

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
```

```

66 {
67     [HttpPost]
68     public IActionResult CreateOrder(Order order)
69     {
70         // ModelState automatically validated
71         return Ok();
72     }
73 }
74

```

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:

```

85
86 [Route("api/products")]
87 public class ProductsController : ControllerBase
88 {
89     [HttpGet("{id}")]
90     public IActionResult GetProductById(int id)
91     {
92         // Fetch product
93         return Ok();
94     }
95 }
96

```

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:

```

107
108 [HttpGet("{id}")]
109 public IActionResult GetProduct(int id)
110 {
111     var product = repository.Get(id);
112     if (product == null)
113         return NotFound();
114
115     return Ok(product);
116 }
117

```

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:

```

128 builder.Services.AddScoped<IProductService, ProductService>();
129

```

Use constructor injection:

```

132 private readonly IProductService _service;

```

```
133 public ProductsController(IProductService service)
134 {
135     _service = service;
136 }
137
```

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:

```
151 [HttpGet]
152 public IActionResult Get()
153 {
154     var user = new { Name = "Karthik", Age = 30 };
155     return Ok(user);
156 }
```

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:

```
170 [HttpPost]
171 public IActionResult CreateProduct([FromBody] Product product)
172 {
173     // product object populated automatically from JSON body
174 }
```

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:

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

```

200
201 vs
202
203 public ActionResult<Product> GetProduct(int id)
204
205 Note: Using ActionResult<Product> allows you to return either a Product (200 OK) or
    other error codes cleanly.
206
207
208
209 Q10. How do you validate incoming request data in Web API?
210
211 Answer:
212
213 a. Use Data Annotations in Models (like [Required], [MaxLength], [Range]).
214 b. Use [ApiController] attribute to auto-check ModelState.
215 c. Handle invalid data gracefully.
216
217 Model:
218
219 public class Product
220 {
221     [Required]
222     public string Name { get; set; }
223
224     [Range(1, 10000)]
225     public decimal Price { get; set; }
226 }
227
228 Controller:
229
230 [HttpPost]
231 public IActionResult Create(Product product)
232 {
233     if (!ModelState.IsValid)
234         return BadRequest(ModelState);
235
236     return Ok();
237 }
238
239 Note: In most cases, with [ApiController], you don't need to write ModelState.IsValid
    manually – it auto returns 400!
240
241
242 Q11. How do you secure a Web API using JWT (JSON Web Token)?
243
244 Answer:
245
246 1. Install NuGet package:
247 Microsoft.AspNetCore.Authentication.JwtBearer
248
249 2. Configure authentication in Program.cs:
250 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
251     .AddJwtBearer(options =>
252     {
253         options.TokenValidationParameters = new TokenValidationParameters
254         {
255             ValidateIssuer = true,
256             ValidateAudience = true,
257             ValidateLifetime = true,
258             IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKey"))
259         };
260     });
261
262 3. Protect API endpoints:
263
264 [Authorize]
265 [HttpGet]

```

```
266 public IActionResult GetSecretData()
267 {
268     return Ok("This is protected");
269 }
```

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

272
273
274

275

276
277 Q12. How can you enable CORS in Web API?

278

279 Answer:

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

281

282 Register CORS in Program.cs:

283

```
284 builder.Services.AddCors(options =>
285 {
286     options.AddPolicy("AllowAll", builder =>
287     {
288         builder.AllowAnyOrigin()
289             .AllowAnyMethod()
290             .AllowAnyHeader();
291     });
292 });
```

293

294 Add it to the HTTP pipeline:

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

296

297 Now your API can accept requests from any domain.

298 Important: For production, restrict origins instead of AllowAnyOrigin().

299

300

301

302

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

304

305 Answer:

306

307 Install package:

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

309

310 Configure it:

```
311 builder.Services.AddApiVersioning(options =>
312 {
313     options.AssumeDefaultVersionWhenUnspecified = true;
314     options.DefaultApiVersion = new ApiVersion(1, 0);
315     options.ReportApiVersions = true;
316 });
```

317

318 Use attributes:

319

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

323

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

325

326

327

328

329

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

331

332 Answer:

333 Action Filters run before and after an action method executes.

```

334
335 Custom filter:
336 public class LogActionFilter : IActionFilter
337 {
338     public void OnActionExecuting(ActionExecutingContext context)
339     {
340         Console.WriteLine("Action Starting: " + context.ActionDescriptor.DisplayName);
341     }
342
343     public void OnActionExecuted(ActionExecutedContext context)
344     {
345         Console.WriteLine("Action Ended: " + context.ActionDescriptor.DisplayName);
346     }
347 }
348

```

```

349 Apply it:
350 [ServiceFilter(typeof(LogActionFilter))]
351 public class ProductsController : ControllerBase
352
353 Or globally in Program.cs.
354
355
356
357
358

```

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

360 Answer:

361 You can use ProblemDetails:

```

362 return Problem(detail: "Invalid Request", statusCode: 400);
363
364

```

365 Or create a custom response:

```

366 return BadRequest(new { ErrorCode = 1001, ErrorMessage = "Product not found" });
367
368

```

369 Note: Customizing error responses improves client debugging and UX.

370

371

372

373

374 Q16. What is the difference between FromBody and FromQuery?

375

376 Answer:

377

378

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

383

384

385 Example:

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

402

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

Answer:

Use built-in ILogger<T> service.

Inject logger:

```
403 private readonly ILogger<ProductsController> _logger;
404 public ProductsController(ILogger<ProductsController> logger)
405 {
406     _logger = logger;
407 }
408
409 Use it:
410 _logger.LogInformation("Fetching all products");
411
412 Note: Logging supports multiple sinks like Console, File, Azure Monitor, etc.
```

413
414

415
416

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

418
419

420 Answer:
421 Accept files as IFormFile.

422
423

424 Controller:

