.NET Web API Interview Questions

100 practical, scenario-based questions with expert answers, code examples, and red flag warnings.

100	9	3
QUESTIONS	CATEGORIES	LEVELS

By **Mukesh Murugan**

codewithmukesh.com

Updated for .NET 10 & C# 14 | 2026 Edition

Table of Contents

How to Use This Guide

- **Difficulty badges** tell you the expected seniority level.
- **Why interviewers ask this** explains the intent behind the question.
- **Great answer** is how I would answer it, with code and context.
- **Red flag answer** is what gets you rejected.
- **Follow-up question** is what the interviewer asks next.

Junior	Fundamentals
Mid	Practical experience
Senior	Architecture & production

01

API Design & REST Best Practices

12 questions 2 Junior 5 Mid 5 Senior

1 Your API returns a list of products. A client asks for pagination. How do you implement it, and what metadata do you include in the response?

SENIOR

Why interviewers ask this: Pagination is table stakes for any production API. They want to see if you think beyond `skip/take` — do you consider cursor-based pagination, total counts (expensive!), and response envelope design?

GREAT ANSWER

I'd default to offset-based pagination for simple use cases — `?page=1&pageSize=20`. The response includes metadata so the client isn't guessing:

```json
{
  "data": [...],
  "pagination": {
    "currentPage": 1,
    "pageSize": 20,
    "totalCount": 487,
    "totalPages": 25,
    "hasNextPage": true,
    "hasPreviousPage": false
  }
}
```

But here's the thing — `totalCount` requires a `COUNT(*)` query which gets expensive on large tables. I've seen APIs slow down just because of that count. For high-volume endpoints, I'd switch to cursor-based pagination (keyset pagination) using the last item's ID: `?cursor=abc123&pageSize=20`. It's faster because it uses an index seek instead of offset skip.

I'd also set a max page size (say 50) server-side so a client can't request `pageSize=10000` and kill the database.

> **Red Flag:** I'd just use `.Skip()` and `.Take()` in EF Core." — Shows no awareness of performance implications or response design.

> ***Follow-up:** What happens to offset pagination when a record is deleted between page requests? How does cursor-based pagination handle this?*

2 Your POST endpoint creates a resource but also triggers an async background job (like sending a welcome email). What HTTP status code do you return?

MID

Why interviewers ask this: Tests whether you understand the nuance between `201 Created` and `202 Accepted`, and whether you're thoughtful about API contracts.

GREAT ANSWER

It depends on what the client cares about. If the resource itself is created synchronously and the email is just a side effect, I'd return `201 Created` with a `Location` header pointing to the new resource. The email is fire-and-forget — the client doesn't need to know about it.

But if the entire operation is async — say the resource creation itself is queued — then `202 Accepted` is correct. It means "I got your request, I'll process it later." I'd include a status endpoint URL so the client can poll: `Location: /api/orders/123/status`.

The worst thing you can do is return `200 OK` for a creation endpoint. It tells the client nothing.

⚠ **Red Flag:** 200 OK, because it worked." — Doesn't understand REST semantics.

💬 ***Follow-up:*** *When would you return `204 No Content` vs `200 OK` on a PUT endpoint?*

3 A client sends a PATCH request to update only the `email` field of a user. How do you implement partial updates in your API?

SENIOR

Why interviewers ask this: PATCH is tricky in .NET. They want to see if you've dealt with the difference between "field is null because the client wants to clear it" vs "field is null because the client didn't send it."

GREAT ANSWER

There are a few approaches, and I've used different ones depending on the situation:

Option 1 — JsonPatchDocument (Microsoft.AspNetCore.JsonPatch):

```csharp
app.MapPatch("/users/{id}", (int id, JsonPatchDocument<UserDto> patch) => { ... });
```

The client sends JSON Patch operations (`[{"op": "replace", "path": "/email", "value": "new@email.com"}]`). It works but the client-side experience is awkward.

Option 2 — Nullable DTOs with a "fields sent" tracker: I prefer a simpler approach where the DTO uses `Optional<T>` wrapper or I check `HttpContext.Request` to see which fields were actually in the JSON body. This way `null` means "clear this field" and "missing" means "don't touch it."

Option 3 — Just use PUT with full replacement. Honestly, for most internal APIs, this is fine. PATCH adds complexity, and if your entities are small, just send the whole thing.

My take: unless you have a strong reason for PATCH (large entities, bandwidth constraints, mobile clients), a well-designed PUT is simpler and less error-prone.

⚠ **Red Flag:** I'd just use a regular DTO and check for nulls." — Doesn't understand the null vs missing distinction.

***Follow-up:** How would you validate a PATCH request? Can you apply FluentValidation to a partial update?*

4 You're designing an API that serves both a web frontend and a mobile app. The mobile app needs less data than the web app. How do you handle this?

MID

Why interviewers ask this: Tests API design thinking — do you create separate endpoints, use query parameters, or something else?

GREAT ANSWER

A few options, each with trade-offs:

1. 1. **Sparse fieldsets** — `GET /api/products?fields=id,name,price`. The client specifies what it wants. This is flexible but adds complexity to your query builder. You have to be careful not to expose fields the client shouldn't see.
1. 2. **Separate DTOs per consumer** — `ProductSummaryDto` (mobile) vs `ProductDetailDto` (web). This is what I usually do for known consumers. It's explicit, easy to test, and doesn't require dynamic projection.
1. 3. **GraphQL** — If you genuinely have many consumers with wildly different data needs, GraphQL solves this natively. But for two consumers? It's overkill.

I'd go with option 2 for most cases. Two endpoints: `GET /api/products` returns the summary, `GET /api/products/{id}` returns the full detail. The mobile app uses the list endpoint and rarely needs the detail. Simple, predictable, cacheable.

⚠ **Red Flag:** I'd return everything and let the client ignore what it doesn't need." — Wastes bandwidth and exposes unnecessary data.

***Follow-up:** How do you version your API when the mobile app and web app evolve at different speeds?*

5 What's the difference between query string parameters and route parameters? When do you use each?

JUNIOR

Why interviewers ask this: Basic but reveals whether someone thinks about API design or just copies patterns.

GREAT ANSWER

Route parameters identify a specific resource: `/api/products/42`. They're required — the endpoint doesn't make sense without them.

Query string parameters are for optional modifiers: `/api/products?category=electronics&sort=price`. They filter, sort, page, or modify the response but the endpoint still works without them.

My rule of thumb: if removing the parameter makes the URL meaningless, it's a route parameter. If removing it just gives you the default behavior, it's a query parameter.

A common mistake I see: `/api/products?id=42`. That `id` should be in the route, not the query string. It identifies a specific resource.

Red Flag: They're the same thing, just different syntax.

Follow-up: *How does ASP.NET Core bind parameters from different sources — route, query, body, header? What's the binding order?*

6 Your API needs to support bulk operations — creating 500 products in a single request. How do you design this endpoint?

SENIOR

Why interviewers ask this: Bulk operations expose thinking about transactions, partial failures, request size limits, and API design.

GREAT ANSWER

I'd create a dedicated bulk endpoint: `POST /api/products/bulk`. The request body is an array of product DTOs. But the real question is: what happens when item 347 out of 500 fails validation?

Option A — All-or-nothing: Wrap everything in a transaction. If one fails, all fail. Return `400` with the specific validation errors. Simple but frustrating for the client — they have to fix one item and resubmit all 500.

Option B — Partial success: Process each item independently. Return `207 Multi-Status` with per-item results:

```json
{
  "succeeded": 498,
  "failed": 2,
  "errors": [
    { "index": 347, "error": "Name is required" },
    { "index": 412, "error": "Duplicate SKU" }
  ]
}
```

I prefer Option B for most cases. I'd also set a max batch size (500-1000), use `ExecuteUpdateAsync` or `BulkExtensions` instead of calling `SaveChangesAsync` 500 times, and consider making it async with `202 Accepted` if processing takes more than a few seconds.

⚠ **Red Flag:** I'd just loop through and call the single-create endpoint 500 times." — Shows no understanding of performance or transaction boundaries.

***Follow-up:** How would you handle idempotency for bulk operations? What if the client retries after a timeout?*

7 You need to add API versioning to an existing API that's already in production. What's your approach?

MID

Why interviewers ask this: Versioning is a real-world pain point. They want to see if you've actually done it.

GREAT ANSWER

I'd use the `Asp.Versioning` library (formerly `Microsoft.AspNetCore.Mvc.Versioning`). For the versioning strategy, there are three options:

1. 1. **URL segment** — `/api/v1/products` → `/api/v2/products`. Most explicit, easiest for clients to understand. This is what I default to.
2. 2. **Header** — `api-version: 2.0`. Keeps URLs clean but is invisible and harder to test in a browser.
3. 3. **Query string** — `?api-version=2.0`. Easy to add but clutters the URL.

For an existing API that already has clients, I'd: - Add versioning with URL segments - Mark all existing endpoints as `v1` (the default version) - New/changed endpoints go in `v2` - Deprecate `v1` endpoints with a sunset header, but don't remove them for at least 6 months - Document the migration path

The key mistake I see: people version the entire API when only 2 out of 20 endpoints changed. Version individual endpoints or controllers, not the whole API.

⚠ **Red Flag:** I'd just change the existing endpoints and tell clients to update." — Breaks existing integrations.

***Follow-up:** How do you handle versioning in your OpenAPI documentation? Can Scalar show multiple versions?*

8 Your API endpoint accepts a JSON body, but you want to enforce a maximum request size of 1MB. How do you do this, and what happens when a client exceeds it?

SENIOR

Why interviewers ask this: Security and resource management. Shows whether someone thinks about abuse scenarios.

GREAT ANSWER

In ASP.NET Core, Kestrel has a default max request body size of ~28.6 MB. That's way too high for most APIs.

I'd configure it at multiple levels:

Global (Program.cs):

```csharp
builder.WebHost.ConfigureKestrel(options =>
    options.Limits.MaxRequestBodySize = 1_048_576); // 1MB
```

Per-endpoint (for specific endpoints that need different limits):

```csharp
app.MapPost("/api/uploads", handler)
   .WithMetadata(new RequestSizeLimitAttribute(10_485_760)); // 10MB for uploads
```

When exceeded, Kestrel returns `413 Payload Too Large` automatically. But I'd also add request size validation in middleware to return a proper `ProblemDetails` response instead of the raw 413.

Don't forget: if you're behind a reverse proxy (Nginx, Azure App Gateway), you need to configure the limit there too. I've seen cases where Kestrel's limit was correct but Nginx had a 100MB default and happily forwarded massive payloads.

⚠ **Red Flag:** I didn't know there was a limit." — Dangerous in production.

***Follow-up:** How would you rate-limit file uploads specifically, separate from regular API rate limiting?*

9 A client sends `Accept: application/xml` but your API only supports JSON. What should happen?

MID

Why interviewers ask this: Content negotiation is often ignored. Tests whether someone has configured it intentionally.

GREAT ANSWER

The correct HTTP response is `406 Not Acceptable`. The client asked for a format the server can't provide.

In ASP.NET Core, the default behavior is... not great. By default, it'll just return JSON regardless of the `Accept` header. To enforce content negotiation properly:

```csharp
builder.Services.AddControllers(options =>
{
    options.ReturnHttpNotAcceptable = true; // Returns 406 instead of defaulting to JSON
});
```

For minimal APIs, content negotiation is simpler — `TypedResults.Ok(data)` returns JSON. If you genuinely need XML support, add `AddXmlSerializerFormatters()`.

My take: unless you have a specific client that needs XML (enterprise/SOAP integrations), don't add XML support. It increases your attack surface and doubles your serialization testing.

⚠ **Red Flag:** Just return JSON anyway, nobody uses XML." — Correct instinct, but shows no awareness of HTTP standards or how to configure it properly.

***Follow-up:** How does content negotiation work with custom media types like `application/vnd.company.product+json`?*

10 What's the difference between `[FromBody]`, `[FromQuery]`, `[FromRoute]`, and `[FromHeader]` in ASP.NET Core? When does it matter to specify them explicitly?

JUNIOR

Why interviewers ask this: Parameter binding is fundamental, and implicit binding can cause subtle bugs.

GREAT ANSWER

These attributes tell ASP.NET Core where to look for a parameter value:

- - `[FromRoute]` — URL path segments: `/api/products/{id}`
- - `[FromQuery]` — Query string: `?search=phone`
- - `[FromBody]` — Request body (JSON usually)
- - `[FromHeader]` — HTTP headers: `X-Correlation-Id`

For controllers, ASP.NET Core infers the source: simple types come from route/query, complex types from body. But this inference can go wrong — if you have a complex type as a query parameter (like a filter DTO), you need `[FromQuery]` explicitly.

For minimal APIs, the rules are slightly different. Complex types default to `[FromBody]`, and you must be explicit about `[FromQuery]` for complex types.

