status codes returned by APIs and their meanings?

Answer:

StatusCode	Meaning
200 OK	Request successful
201 Created	Resource created successfully
400 Bad Request	Client error
401 Unauthorized	Authentication needed
403 Forbidden	Access denied
404 Not Found	Resource not found
500 Internal Server Error	Server crash or unexpected error

Note: Using proper status codes makes APIs professional and client-friendly.

Q81. How do you use Middleware to modify request and response globally?

Answer:
Example of a simple custom Middleware:

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

    public RequestResponseLoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        Console.WriteLine($"Request Path: {context.Request.Path}");

        await _next(context);

        Console.WriteLine($"Response Status Code: {context.Response.StatusCode}");
    }
}
```

Register in Program.cs:

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

Note: Useful for logging, security, global header modifications.

Q82. What is Minimal API in .NET 6+?

Answer:
Minimal API allows you to define API endpoints without Controllers.

Example:

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

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

app.Run();
```

Note: Perfect for small microservices or lightweight APIs!

Q83. How to upload multiple files in ASP.NET Core Web API?

Answer:
Controller:

```
[HttpPost("upload-multiple")]
public async Task<IActionResult> UploadFiles(List<IFormFile> files)
{
    foreach (var file in files)
    {
        var path = Path.Combine("uploads", file.FileName);
        using (var stream = new FileStream(path, FileMode.Create))
        {
            await file.CopyToAsync(stream);
        }
    }
    return Ok("Files uploaded successfully");
}
```

Note: Important to set enctype="multipart/form-data" on client forms.

Q84. How do you create strongly typed appsettings.json configuration?

Answer:
Define POCO class:

```
public class MySettings
{
    public string SiteTitle { get; set; }
    public int RefreshInterval { get; set; }
}
```

In Program.cs:

```
builder.Services.Configure<MySettings>(builder.Configuration.GetSection("MySettings"));
```

Inject and use:

```
private readonly MySettings _settings;
public MyController(IOptions<MySettings> options)
{
    _settings = options.Value;
}
```

Note: Clean and safe way to handle configuration values.

Q85. How do you return a custom error model instead of default error in API?

Answer:
Custom error response:

```
[HttpGet("get")]
public IActionResult Get()
{
    return BadRequest(new { Message = "Invalid request", ErrorCode = 4001 });
}
```

Note: Always preferred for API-first designs where clients expect structured errors.

Q86. What is the difference between 401 and 403 HTTP status codes?

Answer:

Status Code	Meaning
401 Unauthorized	User not authenticated
403 Forbidden	User authenticated but not authorized

Note: "Unauthorized" = Login needed, "Forbidden" = Login done but no permission!

Q87. How do you implement Retry Policies for API calls in ASP.NET Core?

Answer:
Use Polly NuGet package:

```
builder.Services.AddHttpClient("RetryApi")
    .AddTransientHttpErrorPolicy(policy =>
        policy.WaitAndRetryAsync(3, _ => TimeSpan.FromSeconds(2)));
```

Note: Automatically retries failed HTTP requests due to timeout, server error, etc.

Q88. What are Fluent Validation libraries in ASP.NET Core?

Answer:

a. Third-party validation framework
b. More expressive than DataAnnotations

Install FluentValidation:

```
builder.Services.AddFluentValidationAutoValidation();
```

Create validator:

```
public class ProductValidator : AbstractValidator<Product>
{
    public ProductValidator()
    {
        RuleFor(x => x.Name).NotEmpty();
        RuleFor(x => x.Price).GreaterThan(0);
    }
}
```

Note: Clean, powerful and testable model validation.

Q89. How to add HSTS (HTTP Strict Transport Security) in ASP.NET Core?

Answer:
Add in Program.cs:

```
app.UseHsts();
```

Or configure:

```
app.UseHsts(hsts =>
{
    hsts.MaxAge = TimeSpan.FromDays(365);
    hsts.IncludeSubDomains();
    hsts.Preload();
});
```

Note: Forces browsers to use HTTPS instead of HTTP, even if typed manually.

Q90. How do you implement Health Checks in ASP.NET Core Web API?

Answer:
Configure:

```
builder.Services.AddHealthChecks();
```

Map endpoint:

```
app.MapHealthChecks("/health");
```

Browser:
https://localhost:5001/health

Note: Easy way to monitor API uptime for DevOps and Kubernetes readiness.