```
424 [HttpPost("upload")]
425 public async Task<IActionResult> UploadFile(IFormFile file)
426 {
427     if (file == null || file.Length == 0)
428         return BadRequest("No file selected");
429
430     var path = Path.Combine("uploads", file.FileName);
431
432     using (var stream = new FileStream(path, FileMode.Create))
433     {
434         await file.CopyToAsync(stream);
435     }
436
437     return Ok("Uploaded Successfully");
438 }
```

439
440

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

442
443

444
445

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

447
448

449 Answer:

450
451

452 Install package:
453 Swashbuckle.AspNetCore

454
455

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

459
460

461 Add middleware:
462 app.UseSwagger();
463 app.UseSwaggerUI();

464
465

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

468
469

470
471

472
473

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

475
476

477 Answer:

478
479

480 Status Code Meaning

471 200 OK Success, with content in response body
472 204 No Content Success, but no content returned

473

474 Example:

475

476 `return Ok(product);` → Returns 200 OK with product data.

477 `return NoContent();` → Returns 204 No Content (useful after Delete operations).

478

479

480

481

482

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

484

485 Answer:

486 Use a custom middleware to catch unhandled exceptions.

487

488 Example:

489

490 `public class ExceptionMiddleware`

491 `{`

492 `private readonly RequestDelegate _next;`

493 `private readonly ILogger<ExceptionMiddleware> _logger;`

494

495 `public ExceptionMiddleware(RequestDelegate next, ILogger<ExceptionMiddleware> logger)`

496 `{`

497 `_next = next;`

498 `_logger = logger;`

499 `}`

500

501 `public async Task InvokeAsync(HttpContext context)`

502 `{`

503 `try`

504 `{`

505 `await _next(context);`

506 `}`

507 `catch (Exception ex)`

508 `{`

509 `_logger.LogError(ex, "Unhandled Exception");`

510 `context.Response.StatusCode = 500;`

511 `await context.Response.WriteAsync("Internal Server Error");`

512 `}`

513 `}`

514 `}`

515

516 Register it:

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

518

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

520

521

522

523

524

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

526

527 Answer:

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

529

530 Examples:

531

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

533 `public IActionResult GetProduct([FromRoute] int id)`

534

535 `[HttpGet("search")]`

536 `public IActionResult SearchProduct([FromQuery] string name)`

537

538 `[HttpPost]`

539 `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.

609
610
611 Q26. How do you implement Rate Limiting in ASP.NET Core Web API?
612

613 Answer:

614 Use third-party libraries like `AspNetCoreRateLimit`.

615
616 Example basic rule:

617 Max 100 requests per 1 minute per client.

618
619 You configure it in `appsettings.json` and register services like:
620 `builder.Services.AddMemoryCache();`
621 `builder.Services.Configure<IpRateLimitOptions>(Configuration.GetSection("IpRateLimiting"))`
622 `);`
623 `builder.Services.AddInMemoryRateLimiting();`
624 `builder.Services.AddSingleton<IRateLimitConfiguration, RateLimitConfiguration>();`

625 Note: Protects your API from abuse.

626
627
628
629
630 Q27. What is Content Negotiation in ASP.NET Core Web API?
631

632 Answer:

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

634
635 Example:

636
637 Request Header: `Accept: application/xml`

638
639 ASP.NET Core Web API will respond with XML if XML formatter is added:

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

641
642 Note: Otherwise, JSON is the default.

643
644
645
646 Q28. How do you implement soft delete in Web API?
647

648 Answer:

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

650
651 Entity:

652
653 `public class Product`
654 `{`
655 `public int Id { get; set; }`
656 `public string Name { get; set; }`
657 `public bool IsDeleted { get; set; }`
658 `}`

659
660 Instead of deleting:
661 `product.IsDeleted = true;`
662 `dbContext.SaveChanges();`

663
664 Query only non-deleted records:
665 `dbContext.Products.Where(p => !p.IsDeleted);`
666

667
668
669
670 Q29. How can you handle large file uploads efficiently in Web API?
671

672 Answer:

673
674 Increase limits:
675 `builder.Services.Configure<FormOptions>(options =>`

```
676 {
677     options.MultipartBodyLengthLimit = 209715200; // 200 MB
678 });
679
680 Use streaming:
681 Instead of loading into memory, read stream directly.
682
683 using var stream = file.OpenReadStream();
684 using var destination = File.Create(path);
685 await stream.CopyToAsync(destination);
686
687 Note: This avoids memory overflow for large files.
688
689
690
691
692 Q30. How do you inject configuration settings into Controllers?
693
694 Answer:
695 Inject IConfiguration.
696
697 Example:
698
699 private readonly IConfiguration _config;
700 public ProductsController(IConfiguration config)
701 {
702     _config = config;
703 }
704
705 Usage:
706 var connectionString = _config.GetConnectionString("DefaultConnection");
707
708 Note: You can access appsettings.json or environment variables easily!
709
710
711
712
713 Q31. How do you send custom headers from a Web API response?
714
715 Answer:
716 You can add custom headers to the HTTP response like this:
717
718 [HttpGet]
719 public IActionResult GetProduct()
720 {
721     Response.Headers.Add("X-Custom-Header", "KarthikAPI");
722     return Ok(new { Message = "Success" });
723 }
724
725 Note: Useful for sending metadata, version info, correlation IDs, etc.
726
727
728
729
730
731 Q32. How can you consume a Web API using HttpClient in .NET?
732
733 Answer:
734 Example client:
735
736 var client = new HttpClient();
737 var response = await client.GetAsync("https://localhost:5001/api/products");
738 if (response.IsSuccessStatusCode)
739 {
740     var content = await response.Content.ReadAsStringAsync();
741     Console.WriteLine(content);
742 }
743
744 Note: HttpClient is the standard way to call external APIs from your code.
```

745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812

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. HttpResponseMessage (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]
```

```
813 public async Task<IActionResult> GetProducts()
814 {
815     var products = await _service.GetAllAsync();
816     return Ok(products);
817 }
```

818
819 Note: Async/await should be preferred in production Web APIs!

820
821
822
823
824
825
826 Q36. How do you unit test a Web API Controller?