The real gotcha: you can only have **one** `[FromBody]` parameter per endpoint. If you need multiple body values, wrap them in a single DTO.

⚠ **Red Flag:** I never use those attributes, it just works automatically." — Works until it doesn't.

Follow-up: *How does `[AsParameters]` work in minimal APIs, and how does it differ from `[FromBody]`?*

11 You need to design an endpoint that triggers a long-running process (like report generation). The process takes 2-5 minutes. How do you design the API for this?

SENIOR

Why interviewers ask this: Tests async API design patterns — one of the most common real-world challenges.

GREAT ANSWER

Never make the client wait 5 minutes on an HTTP request. I'd use the async request-reply pattern:

Step 1 — Accept the request:

```
POST /api/reports
→ 202 Accepted
→ Location: /api/reports/abc123/status
```

Step 2 — Client polls for status:

```
GET /api/reports/abc123/status
→ 200 OK
→ { "status": "processing", "percentComplete": 45 }
```

Step 3 — When done:

```
GET /api/reports/abc123/status
→ 200 OK
→ { "status": "completed", "downloadUrl": "/api/reports/abc123/download" }
```

The actual processing happens in a background service (`BackgroundService` or a message queue like RabbitMQ). I'd store the job status in the database or Redis.

For a better experience, I'd also add SignalR notifications so the client gets a push when the report is ready instead of polling.

⚠ **Red Flag:** I'd increase the request timeout to 10 minutes." — Shows no understanding of async patterns.

Follow-up: *How do you handle the case where the background job fails? How does the client find out?*

12 Your API needs idempotency for POST requests. How do you implement it? MID

Why interviewers ask this: Critical for payment APIs, order processing — anywhere duplicate requests cause real damage.

GREAT ANSWER

I'd require an `Idempotency-Key` header on POST requests. The flow:

1. 1. Client sends `POST /api/orders` with header `Idempotency-Key: uuid-abc-123`
2. 2. Server checks if this key exists in the idempotency store (Redis or database table)
3. 3. If new: process the request, store the response with the key, return the response
4. 4. If duplicate: return the stored response without processing again

CSHARP

```csharp
app.MapPost("/api/orders", async (CreateOrderRequest request,
    [FromHeader(Name = "Idempotency-Key")] string idempotencyKey,
    IIdempotencyService idempotency) =>
{
    var cached = await idempotency.GetAsync(idempotencyKey);
    if (cached is not null) return cached;

    var result = await ProcessOrder(request);
    await idempotency.StoreAsync(idempotencyKey, result, TimeSpan.FromHours(24));
    return result;
});
```

Key decisions: the idempotency key should expire (24h is typical), and you need to handle the race condition where two identical requests arrive simultaneously (use a distributed lock or database unique constraint on the key).

⚠ **Red Flag:** GET and PUT are already idempotent, so it's not an issue." — Misses POST entirely.

***Follow-up:** How do you handle idempotency when the request succeeds but the response is lost due to a network error?*

02

ASP.NET Core Internals & Middleware

12 questions 2 Junior 6 Mid 4 Senior

13 You need request logging, authentication, rate limiting, CORS, and exception handling middleware. What order do you register them, and what breaks if you get it wrong?

MID

Why interviewers ask this: Middleware order is one of the most common sources of production bugs. Shows if someone understands the pipeline.

GREAT ANSWER

Order matters because middleware runs top-to-bottom on the request, bottom-to-top on the response. Here's the correct order:

```csharp
app.UseExceptionHandler();     // 1. Catch everything — must be first
app.UseHsts();                 // 2. Security headers
app.UseCors();                 // 3. CORS — before auth so preflight requests don't get 4
01
app.UseRateLimiter();          // 4. Rate limiting — before auth to protect from brute fo
rce
app.UseAuthentication();       // 5. Who are you?
app.UseAuthorization();        // 6. Are you allowed?
// Request logging here        // 7. Log after auth so you have the user identity
app.MapControllers();          // 8. Endpoints
```

What breaks with wrong order: - CORS after auth → preflight `OPTIONS` requests get `401 Unauthorized` - Exception handler not first → unhandled exceptions in early middleware crash with no ProblemDetails response - Rate limiter after auth → brute force attackers still hit your auth middleware - Auth before CORS → browser CORS errors that look like auth errors

I've seen a production bug where CORS was registered after authentication. The API worked fine in Postman but failed in the browser. Took hours to debug because the error message was misleading.

Red Flag: I just put them in whatever order and it works." — Works in dev, fails in production.

***Follow-up:** Where would you put response compression middleware? Before or after static files?*

14 You have a scoped service injected into a singleton. What happens, and how do you detect this before it reaches production?

SENIOR

Why interviewers ask this: The captive dependency problem. One of the most common DI mistakes in .NET.

GREAT ANSWER

This is the captive dependency problem. The scoped service gets captured by the singleton and lives forever — it never gets disposed when the scope ends. If that scoped service holds a database connection (like `DbContext`), you now have a connection that's shared across all requests. You'll get threading issues, stale data, and eventually connection pool exhaustion.

Detection: In .NET, add `ValidateScopes` and `ValidateOnBuild` in development:

```csharp
builder.Host.UseDefaultServiceProvider(options =>
{
    options.ValidateScopes = true;    // Catches captive dependencies
    options.ValidateOnBuild = true;   // Validates all registrations at startup
});
```

This throws an `InvalidOperationException` at startup instead of failing silently at runtime.

The fix: If a singleton needs data from a scoped service, inject `IServiceScopeFactory` and create a scope manually:

```csharp
public class MySingleton(IServiceScopeFactory scopeFactory)
{
    public async Task DoWork()
    {
        using var scope = scopeFactory.CreateScope();
        var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
        // Use db within this scope
    }
}
```

Red Flag: I'd just register everything as singleton to avoid the issue." — Makes it worse.

Follow-up: *What's the difference between* `Transient` *and* `Scoped` *in the context of a Web API request?*

15 What's the difference between middleware and endpoint filters in ASP.NET Core? When would you use each?

MID

Why interviewers ask this: Filters are often confused with middleware. Shows depth of framework understanding.

GREAT ANSWER

Middleware runs on every request and has access to the raw HTTP pipeline. It sees requests before routing happens. Use middleware for cross-cutting concerns like logging, CORS, exception handling, and authentication.

Endpoint filters run only on matched endpoints and have access to the endpoint's parameters and return type. They're like mini-middleware scoped to specific endpoints.

CSHARP

```csharp
// Middleware — runs on ALL requests
app.Use(async (context, next) => { /* raw HttpContext */ });

// Endpoint filter — runs only on this endpoint, has typed access
app.MapGet("/api/products", GetProducts)
    .AddEndpointFilter(async (context, next) =>
    {
        var id = context.GetArgument<int>(0); // Typed parameter access
        if (id <= 0) return Results.BadRequest("Invalid ID");
        return await next(context);
    });
```

My rule: if it needs to run on every request or before routing, use middleware. If it's specific to certain endpoints and needs parameter access, use filters.

Filters are perfect for validation, logging specific endpoints, caching headers, or transforming responses.

Red Flag: They're the same thing." — They operate at different levels of the pipeline.

Follow-up: *How do endpoint filters compose? What's the execution order when you have multiple filters?*

16 What is dependency injection, and why does ASP.NET Core use it by default?

JUNIOR

Why interviewers ask this: Foundational concept. Interviewers want to see if you understand the why, not just the what.

GREAT ANSWER

Instead of a class creating its own dependencies (using `new`), the dependencies are provided from outside — "injected" into the constructor.

```csharp
// Without DI — tightly coupled, hard to test
public class OrderService
{
    private readonly EmailService _email = new EmailService(); // Hardcoded dependency
}

// With DI — loosely coupled, testable
public class OrderService(IEmailService email) // Injected via constructor
{
}
```

ASP.NET Core has a built-in DI container because: 1. **Testability** — swap real implementations with mocks 2. **Lifetime management** — the container handles when objects are created and disposed (singleton, scoped, transient) 3. **Loose coupling** — classes depend on interfaces, not concrete types 4. **Configuration** — swap implementations without changing consuming code (different email provider? Just change the registration)

The three lifetimes matter: `Singleton` (one instance forever), `Scoped` (one per HTTP request), `Transient` (new instance every time). DbContext is scoped because each request needs its own database connection.

⚠ **Red Flag:** It's a design pattern where you inject things into classes." — Too vague, no mention of why.

***Follow-up:** When would you use `AddKeyedSingleton` or `AddKeyedScoped` — the keyed services feature?*

17 Your API has a global exception handler, but you also need to return different error formats for different types of exceptions (validation errors vs business logic errors vs unhandled crashes). How do you structure this?

SENIOR

Why interviewers ask this: Error handling strategy reveals production experience.

GREAT ANSWER

I'd use a layered approach with ProblemDetails (RFC 9457) as the standard format:

```csharp
builder.Services.AddProblemDetails();
app.UseExceptionHandler();
app.UseStatusCodePages();
```

Then customize with `IProblemDetailsService` or an exception handler:

```csharp
app.UseExceptionHandler(exceptionApp =>
{
    exceptionApp.Run(async context =>
    {
        var exception = context.Features.Get<IExceptionHandlerFeature>()?.Error;
        var response = exception switch
        {
            ValidationException ex => new ProblemDetails
            {
                Status = 400,
                Title = "Validation Failed",
                Extensions = { ["errors"] = ex.Errors }
            },
            NotFoundException => new ProblemDetails
            {
                Status = 404,
                Title = "Resource Not Found"
            },
            BusinessRuleException ex => new ProblemDetails
            {
                Status = 422,
                Title = "Business Rule Violation",
                Detail = ex.Message
            },
            _ => new ProblemDetails
            {
                Status = 500,
                Title = "Internal Server Error"
                // Never expose exception details in production
            }
        };

        context.Response.StatusCode = response.Status!.Value;
        await context.Response.WriteAsJsonAsync(response);
    });
});
```

The key insight: validation errors (400) are client's fault, business rule violations (422) are legitimate but rejected, and 500s are our fault. Each gets different detail levels. Never expose stack traces or internal error details in production.

I also pair this with the Result pattern in my service layer — services return `Result<T>` instead of throwing exceptions for expected failures.

> **Red Flag:** I catch all exceptions in a try-catch in every controller action." — Massive code duplication.

> ***Follow-up:*** *How does the Result pattern work? Show me how you'd use it in a service method and map it to an HTTP response.*

18 What happens when you call `builder.Services.AddScoped<IProductService, ProductService>()` twice with different implementations?

MID

Why interviewers ask this: Tests DI container knowledge beyond the basics.

GREAT ANSWER

The second registration wins — when you inject `IProductService`, you get the last registered implementation. But here's the catch: the first registration isn't removed. Both are in the container.

If you inject `IEnumerable<IProductService>`, you get both implementations. This is actually useful for the decorator pattern or chain of responsibility.

If you want to guarantee only one registration, use `TryAddScoped`:

```csharp
builder.Services.TryAddScoped<IProductService, ProductService>();
// This won't register if IProductService is already registered
```

This is how library authors prevent overriding user registrations — they use `TryAdd*` so the consumer's registration takes priority.

Red Flag: It throws an error." — It doesn't, which is why it's a subtle source of bugs.

Follow-up: *How would you implement the decorator pattern using DI — wrapping a service with logging or caching without the consuming code knowing?*

19 How does the ASP.NET Core request pipeline differ between Kestrel direct and Kestrel behind a reverse proxy (Nginx/Azure App Gateway)? What do you need to configure?

SENIOR

Why interviewers ask this: Production deployment awareness. Most APIs run behind a proxy.

GREAT ANSWER

Behind a reverse proxy, the client's real IP, scheme (HTTP/HTTPS), and host are lost. The proxy forwards them in headers like `X-Forwarded-For`, `X-Forwarded-Proto`, and `X-Forwarded-Host`.

Without configuring this, your API thinks every request comes from the proxy's IP (often `127.0.0.1`), and `HttpContext.Request.Scheme` is always `http` even if the client used HTTPS. This breaks: - IP-based rate limiting (everyone looks like the same IP) - HTTPS redirect loops (API sees HTTP, redirects to HTTPS, proxy forwards as HTTP again) - Logging (you log the proxy IP, not the client IP)

The fix:

```csharp
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwar
dedProto;
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.1")); // Your proxy IP
});

app.UseForwardedHeaders(); // Must be before UseAuthentication
```

In Azure App Service or container environments, you might also need to set `ASPNETCORE_FORWARDEDHEADERS_ENABLED=true`.

Red Flag: I just deploy to Kestrel directly in production." — Kestrel shouldn't be directly exposed to the internet.

Follow-up: *Why shouldn't Kestrel be exposed directly to the internet? What does a reverse proxy provide?*

20 You need to add a custom header to every API response — like `X-Request-Id` for request tracing. What's the best approach?

MID

Why interviewers ask this: Simple task, but reveals whether someone understands middleware lifecycle.