Q91. How do you implement Role-Based Authorization in Web API?

Answer:
Add roles to policies:

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

Use in Controller:

```
[Authorize(Roles = "Admin")]
[HttpGet("admin")]
public IActionResult AdminArea()
{
    return Ok("Welcome Admin");
}
```

Note: Ensures that only users with specific roles can access certain API endpoints.

Q92. What is the difference between Claims and Roles?

Answer:

```
Aspect      Role                                Claim
=================================================
Purpose  Authorization    Any info about user (email, age, department)
Type     String value     Key-Value pair (flexible)
Example  Admin", "User"   "Department" = "Finance", "Age" = 30
```

Note: Roles are specific to permission, while Claims are broader user metadata.

Q93. How do you consume a third-party Web API inside ASP.NET Core?

Answer:
Using HttpClientFactory:

```
builder.Services.AddHttpClient();
```

In Controller:

```
private readonly IHttpClientFactory _clientFactory;

public MyController(IHttpClientFactory clientFactory)
{
    _clientFactory = clientFactory;
}

[HttpGet]
public async Task<IActionResult> GetData()
{
    var client = _clientFactory.CreateClient();
    var response = await client.GetAsync("https://api.example.com/data");
    var data = await response.Content.ReadAsStringAsync();
    return Ok(data);
}
```

Note: Clean way to consume REST APIs inside your project.

Q94. How do you configure Swagger UI only for Development Environment?

Answer:
In Program.cs:

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Note: Prevents exposing sensitive API docs on production servers.

Q95. What are Filters in Web API and types of Filters?

Answer:

```
Type            Purpose
================================================
Authorization   Authorize before controller action
Resource        Runs before and after controller
Action          Runs before and after an action
```

```
Exception       Handles unhandled exceptions
Result          Runs before/after writing response

Note:  Filters add cross-cutting concerns like logging, caching, validation globally.
```

Q96. How do you implement Custom Action Filters?

Answer:
Create filter:

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

Use on Controller or Action:

```
[LogActionFilter]
[HttpGet]
public IActionResult Get()
{
    return Ok();
}
```

Note: Custom filters allow plugging custom behavior into the action pipeline.

Q97. How do you generate JWT tokens manually without Identity framework?

Answer:
Example:

```
var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("MySuperSecretKey"));
var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(
    issuer: "myapi.com",
    audience: "myapi.com",
    claims: new[] { new Claim(ClaimTypes.Name, "Karthik") },
    expires: DateTime.Now.AddMinutes(30),
    signingCredentials: credentials
);

string jwt = new JwtSecurityTokenHandler().WriteToken(token);
```

Note: Manual control over token contents, expiry, and claims!

Q98. What is Caching in Web API and why is it important?

Answer:
Caching stores previously generated responses to serve faster next time.

Types:

In-Memory Caching: Cached inside API memory.
Distributed Caching: Cached externally (e.g., Redis).

Add memory cache:

```
builder.Services.AddMemoryCache();

Use in Controller:
private readonly IMemoryCache _cache;

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

[HttpGet]
public IActionResult GetData()
{
    if (!_cache.TryGetValue("data", out string data))
    {
        data = "Database Data";
        _cache.Set("data", data, TimeSpan.FromMinutes(5));
    }
    return Ok(data);
}

Note: Boosts performance and reduces database/API load.



Q99. How to validate incoming JSON against a JSON Schema?

Answer:
Use NJsonSchema or FluentValidation libraries.

Example using FluentValidation:
builder.Services.AddFluentValidationAutoValidation();

Create a Validator class:
public class ProductValidator : AbstractValidator<Product>
{
    public ProductValidator()
    {
        RuleFor(x => x.Name).NotEmpty();
        RuleFor(x => x.Price).GreaterThan(0);
    }
}

Note: Ensures request payloads are correct before processing!



Q100. How to handle large file uploads in ASP.NET Core Web API?

Answer:
Increase request size limit:

builder.WebHost.ConfigureKestrel(options =>
{
    options.Limits.MaxRequestBodySize = 50 * 1024 * 1024; // 50 MB
});

Or use [RequestSizeLimit] attribute:

[RequestSizeLimit(52428800)]
[HttpPost("upload-large")]
public async Task<IActionResult> UploadLargeFile(IFormFile file)
{
    // process file
    return Ok();
}
```

Note: Prevents server crashes when large files are uploaded!