827
828 Answer:

829 Use Mocking frameworks like Moq + Xunit.

830
831 Example basic unit test:

```
832  
833 var mockService = new Mock<IProductService>();
834 mockService.Setup(x => x.GetAll()).Returns(new List<Product> { new Product { Id = 1,
835     Name = "Sample" } });
```

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

```
838  
839 Assert.IsType<OkObjectResult>(result);
```

840
841 Note: Always mock dependencies and test controller logic independently.

842
843
844
845
846
847 Q37. How do you use Token-based Authentication in Web API?

848
849 Answer:

850 Generate JWT Token after login:

```
851  
852 var tokenHandler = new JwtSecurityTokenHandler();
853 var key = Encoding.ASCII.GetBytes("YourSecretKey");
854 var tokenDescriptor = new SecurityTokenDescriptor
855 {
856     Subject = new ClaimsIdentity(new Claim[]
857     {
858         new Claim(ClaimTypes.Name, user.Id.ToString())
859     }),
860     Expires = DateTime.UtcNow.AddHours(1),
861     SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
862     SecurityAlgorithms.HmacSha256Signature)
863 };
864 var token = tokenHandler.CreateToken(tokenDescriptor);
865 return tokenHandler.WriteToken(token);
```

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

867
868
869
870
871
872
873 Q38. What is Kestrel in ASP.NET Core?

874
875 Answer:

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

877
878 Features:

```

879 a. Lightweight, high-performance
880 b. Used as an edge server or behind reverse proxy (e.g., IIS, Nginx)
881 c. Can handle both HTTP/1.x and HTTP/2
882
883 Note: ASP.NET Core apps always run inside Kestrel, whether self-hosted or behind a proxy.
884
885
886
887
888
889 Q39. What is the difference between AddSingleton, AddScoped, and AddTransient services?
890
891 Answer:
892
893
894 Lifetime          Scope          Usage
895 =====
896 Singleton         One instance for entire app      Configurations, stateless services
897 Scoped            One instance per HTTP request    Database contexts
898 Transient         New instance every time requested Lightweight services
899
900
901 Example:
902
903 builder.Services.AddSingleton<IMyService, MyService>();
904 builder.Services.AddScoped<IMyDbContext, MyDbContext>();
905 builder.Services.AddTransient<ILogicHelper, LogicHelper>();
906
907 Note: Choosing the right lifetime impacts memory and performance.
908
909
910
911
912 Q40. How do you use Swagger to test secured APIs (with JWT Token)?
913
914 Answer:
915
916 Add security definition in Program.cs:
917
918 builder.Services.AddSwaggerGen(c =>
919 {
920     c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
921     {
922         In = ParameterLocation.Header,
923         Description = "Please enter JWT with Bearer into field",
924         Name = "Authorization",
925         Type = SecuritySchemeType.ApiKey
926     });
927
928     c.AddSecurityRequirement(new OpenApiSecurityRequirement
929     {
930         {
931             new OpenApiSecurityScheme
932             {
933                 Reference = new OpenApiReference { Type = ReferenceType.SecurityScheme,
934                 Id = "Bearer" }
935             },
936             new string[] { }
937         }
938     });
939 });
940
941 Note: Now, Swagger UI will show an Authorize button to paste the token. It Helps easily
942 test secured endpoints during development!
943
944
945

```

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);
```

```
1015 var app = builder.Build();
1016
1017 app.MapGet("/hello", () => "Hello World!");
1018 app.Run();
1019
1020 Note: Great for small, lightweight APIs and microservices.
1021
1022
1023
1024
1025 Q45. How do you bind complex types from the URL in Web API?
1026
1027 Answer:
1028 Bind from query parameters:
1029
1030 [HttpGet]
1031 public IActionResult Search([FromQuery] ProductFilter filter)
1032
1033 Where ProductFilter is:
1034
1035 public class ProductFilter
1036 {
1037     public string Name { get; set; }
1038     public int? CategoryId { get; set; }
1039 }
1040
1041 Note: Supports advanced filtering, sorting, and searching via query strings.
1042
1043
1044
1045
1046 Q46. How can you handle circular references in Web API responses (like in EF Core)?
1047
1048 Answer:
1049 Configure JSON options to ignore cycles:
1050
1051 builder.Services.AddControllers().AddJsonOptions(x =>
1052 {
1053     x.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
1054 });
1055
1056 Note: Prevents serialization errors like Self referencing loop detected when returning
entity models.
1057
1058
1059
1060
1061 Q47. What are Filters in ASP.NET Core Web API?
1062
1063 Answer:
1064 Filters allow you to run code before or after key pipeline stages.
1065
1066 Types:
1067
1068 a. Authorization filters
1069 b. Resource filters
1070 c. Action filters
1071 d. Exception filters
1072 e. Result filters
1073
1074 Example Action Filter:
1075
1076 public class LogFilter : ActionFilterAttribute
1077 {
1078     public override void OnActionExecuting(ActionExecutingContext context)
1079     {
1080         Console.WriteLine("Action is starting...");
1081     }
1082 }
```



```
1083
1084 Apply:
1085
1086 [ServiceFilter(typeof(LogFilter))]
```

1087

1088

1089

1090

1091 Q48. How do you configure multiple environments (Development, Staging, Production) in ASP.NET Core?

1092

1093 Answer:

1094 Use launchSettings.json or set ASPNETCORE_ENVIRONMENT variable.

1095

1096 Access inside code:

1097

```
1098 if (env.IsDevelopment())
1099 {
1100     // Enable detailed error page
1101 }
1102 else
1103 {
1104     // Generic error page
1105 }
```

1106

1107 You can inject IWebHostEnvironment anywhere:

1108

```
1109 private readonly IWebHostEnvironment _env;
1110 public HomeController(IWebHostEnvironment env)
1111 {
1112     _env = env;
1113 }
```

1114

1115 Note: Enables environment-specific behavior easily.

1116

1117

1118

1119

1120 Q49. How do you send validation error responses in a standard format?

1121

1122 Answer:

1123 Custom middleware:

1124