GREAT ANSWER

I'd write a simple middleware:

```csharp
app.Use(async (context, next) =>
{
    var requestId = context.TraceIdentifier; // ASP.NET Core already generates one
    context.Response.OnStarting(() =>
    {
        context.Response.Headers["X-Request-Id"] = requestId;
        return Task.CompletedTask;
    });
    await next();
});
```

The key detail: I use `OnStarting` instead of setting the header before `await next()`. Why? Because if downstream middleware changes the response (like a redirect), headers set before `next()` might get cleared. `OnStarting` runs right before headers are sent to the client — guaranteed to work.

I also add the same `requestId` to the logging scope so I can correlate logs with client-reported request IDs:

```csharp
using (logger.BeginScope(new Dictionary<string, object> { ["RequestId"] = requestId }))
{
    await next();
}
```

Red Flag: I'd set `context.Response.Headers` before calling `next()`." — Works most of the time but fragile.

Follow-up: *How does `Activity.Current?.Id` relate to `TraceIdentifier`? When would you use OpenTelemetry trace IDs instead?*

21 You notice that your API's memory usage keeps growing over time and never drops. What's your debugging approach?

SENIOR

Why interviewers ask this: Memory leak debugging is a senior-level skill that separates theory from practice.

GREAT ANSWER

I'd follow this systematic approach:

1. 1. **Confirm it's a leak, not just normal allocation.** Take a memory dump with `dotnet-dump collect` and compare two dumps taken 30 minutes apart.

1. 2. **Check the usual suspects first:**
2. - Singletons holding onto scoped services (captive dependency)
3. - Event handlers not being unsubscribed
4. - Static collections that grow forever (like an in-memory cache without eviction)
5. - `HttpClient` created with `new` instead of `IHttpClientFactory` (socket exhaustion, not memory, but often confused)
6. - Large object heap (LOH) fragmentation from large byte arrays

1. 3. **Analyze the dump:**

2.
   ```bash
3. dotnet-dump analyze dump.dmp
4. > dumpheap -stat # Shows object counts by type — look for unexpected growth
5. > gcroot <address> # Find what's keeping an object alive
6. ```

1. 4. **Use `dotnet-counters` for real-time monitoring:**
2. ```bash
3. dotnet-counters monitor --process-id 1234 System.Runtime
4. ```
5. Watch `gc-heap-size`, `gen-2-gc-count`, and `gen-2-size`.

1. 5. **In production**, I'd add `/health/memory` endpoint that reports GC stats so I can monitor trends without attaching a debugger.

**Red Flag:** I'd just restart the service periodically." — That's a workaround, not a fix.

***Follow-up:** What's the difference between a managed memory leak and a native memory leak in .NET?*
   ```

22 What is `IOptions<T>` vs `IOptionsSnapshot<T>` vs `IOptionsMonitor<T>`? When do you use each?

MID

Why interviewers ask this: Configuration is fundamental but the three interfaces confuse everyone.

GREAT ANSWER

INTERFACE	LIFETIME	RELOADS ON CHANGE?	USE WHEN
`IOptions<T>`	Singleton	No — reads config once at startup	Config that never changes at runtime
`IOptionsSnapshot<T>`	Scoped	Yes — per request	Config that might change (feature flags, limits)
`IOptionsMonitor<T>`	Singleton	Yes — immediately	Singletons that need live config updates

The practical rule: if you're in a scoped service (controller, endpoint handler), use `IOptionsSnapshot<T>`. If you're in a singleton (background service, hosted service), use `IOptionsMonitor<T>` with `CurrentValue`.

Common mistake: using `IOptions<T>` everywhere and then wondering why config changes from `appsettings.json` don't take effect without restarting the app.

CSHARP

```csharp
// In a scoped service — new value each request
public class ProductService(IOptionsSnapshot<PricingOptions> options)
{
    public decimal GetMarkup() => options.Value.MarkupPercentage;
}

// In a singleton — live updates
public class PricingWorker(IOptionsMonitor<PricingOptions> options)
{
    public decimal GetMarkup() => options.CurrentValue.MarkupPercentage;
}
```

⚠ **Red Flag:** I always use `IOptions<T>`, I didn't know there were other versions.

***Follow-up:** How do you validate options at startup using `ValidateDataAnnotations` or `ValidateOnStart`?*

23 How would you implement rate limiting in a .NET API? What rate limiting algorithm would you choose and why?

MID

Why interviewers ask this: Security and resource protection — every production API needs rate limiting.

GREAT ANSWER

.NET has built-in rate limiting since .NET 7. I'd use `Microsoft.AspNetCore.RateLimiting`:

```csharp
builder.Services.AddRateLimiter(options =>
{
    options.AddFixedWindowLimiter("api", opt =>
    {
        opt.Window = TimeSpan.FromMinutes(1);
        opt.PermitLimit = 100;
        opt.QueueLimit = 0; // Reject immediately, don't queue
    });

    options.RejectionStatusCode = StatusCodes.Status429TooManyRequests;
});

app.UseRateLimiter();
app.MapGet("/api/products", GetProducts).RequireRateLimiting("api");
```

Algorithm choice depends on the use case: - **Fixed window** — simple, good for most APIs. Weakness: burst at window boundaries (200 requests if 100 at end of window 1 + 100 at start of window 2) - **Sliding window** — smooths out the boundary burst problem. Better for APIs with consistent traffic - **Token bucket** — allows controlled bursts. Good for APIs where occasional spikes are OK (like search) - **Concurrency limiter** — limits simultaneous requests, not requests per time. Good for expensive endpoints (report generation)

For production, I'd use sliding window by default and partition by client IP or API key:

```csharp
options.AddSlidingWindowLimiter("api", opt =>
{
    opt.Window = TimeSpan.FromMinutes(1);
    opt.SegmentsPerWindow = 6; // 10-second segments
    opt.PermitLimit = 100;
});
```

I'd also return `Retry-After` header in the 429 response so clients know when to retry.

⚠ **Red Flag:** I'd build my own rate limiter with a dictionary." — Don't reinvent thread-safe infrastructure.

Follow-up: *How would you implement different rate limits for authenticated vs anonymous users?*

24 What's the difference between `app.Map`, `app.MapGet`, `app.MapPost` and `app.MapControllers` in ASP.NET Core?

JUNIOR

Why interviewers ask this: Tests understanding of minimal APIs vs controller-based APIs.

GREAT ANSWER

`MapGet`, `MapPost`, `MapPut`, `MapDelete`, `MapPatch` — these are minimal API endpoints. You define routes and handlers inline:

```csharp
app.MapGet("/api/products", () => "Hello");
app.MapPost("/api/products", (Product p) => Results.Created($"/api/products/{p.Id}",
p));
```

`MapControllers` registers all controller-based endpoints (classes decorated with `[ApiController]`).

`Map` is the generic version — it matches all HTTP methods for a path. Rarely used for APIs.

`MapGroup` is for grouping endpoints with shared prefixes, filters, or metadata:

```csharp
var group = app.MapGroup("/api/products").RequireAuthorization();
group.MapGet("/", GetAll);
group.MapPost("/", Create);
```

My preference: minimal APIs for new projects. They're simpler, faster (no reflection), and more explicit. Controllers still make sense for large APIs where you want the organizational structure.

Red Flag: I only use controllers, I haven't used minimal APIs." — Minimal APIs are the default in .NET templates since .NET 6.

***Follow-up:** Can you mix minimal APIs and controllers in the same project? When would you?*

03

Entity Framework Core & Data Access

12 questions 2 Junior 5 Mid 5 Senior

25 Your API endpoint returns 200 products but SQL Profiler shows 201 queries. What happened and how do you fix it?

SENIOR

Why interviewers ask this: The classic N+1 problem. Tests whether someone can diagnose performance issues.

GREAT ANSWER

That's the N+1 query problem. One query fetches 200 products, then EF Core lazily loads a navigation property (like `Category`) for each product individually — 200 extra queries.

How it happens:

```csharp
var products = await db.Products.ToListAsync(); // 1 query
foreach (var p in products)
{
    var name = p.Category.Name; // 200 queries — lazy loading each Category
}
```

Fixes, in order of preference:

1. 1. **Eager loading with `.Include()`:**

```csharp
var products = await db.Products
    .Include(p => p.Category)
    .ToListAsync(); // 1 query with JOIN
```

1. 2. **Projection (best performance):**

```csharp
var products = await db.Products
    .Select(p => new ProductDto
    {
        Name = p.Name,
        CategoryName = p.Category.Name
    })
    .ToListAsync(); // 1 query, minimal data
```

1. 3. **Split query (when Include causes cartesian explosion):**

```csharp
var products = await db.Products
    .Include(p => p.Category)
    .Include(p => p.Tags)
    .AsSplitQuery() // Separate queries instead of one massive JOIN
    .ToListAsync();
```

Detection: I always enable EF Core query logging in development and use `DbContextOptionsBuilder.LogTo()` or `Microsoft.EntityFrameworkCore.Query` log category to see generated SQL.

⚠ **Red Flag:** I'd disable lazy loading." — That masks the problem, doesn't fix it.

Follow-up: *When would `AsSplitQuery()` be better than the default single query? What's the cartesian explosion problem?*

26 How do you run EF Core migrations in a production CI/CD pipeline? Do you apply them at application startup?

MID

Why interviewers ask this: Migration strategy in production is where theory meets reality.

GREAT ANSWER

Never apply migrations at app startup in production. Here's why: if you have multiple instances (pods, app services), they'll all try to run the migration simultaneously. You'll get lock contention, timeouts, or duplicate migration attempts.

My approach:

1. 1. **Generate a SQL script from the migration:**

```bash
dotnet ef migrations script --idempotent -o migration.sql
```

1. 2. **Review the SQL** — never blindly apply generated migrations. Check for table locks, data loss, index creation on large tables.

1. 3. **Apply via CI/CD pipeline** — a dedicated migration step that runs before the application deployment:

```yaml
# GitHub Actions example
- name: Apply Migration
  run: dotnet ef database update --connection "${{ secrets.DB_CONNECTION }}"
```

1. 4. **Use idempotent scripts** (`--idempotent` flag) so re-running the same migration is safe.

For teams, I also recommend: never modify a migration after it's been applied to any environment. Create a new migration instead.

⚠ **Red Flag:** I call `db.Database.Migrate()` in `Program.cs`." — Dangerous in multi-instance deployments.

***Follow-up:** How do you handle a migration that needs to be rolled back in production?*

27 You have an endpoint that updates a product's price. Two users submit conflicting updates at the same time. How do you handle this?

SENIOR

Why interviewers ask this: Concurrency handling is critical for data integrity. Tests real-world experience.

GREAT ANSWER

This is a concurrency conflict. I'd use optimistic concurrency control with a concurrency token:

```csharp
public class Product
{
    public int Id { get; set; }
    public decimal Price { get; set; }

    [Timestamp]
    public byte[] RowVersion { get; set; } = null!;
}
```

EF Core automatically includes the `RowVersion` in the `WHERE` clause of UPDATE statements:

```sql
UPDATE Products SET Price = @newPrice
WHERE Id = @id AND RowVersion = @originalRowVersion
```

If another user changed the row since it was read, the `WHERE` clause matches zero rows, and EF Core throws `DbUpdateConcurrencyException`.

I handle it like this:

```csharp
try
{
    product.Price = request.NewPrice;
    await db.SaveChangesAsync();
}
catch (DbUpdateConcurrencyException)
{
    return Results.Conflict(new ProblemDetails
    {
        Title = "Concurrency Conflict",
        Detail = "This product was modified by another user. Please refresh and try agai
n."
    });
}
```

The client receives a `409 Conflict` and can reload the latest data.

Pessimistic locking (database-level row locks) is an alternative but doesn't scale well — it holds locks and blocks other readers/writers.

⚠ **Red Flag:** Last write wins — whoever saves last gets their change." — Data loss.

💬 ***Follow-up:*** *How does the client know what the current RowVersion is? How do you pass it through the API?*

28 What's the difference between `AsNoTracking()` and the default tracking behavior in EF Core? When do you use each?

MID

Why interviewers ask this: Performance optimization. Shows whether someone thinks about EF Core's change tracker.

GREAT ANSWER

By default, EF Core tracks every entity it queries. It keeps a snapshot of the original values so it can detect changes when you call `SaveChangesAsync()`. This tracking has a memory and CPU cost.

Use `AsNoTracking()` when: - Read-only queries (GET endpoints that just return data) - Reporting or dashboard queries - Any query where you won't modify the returned entities

```csharp
// Read-only — 15-20% faster, less memory
var products = await db.Products.AsNoTracking().ToListAsync();

// Need to update — keep tracking
var product = await db.Products.FindAsync(id);
product.Price = newPrice;
await db.SaveChangesAsync(); // Change tracker detects the Price change
```

For APIs where most endpoints are read-heavy, I configure `NoTracking` as the default:

```csharp
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking));
```

Then opt-in to tracking only for write operations: `.AsTracking()`.