```
1125 public class ValidationExceptionMiddleware
1126 {
1127     private readonly RequestDelegate _next;
1128
1129     public ValidationExceptionMiddleware(RequestDelegate next)
1130     {
1131         _next = next;
1132     }
1133
1134     public async Task InvokeAsync(HttpContext context)
1135     {
1136         try
1137         {
1138             await _next(context);
1139         }
1140         catch (ValidationException ex)
1141         {
1142             context.Response.StatusCode = 400;
1143             await context.Response.WriteAsJsonAsync(new { Error = ex.Message });
1144         }
1145     }
1146 }
```

1147

1148 Note: Makes client-side error handling much easier.

1149

1150

1151
1152
1153
1154
1155 Q50. What is a DTO and why should you use it in Web APIs?
1156

1157 Answer:

1158 DTO = Data Transfer Object
1159

1160 Use DTOs to:

1161
1162 Hide internal domain models
1163

1164 Avoid exposing sensitive data
1165

1166 Customize API response shape
1167

1168 Improve performance
1169

1170 Example DTO:

1171
1172 public class ProductDto
1173 {
1174 public string Name { get; set; }
1175 public decimal Price { get; set; }
1176 }
1177

1178 Controller:

1179
1180 [HttpGet("{id}")]
1181 public ActionResult<ProductDto> Get(int id)
1182 {
1183 var product = _context.Products.Find(id);
1184 if (product == null) return NotFound();
1185
1186 return new ProductDto
1187 {
1188 Name = product.Name,
1189 Price = product.Price
1190 };
1191 }
1192

1193 Note: DTOs are a best practice for secure, clean, and future-proof Web APIs.
1194
1195

1196 Q51. How do you return a file (like PDF, Excel) from an API?
1197

1198 Answer:

1199 Use File() helper method:
1200

1201 [HttpGet("download")]
1202 public IActionResult DownloadFile()
1203 {
1204 var bytes = System.IO.File.ReadAllBytes("files/sample.pdf");
1205 return File(bytes, "application/pdf", "downloaded_sample.pdf");
1206 }
1207

1208 Note: Automatically sets the correct content type and download prompt.
1209
1210
1211

1212 Q52. What is the purpose of ProducesResponseType attribute?
1213

1214 Answer:

1215 Documents possible HTTP responses.
1216

1217 Example:

1218
1219 [HttpGet("{id}")]

```

1220 [ProducesResponseType (StatusCodes.Status200OK)]
1221 [ProducesResponseType (StatusCodes.Status404NotFound)]
1222 public IActionResult GetProduct(int id)
1223 {
1224     ...
1225 }

```

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:

```

1246 public class Worker : BackgroundService
1247 {
1248     protected override async Task ExecuteAsync(CancellationToken stoppingToken)
1249     {
1250         while (!stoppingToken.IsCancellationRequested)
1251         {
1252             Console.WriteLine("Running...");
1253             await Task.Delay(1000, stoppingToken);
1254         }
1255     }
1256 }

```

Q54. What is the purpose of UseEndpoints middleware?

Answer:

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

Example:

```

1267 app.UseRouting();
1268 app.UseAuthentication();
1269 app.UseAuthorization();
1270 app.UseEndpoints(endpoints =>
1271 {
1272     endpoints.MapControllers();
1273 });

```

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:

```

1285 [HttpPost]
1286 public IActionResult AddProduct(Product product)

```

```
1287 {
1288     if (!ModelState.IsValid)
1289         return BadRequest(ModelState);
1290
1291     _context.Products.Add(product);
1292     _context.SaveChanges();
1293
1294     return Ok();
1295 }
```

1296 Note: With [ApiController], validation happens automatically.

1297
1298
1299
1300
1301

1302
1303 Q56. What is the purpose of IHttpContextAccessor?

1304
1305 Answer:
1306 It allows you to access HttpContext outside of Controllers, for example inside Services.
1307
1308 Inject and use:

```
1309
1310 private readonly IHttpContextAccessor _httpContextAccessor;
1311 public MyService(IHttpContextAccessor httpContextAccessor)
1312 {
1313     _httpContextAccessor = httpContextAccessor;
1314 }
1315
1316 var userId =
1317     _httpContextAccessor.HttpContext?.User?.FindFirst(ClaimTypes.NameIdentifier)?.Value;
```

1318 Note: Useful for accessing user claims, headers, session, etc., in deep layers.
1319

1320
1321
1322
1323

1324 Q57. How do you customize model validation error messages?

1325
1326 Answer:
1327 Using Data Annotations:

```
1328
1329 public class Product
1330 {
1331     [Required(ErrorMessage = "Product name is mandatory")]
1332     public string Name { get; set; }
1333
1334     [Range(1, 10000, ErrorMessage = "Price must be between 1 and 10000")]
1335     public decimal Price { get; set; }
1336 }
1337
```

1338 Note: Custom error messages make APIs much more user-friendly.
1339

1340
1341
1342

1343 Q58. How do you create an API that returns pagination results?

1344
1345 Answer:
1346 Standard pagination:

```
1347
1348 [HttpGet]
1349 public IActionResult GetProducts(int page = 1, int pageSize = 10)
1350 {
1351     var skip = (page - 1) * pageSize;
1352     var products = _context.Products.Skip(skip).Take(pageSize).ToList();
1353     return Ok(products);
1354 }
```

1355
1356 Note: Always return total count and current page info in real APIs!
1357
1358
1359
1360

1361
1362 Q59. How do you implement Caching in ASP.NET Core Web API?
1363

1364 Answer:

1365
1366 1. Response Caching:

1367
1368 Add middleware:

1369 `app.UseResponseCaching();`
1370

1371 Controller:

1372
1373 `[HttpGet]`

1374 `[ResponseCache(Duration = 60)]`

1375 `public IActionResult GetProducts()`

1376 `{`

1377 `return Ok(_service.GetProducts());`

1378 `}`
1379

1380 Note: Caches full HTTP response.
1381

1382 2. Memory Caching:

1383
1384 Register service:

1385 `builder.Services.AddMemoryCache();`
1386

1387 Use:

1388 `private readonly IMemoryCache _cache;`

1389 `public MyService(IMemoryCache cache)`

1390 `{`

1391 `_cache = cache;`

1392 `}`
1393

1394 `_cache.Set("key", value, TimeSpan.FromMinutes(10));`
1395
1396
1397
1398

1399 Q60. What is the default maximum request body size in ASP.NET Core?
1400

1401 Answer:

1402
1403 Default maximum size = 30 MB (for Kestrel).

1404 For IIS, it's controlled by `maxRequestLength` and `maxAllowedContentLength`.
1405
1406

1407 You can increase it in Kestrel:

1408
1409 `builder.WebHost.ConfigureKestrel(serverOptions =>`

1410 `{`

1411 `serverOptions.Limits.MaxRequestBodySize = 104857600; // 100 MB`

1412 `});`
1413

1414 Note: Important when uploading files, images, large JSON payloads.
1415
1416
1417
1418

1419 Q61. How do you implement Global Exception Handling in Web API?
1420

1421 Answer:

1422 Use a custom middleware:
1423

```

1424 public class ExceptionMiddleware
1425 {
1426     private readonly RequestDelegate _next;
1427
1428     public ExceptionMiddleware(RequestDelegate next)
1429     {
1430         _next = next;
1431     }
1432
1433     public async Task InvokeAsync(HttpContext context)
1434     {
1435         try
1436         {
1437             await _next(context);
1438         }
1439         catch (Exception ex)
1440         {
1441             context.Response.StatusCode = 500;
1442             await context.Response.WriteAsJsonAsync(new { Error = ex.Message });
1443         }
1444     }
1445 }

```

1446 Register it:

```
1447 app.UseMiddleware<ExceptionMiddleware>();
```

1448 Note: Centralizes error handling for your entire API.

1452 Q62. How can you upload files using ASP.NET Core Web API?

1453 Answer:

1454 Controller method:

```

1455 [HttpPost("upload")]
1456 public async Task<IActionResult> UploadFile(IFormFile file)
1457 {
1458     if (file == null || file.Length == 0)
1459         return BadRequest("No file selected");
1460
1461     var path = Path.Combine("uploads", file.FileName);
1462
1463     using (var stream = new FileStream(path, FileMode.Create))
1464     {
1465         await file.CopyToAsync(stream);
1466     }
1467
1468     return Ok("File uploaded successfully");
1469 }

```

1470 Note: Don't forget to increase MaxRequestBodySize if needed!

1476 Q63. How can you implement Soft Delete in a Web API with Entity Framework Core?

1477 Answer:

1478 Entity:

```

1479 public class Product
1480 {
1481     public int Id { get; set; }
1482     public string Name { get; set; }
1483     public bool IsDeleted { get; set; }
1484 }

```

```

1493
1494 Soft delete method:
1495
1496 public IActionResult Delete(int id)
1497 {
1498     var product = _context.Products.Find(id);
1499     if (product == null) return NotFound();
1500
1501     product.IsDeleted = true;
1502     _context.SaveChanges();
1503
1504     return NoContent();
1505 }
1506

```

1507 Note: This way, you don't physically remove records – good for audit trails!

1508
1509

1510
1511

1512
1513

1514 Q64. What is the difference between FromQuery, FromBody, and FromRoute attributes?

1515
1516

1516 Answer:

1517
1518

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

1524
1525

1525 Example:

1526
1527

```

1527 [HttpPost("product/{id}")]
1528 public IActionResult UpdateProduct([FromRoute] int id, [FromBody] Product product,
1529 [FromQuery] string source)

```

1529
1530

1530 Note: Understanding these helps handle multi-source data correctly!

1531
1532

1533
1534

1535
1536

1537 Q65. How can you return a 201 Created response with a location header?

1538
1539

1539 Answer:

1540 Use CreatedAtAction().

1541
1542

1542 Example:

1543
1544