⚠ **Red Flag:** I always use `AsNoTracking()` because it's faster." — Then how do you update entities?

***Follow-up:** What's `AsNoTrackingWithIdentityResolution()` and when would you use it instead of plain `AsNoTracking()`?*

29 How do you configure a one-to-many relationship in EF Core using Fluent API?

JUNIOR

Why interviewers ask this: Basic but necessary. EF Core configuration is daily work.

GREAT ANSWER

Say a `Category` has many `Products`:

```csharp
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public List<Product> Products { get; set; } = [];
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public int CategoryId { get; set; }        // FK property
    public Category Category { get; set; } = null!; // Navigation
}
```

Fluent API configuration in the entity configuration class:

```csharp
public class ProductConfiguration : IEntityTypeConfiguration<Product>
{
    public void Configure(EntityTypeBuilder<Product> builder)
    {
        builder.HasOne(p => p.Category)
               .WithMany(c => c.Products)
               .HasForeignKey(p => p.CategoryId)
               .OnDelete(DeleteBehavior.Restrict); // Don't cascade-delete products
    }
}
```

I always prefer Fluent API over data annotations because it keeps the domain model clean and all configuration is in one place. I also always specify `OnDelete` behavior explicitly — the default cascade delete can be dangerous.

⚠ **Red Flag:** I use `[ForeignKey]` attribute on the property." — Works but doesn't scale, and you lose explicit delete behavior configuration.

***Follow-up:** When would you use `DeleteBehavior.Restrict` vs `Cascade` vs `SetNull`?*

30 You need to soft-delete records instead of hard-deleting them. How do you implement this globally across all entities?

SENIOR

Why interviewers ask this: Tests ability to implement cross-cutting concerns in EF Core.

GREAT ANSWER

I'd use a global query filter so soft-deleted records are automatically excluded from all queries:

Step 1 — Base entity with `IsDeleted` flag:

CSHARP

```csharp
public interface ISoftDeletable
{
    bool IsDeleted { get; set; }
    DateTimeOffset? DeletedAt { get; set; }
}
```

Step 2 — Global query filter:

CSHARP

```csharp
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    foreach (var entityType in modelBuilder.Model.GetEntityTypes())
    {
        if (typeof(ISoftDeletable).IsAssignableFrom(entityType.ClrType))
        {
            modelBuilder.Entity(entityType.ClrType)
                .HasQueryFilter(
                    GenerateFilter(entityType.ClrType));
        }
    }
}
```

Step 3 — Override `SaveChangesAsync` to intercept deletes:

CSHARP

```csharp
public override Task<int> SaveChangesAsync(CancellationToken ct = default)
{
    foreach (var entry in ChangeTracker.Entries<ISoftDeletable>()
        .Where(e => e.State == EntityState.Deleted))
    {
        entry.State = EntityState.Modified;
        entry.Entity.IsDeleted = true;
        entry.Entity.DeletedAt = DateTimeOffset.UtcNow;
    }
    return base.SaveChangesAsync(ct);
}
```

When you need to include deleted records (admin panel, audit trail):

CSHARP

```csharp
var allProducts = await db.Products.IgnoreQueryFilters().ToListAsync();
```

Red Flag: I'd add `WHERE IsDeleted = false` to every query manually." — Error-prone, someone will forget.

***Follow-up:** How do global query filters interact with `Include`? Are related entities also filtered?*

31 You need to execute a raw SQL query in EF Core. When is this acceptable, and how do you prevent SQL injection?

MID

Why interviewers ask this: Raw SQL is sometimes necessary. Tests security awareness.

GREAT ANSWER

Raw SQL is acceptable when: - EF Core can't generate an efficient query (complex reporting, window functions) - You need database-specific features (full-text search, JSON queries) - Performance-critical queries where you need exact control

Safe approach — parameterized queries:

```csharp
// SAFE — parameterized via string interpolation (EF Core converts to parameters)
var products = await db.Products
    .FromSqlInterpolated($"SELECT * FROM Products WHERE Price > {minPrice}")
    .ToListAsync();

// SAFE — ExecuteUpdateAsync for bulk operations
await db.Products
    .Where(p => p.CategoryId == categoryId)
    .ExecuteUpdateAsync(s => s.SetProperty(p => p.IsActive, false));
```

Dangerous approach — string concatenation:

```csharp
// NEVER DO THIS — SQL injection vulnerability
var sql = $"SELECT * FROM Products WHERE Name = '{userInput}'";
await db.Products.FromSqlRaw(sql).ToListAsync();
```

The key: `FromSqlInterpolated` is safe (parameters), `FromSqlRaw` with string concatenation is dangerous. If you must use `FromSqlRaw`, pass parameters explicitly.

I'd also recommend: if you find yourself writing a lot of raw SQL, consider using Dapper alongside EF Core for read queries.

⚠ **Red Flag:** I concatenate user input into the SQL string." — SQL injection waiting to happen.

***Follow-up:** Can you mix EF Core LINQ and raw SQL? For example, apply `.Where()` on top of a `FromSql()` call?*

32 Your EF Core query works fine with 100 records but times out with 100,000. How do you diagnose and fix it?

SENIOR

Why interviewers ask this: Performance debugging at scale. Separates juniors from seniors.

GREAT ANSWER

Systematic debugging approach:

Step 1 — See the generated SQL:

```csharp
var query = db.Products.Where(p => p.Price > 100).Include(p => p.Category);
var sql = query.ToQueryString(); // Log this
```

Step 2 — Run the SQL in SSMS/pgAdmin with execution plan: Look for table scans (missing indexes), key lookups, and sorts.

Common culprits and fixes:

1. 1. **Missing index** — add an index on filtered/sorted columns:

```csharp
builder.HasIndex(p => p.Price);
```

1. 2. **Loading too much data** — use projection instead of loading full entities:

```csharp
// Bad — loads all columns of 100K rows
var products = await db.Products.ToListAsync();

// Good — only loads what you need
var products = await db.Products
    .Select(p => new { p.Id, p.Name, p.Price })
    .ToListAsync();
```

1. 3. **Cartesian explosion from multiple Includes:**

```csharp
// Bad — 100K products × 10 tags × 5 reviews = 5 million rows
db.Products.Include(p => p.Tags).Include(p => p.Reviews)

// Fix — split query
db.Products.Include(p => p.Tags).Include(p => p.Reviews).AsSplitQuery()
```

1. 4. **No pagination** — return all 100K records? Add pagination.

1. 5. **Compiled queries** for hot paths:

```csharp
private static readonly Func<AppDbContext, decimal, IAsyncEnumerable<Product>>
    GetExpensiveProducts = EF.CompileAsyncQuery(
        (AppDbContext db, decimal minPrice) =>
            db.Products.Where(p => p.Price > minPrice));
```

Red Flag: I'd increase the command timeout." — Hides the problem.

Follow-up: *When would you consider using Dapper instead of EF Core for a specific query?*

33 What are value converters in EF Core and when would you use them?

MID

Why interviewers ask this: Shows deeper EF Core knowledge beyond basic CRUD.

GREAT ANSWER

Value converters transform property values when reading from and writing to the database. The property type in your C# model can differ from the column type in the database.

Common use cases:

1. 1. **Enums as strings** (instead of integers):

```csharp
builder.Property(p => p.Status)
    .HasConversion<string>(); // Stores "Active" instead of 0
```

1. 2. **Strongly-typed IDs:**

```csharp
builder.Property(p => p.Id)
    .HasConversion(
        id => id.Value,       // ProductId → Guid (to database)
        value => new ProductId(value)); // Guid → ProductId (from database)
```

1. 3. **Encrypting sensitive data:**

```csharp
builder.Property(p => p.SSN)
    .HasConversion(
        v => Encrypt(v),     // Encrypt on save
        v => Decrypt(v));    // Decrypt on read
```

1. 4. **JSON columns** (storing complex objects as JSON in a single column):

```csharp
builder.Property(p => p.Metadata)
    .HasConversion(
        v => JsonSerializer.Serialize(v, default(JsonSerializerOptions)),
        v => JsonSerializer.Deserialize<Dictionary<string, string>>(v, default(JsonSeria
lizerOptions))!);
```

Gotcha: Value converters prevent EF Core from translating certain LINQ queries to SQL. If you filter by an enum stored as string, EF Core handles it. But complex converter logic might force client-side evaluation.

⚠ **Red Flag:** I haven't used them." — Missing a powerful EF Core feature.

***Follow-up:** How do value converters interact with migrations? What happens to existing data when you add a converter to an existing property?*

34 What's the difference between `FirstOrDefault()`, `SingleOrDefault()`, and `Find()` in EF Core?

JUNIOR

Why interviewers ask this: Basic but often confused. Performance implications differ.

GREAT ANSWER

METHOD	BEHAVIOR	SQL GENERATED
`FirstOrDefault()`	Returns first match or null	`SELECT TOP 1 ...`
`SingleOrDefault()`	Returns one match, throws if multiple	`SELECT TOP 2 ...`
`Find()`	Checks change tracker first, then database	Maybe no SQL at all

When to use each:

- - `Find(id)` — when looking up by primary key. It checks the change tracker first — if the entity was already loaded in this request, it returns the cached version without hitting the database. Best for single-entity lookups by PK.
- - `FirstOrDefault()` — when you expect multiple matches but only need the first one. Always hits the database (unless you compose it carefully).
- - `SingleOrDefault()` — when you expect zero or one match and want to enforce that constraint. It queries `TOP 2` to verify uniqueness — slightly more expensive than `FirstOrDefault`.

```csharp
// Best for PK lookup — uses change tracker
var product = await db.Products.FindAsync(id);

// Best for filtered lookup — returns first match
var product = await db.Products.FirstOrDefaultAsync(p => p.Sku == sku);

// Use when uniqueness matters — throws if duplicate
var user = await db.Users.SingleOrDefaultAsync(u => u.Email == email);
```

⚠ **Red Flag:** They're all the same, I just use `FirstOrDefault` everywhere.

***Follow-up:** What's the performance difference between `FindAsync` and `FirstOrDefaultAsync` when the entity is already tracked?*

35 How do you seed initial data in EF Core? What are the trade-offs of each approach? SENIOR

Why interviewers ask this: Data seeding strategy affects migrations, testing, and deployment.

GREAT ANSWER

Three approaches, each with different trade-offs:

1. `HasData()` in model configuration (migration-based):

```csharp
builder.HasData(
    new Category { Id = 1, Name = "Electronics" },
    new Category { Id = 2, Name = "Clothing" }
);
```

Pros: tracked in migrations, reproducible. Cons: requires hardcoded PKs, can't use navigation properties, becomes unwieldy for large datasets.

2. Custom initialization logic in `Program.cs` or a hosted service:

```csharp
using var scope = app.Services.CreateScope();
var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
if (!await db.Categories.AnyAsync())
{
    db.Categories.AddRange(seedData);
    await db.SaveChangesAsync();
}
```

Pros: flexible, can use computed values and navigation properties. Cons: runs at startup, not tracked in migrations.

3. SQL scripts in the migration:

```csharp
migrationBuilder.InsertData(...);
// or
migrationBuilder.Sql("INSERT INTO Categories ...");
```

Pros: full control, tracked in migrations. Cons: SQL-specific, can break on database provider changes.

My preference: `HasData()` for small reference data (countries, statuses, roles). Custom initialization for development seed data. SQL scripts for complex production data migrations.

⚠ **Red Flag:** I manually insert data in the database." — Not reproducible or testable.

***Follow-up:** How do you handle seed data that needs to be different per environment (dev vs production)?*

36 You need to execute a bulk update — setting `IsActive = false` for all products in a category. What's the most efficient way?

MID

Why interviewers ask this: Tests knowledge of EF Core 7+ bulk operations vs the old load-and-save pattern.

GREAT ANSWER

Before EF Core 7, you had to load all entities, modify them, and save — terrible for large datasets:

```csharp
// Old way — N queries + N updates. Don't do this.
var products = await db.Products.Where(p => p.CategoryId == 5).ToListAsync();
foreach (var p in products) p.IsActive = false;
await db.SaveChangesAsync();
```

Since EF Core 7, use `ExecuteUpdateAsync`:

```csharp
// New way — single SQL statement
await db.Products
    .Where(p => p.CategoryId == 5)
    .ExecuteUpdateAsync(s => s.SetProperty(p => p.IsActive, false));
```

This generates a single `UPDATE Products SET IsActive = 0 WHERE CategoryId = 5`. No entities loaded, no change tracking overhead.

Similarly, for bulk deletes:

```csharp
await db.Products
    .Where(p => p.IsDiscontinued)
    .ExecuteDeleteAsync();
```

Important caveats: - These bypass the change tracker — `SaveChanges` interceptors and events won't fire - Global query filters still apply (so soft-deleted records are excluded) - No cascade deletes through EF Core — the database must handle cascades

Red Flag: I'd load all the entities and update them in a loop." — Shows no awareness of bulk operations.