```

1544 [HttpPost]
1545 public IActionResult Create(Product product)
1546 {
1547     _context.Products.Add(product);
1548     _context.SaveChanges();
1549
1550     return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
1551 }
1552

```

1553
1554

1554 Note: Correct way to indicate resource creation according to REST standards!

1555
1556

1557
1558

1559 Q66. What is the purpose of [BindProperty] attribute in ASP.NET Core?

1560

1561 Answer:
1562 Primarily used in Razor Pages but can apply to APIs too.
1563

1564 It binds request values automatically:

1565 [BindProperty]
1566 public Product Product { get; set; }

1567
1568 In Razor Pages:

1569
1570
1571 public async Task<IActionResult> OnPostAsync()
1572 {
1573 _context.Products.Add(Product);
1574 await _context.SaveChangesAsync();
1575 return RedirectToPage("../Index");
1576 }
1577

1578 Note: Simplifies data binding in page models.
1579
1580
1581
1582
1583
1584

1585 Q67. What are Action Results and Non-Action Results in Web API?
1586

1587 Answer:

1588
1589 Action Results: Returns HTTP-specific responses (e.g., Ok(), NotFound(), BadRequest()).
1590 Non-Action Results: Directly return object without wrapping (e.g., string, int).
1591

1592 Examples:

1593
1594 Action Result:
1595 public ActionResult<Product> Get(int id)
1596

1597 Non-Action Result:
1598 public Product GetProduct(int id)
1599

1600 Note: Action Results are more flexible and recommended for APIs!
1601
1602
1603
1604
1605

1606 Q68. How do you expose multiple versions of a Web API?
1607

1608 Answer:

1609 Use API versioning.
1610

1611 Install package:

1612
1613 Microsoft.AspNetCore.Mvc.Versioning
1614

1615 Setup:

1616
1617 builder.Services.AddApiVersioning(options =>
1618 {
1619 options.AssumeDefaultVersionWhenUnspecified = true;
1620 options.DefaultApiVersion = new ApiVersion(1, 0);
1621 options.ReportApiVersions = true;
1622 });
1623

1624 Controller:

1625
1626 [ApiVersion("1.0")]
1627 [Route("api/v{version:apiVersion}/{controller}")]
1628

1629 Note: Supports version via URL, query string, headers!

1630
1631
1632
1633
1634
1635 Q69. How can you enable CORS (Cross-Origin Resource Sharing) globally?
1636

1637 Answer:

1638
1639 1. Configure CORS:

1640
1641 builder.Services.AddCors(options =>
1642 {
1643 options.AddPolicy("AllowAll", builder =>
1644 {
1645 builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod();
1646 });
1647 });

1648
1649 2. Apply:

1650 app.UseCors("AllowAll");

1651
1652 Note: CORS is mandatory when APIs are called from different domains.
1653

1654

1655

1656 Q70. How do you pass complex objects in query strings safely?

1657

1658 Answer:

1659 Serialize the object to JSON, then URL encode it.

1660

1661 Client-side:

1662

1663 let obj = { name: "karthik", age: 25 };
1664 let queryString = encodeURIComponent(JSON.stringify(obj));
1665 fetch(/api/users?data=\${queryString});

1666

1667 Server-side:

1668

1669 [HttpGet]
1670 public IActionResult GetUser([FromQuery] string data)
1671 {
1672 var user = JsonSerializer.Deserialize<User>(data);
1673 return Ok(user);
1674 }

1675

1676 Note: Not best practice for very large objects – prefer POST with body instead!
1677

1678

1679

1680

1681 Q71. What is Dependency Injection in ASP.NET Core?

1682

1683 Answer:

1684 Dependency Injection (DI) means giving an object what it needs (dependencies) instead of
1685 letting it create them.

1686

1687 Example:

1688

1689 public interface IMessageService
1690 {
1691 string GetMessage();
1692 }

1693

1694 public class EmailService : IMessageService

1695 {

1696 public string GetMessage() => "Email sent!";

1697 }

1698

1699 Controller:

```

1698
1699 public class HomeController : ControllerBase
1700 {
1701     private readonly IMessageService _service;
1702
1703     public HomeController(IMessageService service)
1704     {
1705         _service = service;
1706     }
1707
1708     [HttpGet]
1709     public string Index() => _service.GetMessage();
1710 }
1711
1712 Register service:
1713 builder.Services.AddScoped<IMessageService, EmailService>();
1714
1715 Note: Promotes loose coupling and easy testing.
1716
1717
1718
1719
1720 Q72. What are the different lifetimes for services in ASP.NET Core?
1721
1722 Answer:
1723
1724
1725 Lifetime      Description
1726 =====
1727 Singleton     One instance for the entire app
1728 Scoped        One instance per request
1729 Transient     New instance every time injected
1730
1731
1732 Examples:
1733
1734 builder.Services.AddSingleton<MyService>();
1735 builder.Services.AddScoped<MyService>();
1736 builder.Services.AddTransient<MyService>();
1737
1738 Note: Choose wisely – wrong lifetime can lead to bugs or memory leaks!
1739
1740
1741
1742
1743
1744 Q73. How can you protect APIs with API Key Authentication?
1745
1746 Answer:
1747
1748 1. Create a Middleware:
1749
1750 public class ApiKeyMiddleware
1751 {
1752     private readonly RequestDelegate _next;
1753     private const string APIKEY = "X-API-KEY";
1754
1755     public ApiKeyMiddleware(RequestDelegate next)
1756     {
1757         _next = next;
1758     }
1759
1760     public async Task Invoke(HttpContext context)
1761     {
1762         if (!context.Request.Headers.TryGetValue(APIKEY, out var extractedApiKey))
1763         {
1764             context.Response.StatusCode = 401;
1765             await context.Response.WriteAsync("API Key missing");
1766             return;

```

```
1767     }
1768
1769     if (extractedApiKey != "MySuperSecretApiKey")
1770     {
1771         context.Response.StatusCode = 403;
1772         await context.Response.WriteAsync("Invalid API Key");
1773         return;
1774     }
1775
1776     await _next(context);
1777 }
1778 }
```

```
1780 2. Register:
1781 app.UseMiddleware<ApiKeyMiddleware>();
1782
```

1783 Note: Light-weight security mechanism for private APIs.

1784
1785
1786
1787
1788

1789 Q74. How do you limit request rates (Throttling) in Web APIs?

1790

1791 Answer:

1792 a. Use third-party middleware like:

1793 b. `AspNetCoreRateLimit`

1794 c. Custom Middleware

1795

1796 Basic example:

1797

```
1798 app.UseRateLimiter(new RateLimiterOptions
1799 {
```

```
1800     {
```

```
1801         PermitLimit = 5,
```

```
1802         Window = TimeSpan.FromMinutes(1)
1803     });
```

1803

1804 Note: Prevents abuse and protects server resources.

1805

1806

1807

1808

1809

1810 Q75. What is Content Negotiation in ASP.NET Core Web API?

1811

1812 Answer:

1813 It decides whether the client wants:

1814

1815 a. JSON

1816 b. XML

1817 c. or some other format

1818

1819 Example:

1820 If the client sends `Accept: application/xml`, Web API can automatically return XML.

1821

1822 To enable XML:

```
1823 builder.Services.AddControllers().AddXmlSerializerFormatters();
1824
```

1824

1825 Note: Smart APIs support both JSON and XML based on client preferences.

1826

1827

1828

1829

1830

1831 Q76. How do you send an email inside ASP.NET Core Web API?

1832

1833 Answer:

1834 Using `SmtpClient`:

1835