***Follow-up:** If you need audit logging on these bulk updates, how do you handle it since the change tracker is bypassed?*

04

Authentication & Security

11 questions 1 Junior 5 Mid 5 Senior

37 Your JWT access token is valid for 15 minutes. A user's role is changed from Admin to User by another admin. The old token still has the Admin claim. How do you handle this?

SENIOR

Why interviewers ask this: JWT stale claims is a real production problem. Tests security depth.

GREAT ANSWER

This is the fundamental JWT trade-off — JWTs are self-contained and can't be revoked once issued. Options:

1. Short-lived access tokens (my preferred approach): Keep access tokens short (5-15 minutes) and use refresh tokens. When the access token expires, the refresh token exchange checks the latest role from the database. The window of exposure is the access token lifetime.

2. Claims refresh on sensitive operations: For critical endpoints (admin actions, payment), re-validate the user's current roles from the database:

CSHARP

```
app.MapDelete("/api/admin/users/{id}", async (int id, ClaimsPrincipal user, UserService
users) =>
{
    // Don't trust the JWT claim for destructive operations
    var currentRoles = await users.GetRolesAsync(user.GetUserId());
    if (!currentRoles.Contains("Admin"))
        return Results.Forbid();
    // proceed
});
```

3. Token blocklist (last resort): Store revoked token IDs in Redis. Check on each request. This defeats the purpose of JWTs (stateless) but sometimes you need it for compliance.

My take: Short access tokens (5 min) + refresh tokens handles 95% of cases. Add real-time validation only for destructive operations. A full blocklist is overkill unless you have regulatory requirements.

Red Flag: JWT tokens can't be revoked, so there's nothing you can do." — True in theory, dangerous in practice.

***Follow-up:** How do refresh tokens work? Where do you store them, and how do you handle refresh token rotation?*

38 How do you implement authorization policies in ASP.NET Core beyond simple role checks?

MID

Why interviewers ask this: Role-based auth is the basics. Policy-based auth is where real-world complexity lives.

GREAT ANSWER

Roles are coarse-grained ("Admin", "User"). Policies let you combine multiple requirements:

```csharp
builder.Services.AddAuthorizationBuilder()
    .AddPolicy("CanEditProduct", policy =>
        policy.RequireRole("Admin", "ProductManager")
              .RequireClaim("department", "inventory")
              .AddRequirements(new MinimumAgeRequirement(18)));
```

For complex logic, implement `IAuthorizationRequirement` and `IAuthorizationHandler`:

```csharp
public class ResourceOwnerRequirement : IAuthorizationRequirement;

public class ResourceOwnerHandler : AuthorizationHandler<ResourceOwnerRequirement, Produ
ct>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        ResourceOwnerRequirement requirement,
        Product resource)
    {
        if (resource.CreatedByUserId == context.User.GetUserId())
            context.Succeed(requirement);

        return Task.CompletedTask;
    }
}
```

Then use resource-based authorization:

```csharp
app.MapPut("/api/products/{id}", async (int id, IAuthorizationService auth, ClaimsPrinci
pal user) =>
{
    var product = await db.Products.FindAsync(id);
    var result = await auth.AuthorizeAsync(user, product, new ResourceOwnerRequirement
());
    if (!result.Succeeded) return Results.Forbid();
    // update
});
```

This lets you answer questions like "can THIS user edit THIS product?" — not just "is this user an admin?"

⚠ **Red Flag:** I check `User.IsInRole()` in every endpoint." — Doesn't scale, duplicates logic.

Follow-up: *How do you unit test authorization handlers?*

39 You're building an API that will be called from a React SPA on a different domain. What CORS configuration do you need, and what are the security implications?

MID

Why interviewers ask this: CORS misconfiguration is a top-10 web security issue.

GREAT ANSWER

```csharp
builder.Services.AddCors(options =>
{
    options.AddPolicy("frontend", policy =>
    {
        policy.WithOrigins("https://app.example.com") // Specific origin, NOT "*"
              .WithMethods("GET", "POST", "PUT", "DELETE")
              .WithHeaders("Content-Type", "Authorization")
              .AllowCredentials(); // Needed for cookies/auth headers
    });
});

app.UseCors("frontend"); // Before auth middleware!
```

Critical security rules:

1. 1. **Never use `AllowAnyOrigin()` with `AllowCredentials()`** — the browser blocks this, and for good reason. It would allow any website to make authenticated requests to your API.
1. 2. **Never use `AllowAnyOrigin()` in production at all** unless it's a truly public API with no authentication.
1. 3. **Be specific about methods and headers** — don't allow everything.
1. 4. **Preflight caching** — browsers send an `OPTIONS` request before the actual request. Cache it:

```csharp
policy.SetPreflightMaxAge(TimeSpan.FromHours(1));
```

Common mistake I've seen: CORS configured correctly in `Program.cs` but a reverse proxy strips the CORS headers. Always test CORS from the actual browser, not Postman (Postman doesn't enforce CORS).

⚠ **Red Flag:** I just use `AllowAnyOrigin` and `AllowAnyMethod`." — Open security hole.

***Follow-up:** How does CORS work with cookie-based authentication vs JWT bearer tokens?*

40 How do you securely store connection strings, API keys, and other secrets in a .NET application across different environments?

SENIOR

Why interviewers ask this: Secret management is fundamental. Hardcoded secrets in source code is a career-ending mistake.

GREAT ANSWER

Layer the secret management by environment:

Development:

```bash
dotnet user-secrets set "Database:ConnectionString" "Server=localhost;..."
```

User secrets are stored in `%APPDATA%\Microsoft\UserSecrets\<guid>\secrets.json` — never in source control.

Production (cloud): - **Azure:** Azure Key Vault with `Azure.Extensions.AspNetCore.Configuration.Secrets`

```csharp
builder.Configuration.AddAzureKeyVault(
    new Uri("https://myvault.vault.azure.net/"),
    new DefaultAzureCredential());
```

- - **AWS:** AWS Secrets Manager or Parameter Store
- - **Docker/K8s:** Kubernetes secrets mounted as environment variables

What NOT to do: - Never commit secrets to `appsettings.json` (even `appsettings.Development.json`) - Never hardcode connection strings in source code - Never log secrets (use `[LogPropertyIgnore]` or mask them)

Environment variable approach (12-factor app):

```csharp
// appsettings.json has placeholder
"ConnectionStrings": { "Default": "" }
// Environment variable overrides it
// CONNECTIONSTRINGS__DEFAULT=Server=prod;...
```

My production setup: Azure Key Vault for secrets, environment variables for non-sensitive config, `appsettings.{Environment}.json` for environment-specific settings that aren't secrets.

⚠ **Red Flag:** I put them in `appsettings.json` and add it to `.gitignore`." — Someone will commit it eventually.

***Follow-up:** How does the .NET configuration system's precedence work? What overrides what?*

41 Your API needs to support both JWT bearer tokens and API key authentication. How do you configure multiple authentication schemes?

MID

Why interviewers ask this: Real APIs often need multiple auth mechanisms.

GREAT ANSWER

CSHARP

```csharp
builder.Services.AddAuthentication()
    .AddJwtBearer("Bearer", options ⇒ { /* JWT config */ })
    .AddScheme<ApiKeyAuthOptions, ApiKeyAuthHandler>("ApiKey", options ⇒ { });

builder.Services.AddAuthorizationBuilder()
    .SetDefaultPolicy(new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .AddAuthenticationSchemes("Bearer", "ApiKey")
        .Build());
```

The API key handler is a custom `AuthenticationHandler<T>`:

CSHARP

```csharp
public class ApiKeyAuthHandler : AuthenticationHandler<ApiKeyAuthOptions>
{
    protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        if (!Request.Headers.TryGetValue("X-API-Key", out var key))
            return AuthenticateResult.NoResult(); // Let other schemes try

        var apiKey = await ValidateKey(key!);
        if (apiKey is null)
            return AuthenticateResult.Fail("Invalid API key");

        var claims = new[] { new Claim(ClaimTypes.Name, apiKey.ClientName) };
        var identity = new ClaimsIdentity(claims, Scheme.Name);
        return AuthenticateResult.Success(new AuthenticationTicket(
            new ClaimsPrincipal(identity), Scheme.Name));
    }
}
```

Key detail: returning `AuthenticateResult.NoResult()` (not `Fail`) when the header is missing lets the next scheme (JWT) try. `Fail` would short-circuit.

⚠ **Red Flag:** I'd check for the API key manually in middleware before the auth middleware runs." — Bypasses the authentication pipeline.

***Follow-up:** How do you rate-limit API key clients differently than JWT-authenticated users?*

42 How do you prevent mass assignment (over-posting) attacks in your API?

SENIOR

Why interviewers ask this: Common vulnerability where clients send unexpected fields to elevate privileges.

GREAT ANSWER

Mass assignment happens when you bind request data directly to your entity model. A client could send `{"name": "test", "isAdmin": true}` and promote themselves.

Prevention — always use DTOs:

CSHARP

```csharp
// DANGEROUS — binding directly to entity
app.MapPost("/api/users", async (User user, AppDbContext db) => { ... });

// SAFE — binding to a DTO with only allowed fields
public record CreateUserRequest(string Name, string Email);

app.MapPost("/api/users", async (CreateUserRequest request, AppDbContext db) =>
{
    var user = new User
    {
        Name = request.Name,
        Email = request.Email,
        IsAdmin = false // Explicitly set, never from client
    };
    db.Users.Add(user);
    await db.SaveChangesAsync();
});
```

Rules: 1. Never bind directly to entity classes 2. Create separate DTOs for create, update, and response 3. Map explicitly — don't use auto-mapping from request to entity without controlling which fields are mapped 4. Validate that readonly fields (CreatedAt, Id, IsAdmin) can't be set from the client

⚠ **Red Flag:** I use `[Bind]` attribute to exclude fields." — Fragile, easy to forget a field.

***Follow-up:** How does this interact with PATCH endpoints? How do you prevent over-posting on partial updates?*

43 What's the difference between authentication and authorization? Give a real example from an API.

JUNIOR

Why interviewers ask this: Foundational security concepts. Many junior devs confuse them.

GREAT ANSWER

Authentication = "Who are you?" — proving your identity. **Authorization** = "What are you allowed to do?" — checking your permissions.

Real API example:

```
POST /api/login { email, password }
→ Server verifies credentials (AUTHENTICATION)
→ Returns JWT token with claims: { userId: 42, role: "Editor" }

GET /api/admin/users (with JWT token)
→ Server validates the token is real (AUTHENTICATION ✓)
→ Server checks: does "Editor" role have access to /admin/users? (AUTHORIZATION ✗)
→ Returns 403 Forbidden
```

In ASP.NET Core: - `app.UseAuthentication()` — extracts identity from the token - `app.UseAuthorization()` — checks if that identity has permission

Status codes tell you which failed: - `401 Unauthorized` — actually means "unauthenticated" (bad or missing token) - `403 Forbidden` — authenticated but not authorized (valid token, insufficient permissions)

Yes, the HTTP status code names are misleading. `401` should really be called "Unauthenticated."

⚠ **Red Flag:** They're the same thing — checking if the user can access the endpoint.

***Follow-up:** In which order do you register `UseAuthentication()` and `UseAuthorization()`, and why?*

44 How do you implement API key authentication that's actually secure — not just checking a string against a config value?

SENIOR

Why interviewers ask this: Many developers implement API keys poorly. Tests production security thinking.

GREAT ANSWER

A naive implementation stores API keys in `appsettings.json` and compares strings. This has problems: no auditing, no per-key permissions, timing attacks on string comparison, and you can't rotate keys without redeploying.

Production-grade approach:

1. 1. **Store hashed keys in the database:**

```csharp
public class ApiKey
{
    public int Id { get; set; }
    public string HashedKey { get; set; } = string.Empty; // SHA-256 of the actual key
    public string ClientName { get; set; } = string.Empty;
    public List<string> Scopes { get; set; } = []; // What this key can do
    public DateTimeOffset? ExpiresAt { get; set; }
    public bool IsRevoked { get; set; }
}
```

1. 2. **Generate keys properly:** Use `RandomNumberGenerator.GetBytes(32)` for cryptographically random keys. Show the key once at creation, store only the hash.

1. 3. **Use constant-time comparison** to prevent timing attacks:

```csharp
CryptographicOperations.FixedTimeEquals(
    SHA256.HashData(Encoding.UTF8.GetBytes(providedKey)),
    Convert.FromBase64String(storedHash));
```

1. 4. **Rate-limit failed attempts** and log them.

1. 5. **Support key rotation:** Each client can have multiple active keys, so they can generate a new one before revoking the old one.

Red Flag: I compare the key with == against a string in my config." — Timing attacks, no rotation, no auditing.

***Follow-up:** How would you implement API key scopes — like a key that can only read products but not delete them?*

45 Your API returns user data. How do you ensure sensitive fields (password hash, SSN, internal IDs) never appear in API responses?

MID

Why interviewers ask this: Data leakage prevention. A common security oversight.

GREAT ANSWER

The only reliable approach: response DTOs.

```csharp
// Entity (internal)
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string PasswordHash { get; set; } // NEVER expose
    public string InternalNotes { get; set; } // NEVER expose
}

// Response DTO (what the client sees)
public record UserResponse(int Id, string Name, string Email);

// Endpoint
app.MapGet("/api/users/{id}", async (int id, AppDbContext db) =>
{
    var user = await db.Users
        .Where(u => u.Id == id)
        .Select(u => new UserResponse(u.Id, u.Name, u.Email)) // Projection
        .FirstOrDefaultAsync();

    return user is null ? Results.NotFound() : Results.Ok(user);
});
```

Why not `[JsonIgnore]`?

```csharp
public class User
{
    [JsonIgnore] public string PasswordHash { get; set; } // Fragile!
}
```

This works but is fragile — one developer forgets the attribute, or serialization settings change, and you leak data. DTOs make it impossible to accidentally expose fields.

Additional safeguards: - Use projection (`.Select()`) to avoid loading sensitive data from the database at all - Add response serialization tests that verify sensitive fields aren't present - Never return entity objects directly from endpoints

⚠ **Red Flag:** I use `[JsonIgnore]` on the entity." — Works until someone bypasses it.

***Follow-up:** How would you handle a scenario where admin users should see more fields than regular users?*

46 What are the most common security headers your API should return, and how do you configure them?

MID

Why interviewers ask this: Security headers are low effort, high impact. Shows production awareness.

GREAT ANSWER

```csharp
app.Use(async (context, next) =>
{
    context.Response.Headers["X-Content-Type-Options"] = "nosniff"; // Prevent MIME sniffing
    context.Response.Headers["X-Frame-Options"] = "DENY"; // Prevent clickjacking
    context.Response.Headers["X-XSS-Protection"] = "0"; // Disable browser XSS filter (it's buggy)
    context.Response.Headers["Referrer-Policy"] = "strict-origin-when-cross-origin";
    context.Response.Headers["Permissions-Policy"] = "camera=(), microphone=()";
    await next();
});

// HSTS for HTTPS enforcement
app.UseHsts();
```

For APIs specifically: - `X-Content-Type-Options: nosniff` — prevents browsers from guessing MIME types - `Strict-Transport-Security` (via `UseHsts()`) — forces HTTPS - **Remove `Server` header** — don't advertise you're running Kestrel:

```csharp
builder.WebHost.ConfigureKestrel(options => options.AddServerHeader = false);
```

I'd also configure the `Content-Security-Policy` if the API serves any HTML (error pages, Scalar UI).

Red Flag: APIs don't need security headers, that's a frontend thing." — APIs are still HTTP, headers still matter.

***Follow-up:** How do you verify security headers are correctly configured? What tools do you use?*

47 How do you protect your API against common attacks like SQL injection, XSS, and CSRF?

SENIOR

Why interviewers ask this: OWASP awareness. Tests breadth of security knowledge.

GREAT ANSWER

SQL Injection: - Use EF Core or parameterized queries — never concatenate user input into SQL - If using raw SQL, use `FromSqlInterpolated` (parameterized) not `FromSqlRaw` with string concatenation - EF Core's LINQ queries are safe by default

XSS (Cross-Site Scripting): - For APIs returning JSON, the risk is lower than HTML — JSON serialization escapes by default - But if your API returns HTML (error messages, emails), sanitize output - Never return raw user input in error messages: `$"User {username} not found"` — if `username` contains script tags, you have XSS - Use `HtmlEncoder.Default.Encode()` for any user input rendered as HTML

CSRF (Cross-Site Request Forgery): - For APIs using JWT bearer tokens: CSRF is not a concern because the browser doesn't automatically send the token (unlike cookies) - For APIs using cookie authentication: add anti-forgery tokens or use `SameSite=Strict` cookies - For SPAs: `SameSite=Lax` on auth cookies + verify `Origin` header

Additional protections: - Rate limiting on auth endpoints - Input validation with FluentValidation on all endpoints - Request size limits - Disable detailed error messages in production

Red Flag: EF Core handles all of that automatically." — EF Core helps with SQL injection, not XSS or CSRF.

***Follow-up:** What's the difference between SameSite=Strict and SameSite=Lax for cookies?*

05

Performance & Caching

11 questions 1 Junior 5 Mid 5 Senior

48 Your API endpoint takes 800ms to respond. How do you systematically diagnose and reduce the latency?

SENIOR

Why interviewers ask this: Performance debugging is senior-level. Shows systematic thinking.

GREAT ANSWER

I'd follow a layered approach — find the bottleneck before optimizing:

Step 1 — Measure, don't guess: Add timing middleware or use OpenTelemetry to see where time is spent:

```csharp
app.Use(async (context, next) =>
{
    var sw = Stopwatch.StartNew();
    await next();
    sw.Stop();
    logger.LogInformation("Request {Path} took {Duration}ms",
        context.Request.Path, sw.ElapsedMilliseconds);
});
```

Step 2 — Common bottleneck checklist:

SUSPECT	HOW TO CHECK	FIX
Database queries	EF Core logging, SQL Profiler	Add indexes, use projection, fix N+1
External API calls	HttpClient logging	Add caching, make parallel with `Task.WhenAll`
Serialization	Large response payloads	Paginate, use projection, compress
Auth middleware	Token validation hitting database	Cache token validation, use asymmetric key validation
Memory allocation	`dotnet-counters` GC metrics	Reduce allocations, use `Span<T>`

Step 3 — Apply the appropriate fix:

If it's database: caching with `HybridCache`, indexes, compiled queries. If it's external calls: `Task.WhenAll` for parallel requests, resilience policies. If it's serialization: response compression, `System.Text.Json` source generators.

Real example: I once traced an 800ms endpoint to two sequential HTTP calls (300ms each) plus a database query (200ms). Made the HTTP calls parallel with `Task.WhenAll` — dropped to 500ms. Added response caching — dropped to 50ms for repeated requests.

⚠ **Red Flag:** I'd add caching." — Caching what? You haven't found the bottleneck yet.

***Follow-up:** How would you set up continuous performance monitoring so you catch regressions before users notice?*

49 What's the difference between output caching, response caching, and `HybridCache` in .NET? When do you use each?

MID

Why interviewers ask this: Caching is multi-layered. Tests whether someone picks the right tool.

GREAT ANSWER

FEATURE	WHERE IT CACHES	WHO RESPECTS IT	BEST FOR
Response Caching	Client/CDN (HTTP headers)	Browser, CDN, proxy	Static content, public endpoints
Output Caching	Server (in-memory)	ASP.NET Core middleware	Server-side response caching with tag-based invalidation
HybridCache	Server (memory + distributed)	Your application code	Application-level data caching

Response Caching — sets HTTP headers (`Cache-Control`, `Expires`). The browser or CDN caches the response. You have no server-side control over invalidation.

```csharp
app.MapGet("/api/products", () => ...).CacheOutput(p => p.Expire(TimeSpan.FromMinutes
(5)));
```

Output Caching — caches the full HTTP response on the server. Supports tag-based invalidation:

```csharp
app.MapGet("/api/products", GetProducts).CacheOutput("products");
app.MapPost("/api/products", async (IOutputCacheStore cache) =>
{
    // Create product...
    await cache.EvictByTagAsync("products", default); // Invalidate cache
});
```

HybridCache (.NET 9+) — application-level caching with stampede protection:

```csharp
var product = await cache.GetOrCreateAsync($"product-{id}",
    async ct => await db.Products.FindAsync(id, ct),
    new HybridCacheEntryOptions { Expiration = TimeSpan.FromMinutes(10) });
```

My rule: Use output caching for read-heavy API endpoints. Use HybridCache for expensive business logic or database queries that multiple endpoints share. Use response caching headers for content that CDNs should cache.

Red Flag: I use `IMemoryCache` for everything." — No distributed support, no stampede protection.

Follow-up: *What is cache stampede and how does HybridCache prevent it?*

50 You have three independent async operations: fetching user profile, fetching user orders, and fetching user recommendations. Each takes ~200ms. How do you optimize this endpoint?

SENIOR

Why interviewers ask this: Async parallelism. The difference between 600ms and 200ms.

GREAT ANSWER

Run them in parallel with `Task.WhenAll`:

```csharp
app.MapGet("/api/users/{id}/dashboard", async (int id) =>
{
    var profileTask = userService.GetProfileAsync(id);
    var ordersTask = orderService.GetRecentOrdersAsync(id);
    var recommendationsTask = recommendationService.GetAsync(id);

    await Task.WhenAll(profileTask, ordersTask, recommendationsTask);

    return Results.Ok(new DashboardResponse
    {
        Profile = profileTask.Result,
        RecentOrders = ordersTask.Result,
        Recommendations = recommendationsTask.Result
    });
});
```

This takes ~200ms (the slowest task) instead of ~600ms (sequential).

Important caveats:

1. 1. **Error handling** — if one task fails, `Task.WhenAll` throws the first exception. If you want partial results:

```csharp
try { await Task.WhenAll(tasks); }
catch { /* Handle individual task failures */ }

var profile = profileTask.IsCompletedSuccessfully ? profileTask.Result : null;
```

1. 2. **Don't overdo it** — parallel database queries on the same `DbContext` don't work. DbContext is not thread-safe. You'd need separate scopes or separate DbContext instances.

1. 3. `Task.WhenAll` **vs** `Parallel.ForEachAsync` — `WhenAll` is for a known set of async tasks. `Parallel.ForEachAsync` is for processing a collection with controlled concurrency.

⚠ **Red Flag:** I'd just await them one by one." — 3x slower for no reason.

***Follow-up:** What happens if you try to use the same DbContext in parallel tasks? How do you solve it?*

51 What is response compression and when should you NOT use it?

MID

Why interviewers ask this: Performance optimization with security trade-offs.

GREAT ANSWER

Response compression (gzip, brotli) reduces payload size, typically 60-80% for JSON/text responses.

```csharp
builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true; // Read the caveat below
    options.Providers.Add<BrotliCompressionProvider>(); // Better ratio
    options.Providers.Add<GzipCompressionProvider>();   // Wider support
});

app.UseResponseCompression(); // Before static files and endpoints
```

When NOT to use it:

1. 1. **Small responses (< 1KB)** — compression overhead exceeds savings
2. 2. **Already compressed content** — images, videos, PDFs. You'll waste CPU for zero benefit
3. 3. **HTTPS security concern (BREACH/CRIME attacks)** — compression + HTTPS + user-controlled input in response can leak secrets through response size analysis. For APIs returning only JSON data (not HTML with tokens), this risk is low.
4. 4. **When a reverse proxy handles it** — don't compress twice. If Nginx/Cloudflare compresses, disable it in Kestrel.
5. 5. **Real-time/streaming endpoints** — compression buffers data, adding latency.

My rule: Enable Brotli compression for API responses > 1KB, disable for binary content and streaming endpoints. If behind a CDN, let the CDN handle compression.

Red Flag: Always enable it, it makes everything faster." — Not for small responses, binary content, or streaming.

***Follow-up:** How do you configure different compression levels for different endpoints?*

52 What common async/await mistakes do you see in .NET APIs that hurt performance?

MID

Why interviewers ask this: Async is easy to get wrong. Shows real experience.

GREAT ANSWER

The top mistakes I've seen (and made):

1. Unnecessary `async/await` wrapping:

```csharp
// Bad — adds state machine overhead for nothing
public async Task<Product> GetAsync(int id)
    => await db.Products.FindAsync(id);

// Good — just return the task
public Task<Product?> GetAsync(int id)
    => db.Products.FindAsync(id).AsTask();
```

2. `Task.Result` or `.Wait()` causing deadlocks:

```csharp
// DEADLOCK in ASP.NET (synchronization context blocks)
var product = GetProductAsync(id).Result; // DON'T

// Correct
var product = await GetProductAsync(id);
```

3. Forgetting `ConfigureAwait(false)` in library code: Not relevant in ASP.NET Core (no sync context), but critical in libraries that might be used in UI apps.

4. Fire-and-forget without error handling:

```csharp
// Bad — exception silently swallowed
_ = SendEmailAsync(user);

// Better — at least log errors
_ = Task.Run(async () =>
{
    try { await SendEmailAsync(user); }
    catch (Exception ex) { logger.LogError(ex, "Email failed"); }
});

// Best — use a proper background service or message queue
```

5. Sequential awaits when tasks are independent (covered in Q50).

6. `async void` — never use in APIs. Exceptions crash the process.

⚠ **Red Flag:** I don't use async, it's complicated." — Every I/O operation in an API should be async.