```
1836 var client = new SmtplibClient("smtp.gmail.com")
1837 {
1838     Port = 587,
1839     Credentials = new NetworkCredential("yourEmail@gmail.com", "password"),
1840     EnableSsl = true,
1841 };
1842
1843 await client.SendMailAsync("from@gmail.com", "to@gmail.com", "subject", "body");
1844
1845 Note: For production, always use secured, environment-based configurations.
1846
1847
1848
1849
1850
1851 Q77. How do you secure sensitive data like connection strings in ASP.NET Core?
1852
1853 Answer:
1854 Use Secret Manager during development:
1855
1856 dotnet user-secrets init
1857 dotnet user-secrets set "ConnectionStrings:Default" "Server=mydb..."
1858
1859 Or Environment Variables in production.
1860
1861 Note: Never hardcode sensitive data inside appsettings.json!
1862
1863
1864
1865
1866
1867 Q78. How can you trigger code before the server shuts down?
1868
1869 Answer:
1870 Implement IHostedService or use ApplicationLifetime events.
1871
1872 Example:
1873
1874 app.Lifetime.ApplicationStopping.Register(() =>
1875 {
1876     Console.WriteLine("Application is shutting down...");
1877 });
1878
1879 Note: Useful for graceful shutdown tasks like closing connections.
1880
1881
1882
1883
1884
1885 Q79. How do you bind nested complex objects from JSON?
1886
1887 Answer:
1888 Model classes:
1889
1890 public class Order
1891 {
1892     public int Id { get; set; }
1893     public Customer Customer { get; set; }
1894 }
1895
1896 public class Customer
1897 {
1898     public string Name { get; set; }
1899     public string Email { get; set; }
1900 }
1901
1902 POST JSON:
1903
1904 {
```

```
1905     "id": 1,
1906     "customer": { "name": "Karthik", "email": "kar@example.com" }
1907 }
```

1908
1909 Note: Web API automatically binds deeply nested structures!

1910
1911
1912
1913
1914
1915
1916 Q80. What are some common status codes returned by APIs and their meanings?

1917
1918 Answer:

Status Code	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

1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930 Note: Using proper status codes makes APIs professional and client-friendly.

1931
1932
1933
1934
1935 Q81. How do you use Middleware to modify request and response globally?

1936
1937 Answer:

1938 Example of a simple custom Middleware:

```
1939
1940 public class RequestResponseLoggingMiddleware
1941 {
1942     private readonly RequestDelegate _next;
1943
1944     public RequestResponseLoggingMiddleware(RequestDelegate next)
1945     {
1946         _next = next;
1947     }
1948
1949     public async Task Invoke(HttpContext context)
1950     {
1951         Console.WriteLine($"Request Path: {context.Request.Path}");
1952
1953         await _next(context);
1954
1955         Console.WriteLine($"Response Status Code: {context.Response.StatusCode}");
1956     }
1957 }
```

1958
1959 Register in Program.cs:

```
1960 app.UseMiddleware<RequestResponseLoggingMiddleware>();
```

1961
1962 Note: Useful for logging, security, global header modifications.

1963
1964
1965
1966
1967 Q82. What is Minimal API in .NET 6+?

1968
1969 Answer:

1970 Minimal API allows you to define API endpoints without Controllers.

1971
1972 Example:

```
1974 var builder = WebApplication.CreateBuilder(args);
1975 var app = builder.Build();
1976
1977 app.MapGet("/hello", () => "Hello World!");
1978
1979 app.Run();
1980
```

1981 Note: Perfect for small microservices or lightweight APIs!

1982
1983
1984
1985
1986
1987

1988 Q83. How to upload multiple files in ASP.NET Core Web API?

1989

1990 Answer:

1991

1991 Controller:

1992

```
1993 [HttpPost("upload-multiple")]
1994 public async Task<IActionResult> UploadFiles(List<IFormFile> files)
1995 {
1996     foreach (var file in files)
1997     {
1998         var path = Path.Combine("uploads", file.FileName);
1999         using (var stream = new FileStream(path, FileMode.Create))
2000         {
2001             await file.CopyToAsync(stream);
2002         }
2003     }
2004     return Ok("Files uploaded successfully");
2005 }
2006
```

2007

2007 Note: Important to set enctype="multipart/form-data" on client forms.

2008

2009

2010

2011

2012

2013

2014

2015

2015 Q84. How do you create strongly typed appsettings.json configuration?

2016

2017

2017 Answer:

2018

2018 Define POCO class:

2019

```
2020 public class MySettings
2021 {
2022     public string SiteTitle { get; set; }
2023     public int RefreshInterval { get; set; }
2024 }
2025
```

2026

2026 In Program.cs:

```
2027 builder.Services.Configure<MySettings>(builder.Configuration.GetSection("MySettings"));
2028
```

2029

2029 Inject and use:

```
2030 private readonly MySettings _settings;
2031 public MyController(IOptions<MySettings> options)
2032 {
2033     _settings = options.Value;
2034 }
2035
```

2036

2036 Note: Clean and safe way to handle configuration values.

2037

2038

2039

2040

2041

2042

2042 Q85. How do you return a custom error model instead of default error in API?