> ***Follow-up:*** *What's the performance cost of `async/await`? When would you avoid it for performance reasons?*

53 You need to stream a large CSV export (100K+ rows) from your API without loading everything into memory. How?

SENIOR

Why interviewers ask this: Streaming is advanced. Tests memory management knowledge.

GREAT ANSWER

Use `IAsyncEnumerable<T>` to stream rows and write directly to the response:

```csharp
app.MapGet("/api/products/export", async (AppDbContext db, HttpContext http) =>
{
    http.Response.ContentType = "text/csv";
    http.Response.Headers.ContentDisposition = "attachment; filename=products.csv";

    await using var writer = new StreamWriter(http.Response.Body);
    await writer.WriteLineAsync("Id,Name,Price,Category");

    await foreach (var product in db.Products
        .AsNoTracking()
        .Select(p => new { p.Id, p.Name, p.Price, Category = p.Category.Name })
        .AsAsyncEnumerable())
    {
        await writer.WriteLineAsync($"{product.Id},{Escape(product.Name)},{product.Pric
e},{Escape(product.Category)}");

        if (/* every 1000 rows */) await writer.FlushAsync(); // Flush periodically
    }
});
```

Key decisions: 1. `AsAsyncEnumerable()` — EF Core streams rows from the database instead of loading all 100K into memory 2. **Write directly to** `Response.Body` — no intermediate buffering 3. **Disable response buffering** — `http.Response.Headers["Transfer-Encoding"] = "chunked"` 4. **Periodic flushing** — don't wait until all rows are written

Memory usage: Instead of loading 100K objects (potentially hundreds of MB), this uses constant memory regardless of dataset size.

Red Flag: I'd load all products into a list, build a CSV string, and return it." — Out of memory at scale.

***Follow-up:** How would you add progress reporting or cancellation support to this streaming export?*

54 What's the difference between `System.Text.Json` and `Newtonsoft.Json`? When would you still choose Newtonsoft?

MID

Why interviewers ask this: Serialization choice affects performance and features.

GREAT ANSWER

ASPECT	SYSTEM.TEXT.JSON	NEWTONSOFT.JSON
Performance	2-3x faster, fewer allocations	Slower but more flexible
Default in ASP.NET Core	Yes (since .NET Core 3.0)	Needs `AddNewtonsoftJson()`
Source generators	Yes — AOT-friendly, even faster	No
Polymorphic serialization	Supported (with `[JsonDerivedType]`)	Built-in and easier
Custom converters	More verbose API	Simpler `JsonConverter<T>` API
Missing member handling	`JsonUnmappedMemberHandling` (.NET 8+)	`MissingMemberHandling`

When I'd still use Newtonsoft: 1. **JSON Patch** — `Microsoft.AspNetCore.JsonPatch` requires Newtonsoft 2. **Complex polymorphic deserialization** — Newtonsoft handles `TypeNameHandling` (but be careful: it's a security risk if deserializing untrusted input) 3. **Legacy codebases** — migrating all custom converters at once is risky

For new projects: System.Text.Json, always. With source generators for hot paths:

```csharp
[JsonSerializable(typeof(ProductResponse))]
internal partial class AppJsonContext : JsonSerializerContext;
```

This eliminates reflection-based serialization — important for AOT and trimming.

⚠ **Red Flag:** Newtonsoft is better because I'm more familiar with it." — That's a personal preference, not a technical reason.

***Follow-up:** How do System.Text.Json source generators work, and what's the performance improvement?*

55 Your API serves both real-time data (stock prices) and infrequently changing data (product catalog). How do you design the caching strategy for each?

SENIOR

Why interviewers ask this: Tests ability to match caching strategy to data characteristics.

GREAT ANSWER

Different data = different caching strategies:

Product catalog (changes hourly/daily):

CSHARP

```
// Output caching with tag-based invalidation
app.MapGet("/api/products", GetProducts)
    .CacheOutput(p => p.Expire(TimeSpan.FromMinutes(30)).Tag("products"));

// Invalidate when products change
app.MapPost("/api/products", async (IOutputCacheStore cache) =>
{
    // ... create product
    await cache.EvictByTagAsync("products", default);
});
```

Plus HTTP cache headers for CDN/browser caching:

CSHARP

```
context.Response.Headers.CacheControl = "public, max-age=1800, stale-while-revalidate=6
0";
```

Stock prices (changes every second): - **No HTTP caching** — data is always stale - **Use SignalR/WebSockets** for real-time push instead of polling - **If polling is required:** short-lived HybridCache (5-10 seconds) to prevent database hammering when 1000 clients poll simultaneously

CSHARP

```
var price = await hybridCache.GetOrCreateAsync(
    $"stock-{symbol}",
    async ct => await stockService.GetLatestPrice(symbol, ct),
    new HybridCacheEntryOptions { Expiration = TimeSpan.FromSeconds(5) });
```

Key insight: The caching layer depends on data volatility AND access patterns. High-read, low-write data (catalog) = aggressive caching. High-volatility data (prices) = push-based or micro-cache.

⚠ **Red Flag:** Cache everything for 5 minutes." — One size doesn't fit all.

Follow-up: *How would you implement cache warming for the product catalog after a deployment?*

56 What is the `CancellationToken` and why should every async API endpoint accept one?

MID

Why interviewers ask this: Shows understanding of request lifecycle and resource management.

GREAT ANSWER

The `CancellationToken` signals when a client disconnects or the request is aborted. Without it, your server keeps working on a request nobody is waiting for.

```csharp
app.MapGet("/api/reports", async (AppDbContext db, CancellationToken ct) =>
{
    var report = await db.Reports
        .Where(r => r.IsActive)
        .ToListAsync(ct); // Cancels the DB query if client disconnects

    return Results.Ok(report);
});
```

Why it matters: 1. Client navigates away → request is cancelled → server stops the DB query immediately 2. Load balancer timeout → request cancelled → server frees resources 3. Deployment/shutdown → `ApplicationStopping` token fires → graceful termination

Common mistakes: - Not passing it to EF Core queries, HTTP calls, or I/O operations - Catching `OperationCanceledException` and treating it as a real error (it's not — the client left)

```csharp
// Let cancellation exceptions propagate — ASP.NET Core handles them
// DON'T do this:
catch (OperationCanceledException) { logger.LogError("Something went wrong!"); }
```

Propagate it everywhere:

```csharp
public async Task<Product?> GetAsync(int id, CancellationToken ct)
{
    var data = await httpClient.GetAsync($"/external/{id}", ct);
    return await db.Products.FindAsync([id], ct);
}
```

⚠ **Red Flag:** I've never used CancellationToken in my endpoints." — Wasting server resources.

***Follow-up:** How do you handle cancellation in a background service that's processing a queue?*

57 What does `async` and `await` actually do in C#? Why not just use synchronous code?

JUNIOR

Why interviewers ask this: Fundamental understanding of async in .NET.

GREAT ANSWER

`async/await` frees up the thread while waiting for I/O operations (database queries, HTTP calls, file reads). The thread goes back to the thread pool and can handle other requests.

Without async:

```csharp
// Thread is BLOCKED for 200ms — doing nothing, just waiting
var data = httpClient.GetString("https://api.example.com"); // Synchronous
```

With async:

```csharp
// Thread is RELEASED during the 200ms wait — handles other requests
var data = await httpClient.GetStringAsync("https://api.example.com");
```

Why this matters for APIs: A web server has a limited thread pool (default ~100 threads). If every request blocks a thread for 200ms, you can only handle ~500 requests/second. With async, those threads handle other requests during the wait — you might handle 10,000+ requests/second with the same thread pool.

Key distinction: `async` doesn't make code faster. A single request still takes 200ms. But it makes the server handle MORE concurrent requests because threads aren't wasted waiting.

When NOT to use async: CPU-bound work (calculations, JSON parsing). Async only helps with I/O-bound work. For CPU work, use `Task.Run()` to move it off the request thread.

⚠ **Red Flag:** Async makes code run faster." — It doesn't. It improves throughput, not latency.

***Follow-up:** What's the thread pool in .NET and how does it manage threads? What happens when all threads are exhausted?*

58 How do you benchmark .NET code properly? What mistakes do developers make when writing benchmarks?

SENIOR

Why interviewers ask this: Shows engineering rigor and data-driven decision making.

GREAT ANSWER

Use BenchmarkDotNet — never use `Stopwatch` for benchmarks:

```csharp
[MemoryDiagnoser]
public class SerializationBenchmarks
{
    private readonly Product _product = new() { Name = "Test", Price = 99.99m };

    [Benchmark(Baseline = true)]
    public string SystemTextJson() => JsonSerializer.Serialize(_product);

    [Benchmark]
    public string Newtonsoft() => JsonConvert.SerializeObject(_product);
}
```

Why not `Stopwatch`? 1. **JIT compilation** — first run is always slower (JIT warmup). BenchmarkDotNet handles warmup automatically. 2. **GC interference** — garbage collection can pause execution randomly. BenchmarkDotNet isolates GC effects. 3. **Statistical significance** — one measurement means nothing. BenchmarkDotNet runs hundreds of iterations and reports mean, median, and standard deviation. 4. **Memory allocation tracking** — `[MemoryDiagnoser]` shows Gen0/Gen1/Gen2 collections and bytes allocated.

Common mistakes: - Benchmarking in Debug mode (no optimizations) - Not including memory allocation metrics (faster code that allocates more can be worse) - Benchmarking too little or too much code (isolate the thing you're comparing) - Dead code elimination — the JIT might optimize away code that doesn't produce a used result

⚠ **Red Flag:** I use `Stopwatch.StartNew()` and `Console.WriteLine` the elapsed time." — That's a guess, not a benchmark.

***Follow-up:** Can you benchmark async code with BenchmarkDotNet? What's different?*

06

Architecture & System Design

10 questions 4 Mid 6 Senior

59 Your team debates whether to use the repository pattern with EF Core. You have 6 months of production experience. What's your answer?

SENIOR

Why interviewers ask this: One of the most debated topics in .NET. Tests opinionated thinking backed by experience.

GREAT ANSWER

My take: **you probably don't need a generic repository over EF Core.** Here's why:

`DbContext` already IS a repository + unit of work. Adding `IRepository<T>` on top is often a leaky abstraction:

```csharp
// What people build:
public interface IRepository<T>
{
    Task<T?> GetByIdAsync(int id);
    Task<IEnumerable<T>> GetAllAsync();
    Task AddAsync(T entity);
    // ... generic CRUD that mirrors DbSet<T>
}

// What they actually need:
await db.Products.FindAsync(id); // DbSet already does this
```

When I WOULD use a repository: 1. **Complex query logic** that I want to unit test without a database — wrap specific queries in a service/repository 2. **When EF Core might be swapped out** — unlikely but happens in some architectures 3. **Domain-Driven Design** — aggregate root repositories that enforce invariants

What I use instead: - Inject `DbContext` directly into handlers/services for simple CRUD - Create focused query services for complex queries:

```csharp
public class ProductQueries(AppDbContext db)
{
    public Task<ProductDto?> GetWithReviewsAsync(int id) => db.Products
        .Where(p => p.Id == id)
        .Select(p => new ProductDto { /* projection */ })
        .FirstOrDefaultAsync();
}
```

This gives me testability (mock `ProductQueries`) without the ceremony of generic repositories.

⚠ **Red Flag:** Either extreme — "always use repositories" or "never use repositories" without nuance.

***Follow-up:** In what architecture would you definitely use repositories? How does DDD change your answer?*

60 Explain CQRS. When is it overkill and when is it essential?

SENIOR

Why interviewers ask this: Architecture pattern that's widely discussed but often over-applied.

GREAT ANSWER

CQRS (Command Query Responsibility Segregation) means using different models for reading and writing data. At its simplest:

```csharp
// Write side — validates business rules, uses domain model
public class CreateOrderHandler(AppDbContext db)
{
    public async Task<int> Handle(CreateOrderCommand command) { ... }
}

// Read side — optimized for queries, uses DTOs/projections
public class GetOrderHandler(AppDbContext db)
{
    public async Task<OrderDto?> Handle(GetOrderQuery query) { ... }
}
```

When it's essential: - Read and write workloads have very different performance characteristics (100:1 read/write ratio) - The domain model is complex but the read model is simple (e-commerce, finance) - You need different data stores for reads vs writes (SQL for writes, Elasticsearch for reads) - Event sourcing — CQRS is almost required with event sourcing

When it's overkill: - Simple CRUD APIs with no complex business logic - Small teams where the overhead of separate models isn't worth it - When read and write models are nearly identical (you're just duplicating code)

My approach: Start without CQRS. When I notice that my read queries are fighting with my domain model (adding properties just for display, complex projections), that's the signal to split.

You don't need MediatR or a full CQRS framework. Separate handler classes are enough.

⚠ **Red Flag:** CQRS requires event sourcing and separate databases." — That's the advanced form. Simple CQRS is just separate read/write models.