```

2043
2044 Answer:
2045 Custom error response:
2046
2047 [HttpGet("get")]
2048 public IActionResult Get()
2049 {
2050     return BadRequest(new { Message = "Invalid request", ErrorCode = 4001 });
2051 }
2052
2053 Note: Always preferred for API-first designs where clients expect structured errors.
2054
2055
2056
2057
2058
2059 Q86. What is the difference between 401 and 403 HTTP status codes?
2060
2061 Answer:
2062
2063
2064 Status Code          Meaning
2065 =====
2066 401 Unauthorized      User not authenticated
2067 403 Forbidden         User authenticated but not authorized
2068
2069
2070 Note: "Unauthorized" = Login needed, "Forbidden" = Login done but no permission!
2071
2072
2073
2074
2075
2076 Q87. How do you implement Retry Policies for API calls in ASP.NET Core?
2077
2078 Answer:
2079 Use Polly NuGet package:
2080
2081 builder.Services.AddHttpClient("RetryApi")
2082     .AddTransientHttpErrorPolicy(policy =>
2083         policy.WaitAndRetryAsync(3, _ => TimeSpan.FromSeconds(2)));
2084
2085 Note: Automatically retries failed HTTP requests due to timeout, server error, etc.
2086
2087
2088
2089
2090
2091 Q88. What are Fluent Validation libraries in ASP.NET Core?
2092
2093 Answer:
2094
2095 a. Third-party validation framework
2096 b. More expressive than DataAnnotations
2097
2098 Install FluentValidation:
2099 builder.Services.AddFluentValidationAutoValidation();
2100
2101 Create validator:
2102 public class ProductValidator : AbstractValidator<Product>
2103 {
2104     public ProductValidator()
2105     {
2106         RuleFor(x => x.Name).NotEmpty();
2107         RuleFor(x => x.Price).GreaterThan(0);
2108     }
2109 }
2110
2111 Note: Clean, powerful and testable model validation.

```

2112
2113
2114
2115
2116
2117 Q89. How to add HSTS (HTTP Strict Transport Security) in ASP.NET Core?
2118

2119 Answer:

2120 Add in Program.cs:

2121
2122 `app.UseHsts();`
2123

2124 Or configure:

2125
2126 `app.UseHsts(hsts =>`
2127 `{`
2128 `hsts.MaxAge = TimeSpan.FromDays(365);`
2129 `hsts.IncludeSubDomains();`
2130 `hsts.Preload();`
2131 `});`
2132

2133 Note: Forces browsers to use HTTPS instead of HTTP, even if typed manually.
2134
2135

2136
2137
2138
2139 Q90. How do you implement Health Checks in ASP.NET Core Web API?
2140

2141 Answer:

2142 Configure:

2143
2144 `builder.Services.AddHealthChecks();`
2145

2146 Map endpoint:

2147 `app.MapHealthChecks("/health");`
2148

2149 Browser:

2150 <https://localhost:5001/health>
2151

2152 Note: Easy way to monitor API uptime for DevOps and Kubernetes readiness.
2153
2154

2155 Q91. How do you implement Role-Based Authorization in Web API?
2156

2157 Answer:

2158 Add roles to policies:

2159
2160 `builder.Services.AddAuthorization(options =>`
2161 `{`
2162 `options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));`
2163 `});`
2164

2165 Use in Controller:

2166
2167 `[Authorize(Roles = "Admin")]`
2168 `[HttpGet("admin")]`
2169 `public IActionResult AdminArea()`
2170 `{`
2171 `return Ok("Welcome Admin");`
2172 `}`
2173

2174 Note: Ensures that only users with specific roles can access certain API endpoints.
2175
2176

2177 Q92. What is the difference between Claims and Roles?
2178

2179 Answer:
2180

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

2250 Exception Handles unhandled exceptions
2251 Result Runs before/after writing response
2252
2253 Note: Filters add cross-cutting concerns like logging, caching, validation globally.
2254
2255
2256
2257
2258 Q96. How do you implement Custom Action Filters?
2259
2260 Answer:
2261 Create filter:
2262
2263 public class LogActionFilter : ActionFilterAttribute
2264 {
2265 public override void OnActionExecuting(ActionExecutingContext context)
2266 {
2267 Console.WriteLine("Before action executes");
2268 }
2269 }
2270
2271 Use on Controller or Action:
2272
2273 [LogActionFilter]
2274 [HttpGet]
2275 public IActionResult Get()
2276 {
2277 return Ok();
2278 }
2279
2280 Note: Custom filters allow plugging custom behavior into the action pipeline.
2281
2282
2283
2284
2285
2286 Q97. How do you generate JWT tokens manually without Identity framework?
2287
2288 Answer:
2289 Example:
2290
2291 var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("MySuperSecretKey"));
2292 var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
2293
2294 var token = new JwtSecurityToken(
2295 issuer: "myapi.com",
2296 audience: "myapi.com",
2297 claims: new[] { new Claim(ClaimTypes.Name, "Karthik") },
2298 expires: DateTime.Now.AddMinutes(30),
2299 signingCredentials: credentials
2300);
2301
2302 string jwt = new JwtSecurityTokenHandler().WriteToken(token);
2303
2304 Note: Manual control over token contents, expiry, and claims!
2305
2306
2307
2308 Q98. What is Caching in Web API and why is it important?
2309
2310 Answer:
2311 Caching stores previously generated responses to serve faster next time.
2312
2313 Types:
2314
2315 In-Memory Caching: Cached inside API memory.
2316 Distributed Caching: Cached externally (e.g., Redis).
2317
2318 Add memory cache:

```

2319 builder.Services.AddMemoryCache();
2320
2321 Use in Controller:
2322 private readonly IMemoryCache _cache;
2323
2324 public MyController(IMemoryCache cache)
2325 {
2326     _cache = cache;
2327 }
2328
2329 [HttpGet]
2330 public IActionResult GetData()
2331 {
2332     if (!_cache.TryGetValue("data", out string data))
2333     {
2334         data = "Database Data";
2335         _cache.Set("data", data, TimeSpan.FromMinutes(5));
2336     }
2337     return Ok(data);
2338 }
2339

```

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:

```

2351 builder.Services.AddFluentValidationAutoValidation();
2352
2353 Create a Validator class:
2354 public class ProductValidator : AbstractValidator<Product>
2355 {
2356     public ProductValidator()
2357     {
2358         RuleFor(x => x.Name).NotEmpty();
2359         RuleFor(x => x.Price).GreaterThan(0);
2360     }
2361 }
2362

```

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:

```

2373 builder.WebHost.ConfigureKestrel(options =>
2374 {
2375     options.Limits.MaxRequestBodySize = 50 * 1024 * 1024; // 50 MB
2376 });
2377
2378 Or use [RequestSizeLimit] attribute:
2379
2380 [RequestSizeLimit(52428800)]
2381 [HttpPost("upload-large")]
2382 public async Task<IActionResult> UploadLargeFile(IFormFile file)
2383 {
2384     // process file
2385     return Ok();
2386 }
2387

```

2388 Note: Prevents server crashes when large files are uploaded!

2389

2390

2391

2392

2393