***Follow-up:** How would you implement CQRS in a Vertical Slice Architecture project?*

61 What's Vertical Slice Architecture and how does it differ from Clean Architecture?

MID

Why interviewers ask this: Architecture comparison. Tests whether someone has thought about trade-offs.

GREAT ANSWER

Clean Architecture organizes by technical layer:

```
src/
  Domain/          (entities, value objects)
  Application/     (use cases, interfaces)
  Infrastructure/  (EF Core, external services)
  API/             (controllers, endpoints)
```

Every feature touches all four projects. A single "Create Product" feature changes files in 4 different directories.

Vertical Slice Architecture organizes by feature:

```
src/
  Features/
    Products/
      CreateProduct.cs     (endpoint + handler + validator + DTO, all in one file)
      GetProduct.cs
      UpdateProduct.cs
    Orders/
      CreateOrder.cs
      GetOrders.cs
```

Each file is self-contained — the endpoint, handler, validation, and DTO for one operation.

Key differences:

ASPECT	CLEAN ARCHITECTURE	VERTICAL SLICE
Coupling	Low between layers, high within	Low between features, high within
New feature	Touch 4+ projects	Touch 1 file
Shared code	Forced abstractions (repositories, services)	Share when natural, not by default
Onboarding	Understand the layers first	Open one feature file, understand it
Best for	Complex domains, large teams	API-focused, feature-driven work

My preference: Vertical Slice for Web APIs. Clean Architecture when the domain is genuinely complex (DDD, complex business rules). Most CRUD APIs don't need Clean Architecture.

⚠ **Red Flag:** Clean Architecture is always better because it separates concerns." — Separation by layer isn't always the right separation.

Follow-up: *Can you mix both? Use Vertical Slice for features but have a shared Domain layer?*

62 You're building a microservice that needs to communicate with 3 other services. How do you handle failures when one service is down?

SENIOR

Why interviewers ask this: Distributed systems resilience. Critical for production systems.

GREAT ANSWER

I'd use resilience patterns from `Microsoft.Extensions.Http.Resilience` (built on Polly v8):

```csharp
builder.Services.AddHttpClient("OrderService")
    .AddStandardResilienceHandler(); // Includes retry, circuit breaker, timeout
```

The standard resilience handler includes: 1. **Retry** — 3 attempts with exponential backoff for transient failures (5xx, timeouts) 2. **Circuit breaker** — after 10 failures in 30 seconds, stop calling the service for 30 seconds (fail fast instead of waiting) 3. **Timeout** — 30 second total timeout, 10 second per-attempt timeout

For critical paths, I'd also add:

Fallback — return cached/default data when the service is down:

```csharp
.AddResilienceHandler("fallback", builder =>
{
    builder.AddFallback(new FallbackStrategyOptions<HttpResponseMessage>
    {
        FallbackAction = _ => Outcome.FromResultAsValueTask(
            new HttpResponseMessage { Content = new StringContent("[]") })
    });
});
```

Design patterns: - **Graceful degradation** — if recommendations service is down, show products without recommendations. Don't fail the entire page. - **Bulkhead isolation** — separate HTTP client instances per service so one slow service doesn't exhaust all connections. - **Health checks** — monitor downstream service health and alert before users notice.

⚠ **Red Flag:** I'd just retry the request in a loop." — No backoff, no circuit breaking, you'll DDoS the failing service.

💬 ***Follow-up:*** *What's the difference between retry and hedging? When would you use hedging?*

63 How do you decide between a monolith and microservices for a new project?

MID

Why interviewers ask this: Architecture decision-making. Tests pragmatism vs hype.

GREAT ANSWER

Start with a monolith. Almost always. Here's my decision framework:

Monolith when: - Team is small (< 10 developers) - Domain isn't well understood yet (you'll get boundaries wrong) - Speed of development matters more than independent deployment - You don't have DevOps maturity (K8s, CI/CD, observability)

Microservices when: - Independent teams need to deploy independently - Parts of the system have very different scaling needs (search vs checkout) - Different parts use different tech stacks - You have the infrastructure maturity to operate them

The middle ground I actually recommend: Start with a modular monolith. Organize code by bounded contexts (modules) with clear interfaces between them. Each module has its own DbContext and models. If you need to extract a module into a microservice later, the boundaries are already clean.

```
src/
  Modules/
    Products/     (own DbContext, own endpoints, internal event bus)
    Orders/       (own DbContext, communicates via events)
    Inventory/    (own DbContext)
  SharedKernel/   (domain events, common types)
```

My experience: I've seen more projects fail from premature microservices than from staying monolithic too long. The operational complexity of microservices is massive — distributed transactions, eventual consistency, network failures, deployment orchestration.

⚠ **Red Flag:** Always microservices because they scale better." — Netflix needed microservices. Your CRUD API probably doesn't.

Follow-up: *How do you define bounded context boundaries in a modular monolith?*

64 You're designing an event-driven system where Order Service needs to notify Inventory Service when an order is placed. How do you ensure the event is reliably delivered?

SENIOR

Why interviewers ask this: Distributed messaging reliability. Tests real production experience.

GREAT ANSWER

The Outbox Pattern. Never publish an event directly — you'll have the dual-write problem where the database saves but the message broker publish fails (or vice versa).

The pattern: 1. Within the same database transaction, save the order AND the outbox event:

```csharp
await using var transaction = await db.Database.BeginTransactionAsync();

db.Orders.Add(order);
db.OutboxMessages.Add(new OutboxMessage
{
    Id = Guid.NewGuid(),
    Type = "OrderPlaced",
    Payload = JsonSerializer.Serialize(new OrderPlacedEvent(order.Id)),
    CreatedAt = DateTime.UtcNow,
    ProcessedAt = null
});

await db.SaveChangesAsync();
await transaction.CommitAsync();
```

1. 2. A background worker polls the outbox table and publishes events to the message broker (RabbitMQ, Azure Service Bus):

```csharp
var pending = await db.OutboxMessages
    .Where(m => m.ProcessedAt == null)
    .OrderBy(m => m.CreatedAt)
    .Take(100)
    .ToListAsync();

foreach (var message in pending)
{
    await messageBroker.PublishAsync(message.Type, message.Payload);
    message.ProcessedAt = DateTime.UtcNow;
}
await db.SaveChangesAsync();
```

Why not just publish directly? If the database save succeeds but the broker publish fails → you have an order but no event. If the broker publish succeeds but the database save fails → you have an event but no order. The outbox makes it atomic.

Libraries like **Wolverine** and **MassTransit** have built-in outbox support.

⚠ **Red Flag:** I'd publish the event after saving to the database." — Dual-write problem.

Follow-up: *How do you handle duplicate events? What does 'at-least-once delivery' mean and how does the consumer handle it?*

65 What's the difference between `BackgroundService` and a message queue for background processing?

MID

Why interviewers ask this: Background processing architecture. Common decision point.

GREAT ANSWER

ASPECT	BACKGROUNDSERVICE	MESSAGE QUEUE (RABBITMQ, SQS)
Where it runs	In your API process	Separate infrastructure
Survives app restart	No — loses in-memory state	Yes — messages persist
Scalability	Single instance processes	Multiple consumers, load balanced
Reliability	Work lost if process crashes	At-least-once delivery guaranteed
Complexity	Low — just a class	Medium — needs broker setup

Use `BackgroundService` for: - Non-critical work (sending emails, updating caches) - Work that's OK to retry from scratch if the app restarts - Simple periodic tasks (cleanup jobs, health monitoring)

CSHARP

```csharp
public class EmailBackgroundService(Channel<EmailRequest> channel) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken ct)
    {
        await foreach (var request in channel.Reader.ReadAllAsync(ct))
        {
            await SendEmailAsync(request);
        }
    }
}
```

Use a message queue for: - Critical work that MUST complete (payment processing, order fulfillment) - Work that needs to survive deployments/crashes - Work that needs to scale across multiple workers - Cross-service communication

My approach: Start with `BackgroundService` + `Channel<T>` for simple cases. Move to a message queue when reliability or scale demands it.

⚠ **Red Flag:** I use `Task.Run()` for background work." — Fire-and-forget with no error handling or lifecycle management.

***Follow-up:** How does Channels differ from ConcurrentQueue<T> for producer-consumer scenarios?*

66 How do you design an API that needs to support multi-tenancy — multiple customers sharing the same application but isolated data?

SENIOR

Why interviewers ask this: Multi-tenancy is complex. Tests architecture and security depth.

GREAT ANSWER

Three main strategies with different trade-offs:

1. Database-per-tenant: - Complete isolation — each tenant has their own database - Easiest for compliance and data migration - Most expensive — separate connection strings, separate backups - Best for: enterprise SaaS with strict data isolation requirements

2. Schema-per-tenant: - Same database, different schemas - Good isolation, shared infrastructure - Migration complexity — need to apply migrations to all schemas

3. Shared database with tenant ID (my default):

```csharp
public class Product
{
    public int Id { get; set; }
    public string TenantId { get; set; } = string.Empty; // Every table has this
    public string Name { get; set; } = string.Empty;
}
```

Global query filter ensures data isolation:

```csharp
modelBuilder.Entity<Product>().HasQueryFilter(p => p.TenantId == currentTenant.Id);
```

Tenant resolution via middleware:

```csharp
app.Use(async (context, next) =>
{
    var tenantId = context.Request.Headers["X-Tenant-Id"].FirstOrDefault()
                   ?? context.User.FindFirst("tenant_id")?.Value;
    context.RequestServices.GetRequiredService<ITenantContext>().TenantId = tenantId;
    await next();
});
```

The critical security rule: Never trust client-side tenant identification alone. Validate that the authenticated user belongs to the claimed tenant. Otherwise, a user can access another tenant's data by changing the header.

Red Flag: I'd add a WHERE clause to every query." — Error-prone. Someone will forget.

Follow-up: *How do you handle tenant-specific configuration or feature flags?*

67 What's the Mediator pattern and why do some .NET developers love it while others avoid it? MID

Why interviewers ask this: Controversial pattern. Tests critical thinking, not just pattern knowledge.

GREAT ANSWER

The Mediator pattern decouples the sender of a request from its handler. In .NET, MediatR is the popular library:

```csharp
// Instead of injecting the service directly:
app.MapGet("/products/{id}", async (int id, IProductService service) => ...);

// You send a request through the mediator:
app.MapGet("/products/{id}", async (int id, IMediator mediator) =>
    await mediator.Send(new GetProductQuery(id)));
```

Why people love it: - Cross-cutting concerns via pipeline behaviors (logging, validation, caching applied once) - Clean separation — endpoints are thin, handlers contain logic - Easy to add behaviors without modifying existing code

Why people avoid it: - **Indirection** — `Ctrl+Click` on `mediator.Send()` takes you to MediatR, not your handler. Navigation is harder. - **Service locator smell** — everything goes through one interface. You lose explicit dependencies. - **Overhead for simple apps** — for a CRUD API with 5 endpoints, it's ceremony without benefit.

My take: I don't use MediatR anymore. Wolverine handles messaging and mediation better, or I just use plain handler classes:

```csharp
public class GetProductHandler(AppDbContext db)
{
    public async Task<ProductDto?> HandleAsync(int id) => ...;
}
```

Same decoupling, no library dependency, and `Ctrl+Click` works.

⚠ **Red Flag:** MediatR is required for clean architecture." — It's not. It's a tool, not an architectural requirement.

***Follow-up:** How do pipeline behaviors in MediatR compare to endpoint filters in minimal APIs?*

68 You need to implement a feature flag system in your API. How would you design it?

SENIOR

Why interviewers ask this: Feature flags are essential for modern deployment practices. Tests design thinking.

GREAT ANSWER

For production, I'd use `Microsoft.FeatureManagement`:

CSHARP

```csharp
builder.Services.AddFeatureManagement();

app.MapGet("/api/products/recommendations", async (IFeatureManager features) =>
{
    if (await features.IsEnabledAsync("NewRecommendationEngine"))
        return await GetV2Recommendations();
    return await GetV1Recommendations();
});
```

Feature flag types I'd support:

1. 1. **Boolean flags** — on/off globally
2. 2. **Percentage rollout** — 10% of users get the new feature
3. 3. **User-targeted** — specific users or tenants get the feature
4. 4. **Time-based** — auto-enable on a specific date

Configuration in `appsettings.json`:

JSON

```json
{
  "FeatureManagement": {
    "NewRecommendationEngine": {
      "EnabledFor": [
        { "Name": "Percentage", "Parameters": { "Value": 25 } }
      ]
    }
  }
}
```

For dynamic flags (change without redeployment): - Use Azure App Configuration with `IOptionsSnapshot<T>` style refresh - Or store flags in a database with a short cache TTL - Or use a dedicated service (LaunchDarkly, Unleash)

Critical rule: Always clean up old feature flags. A codebase with 50 stale flags is worse than no flags at all. I add a comment with the flag's expiry date.

⚠ **Red Flag:** I use `#if DEBUG` for features." — Compile-time flags can't be changed in production.

***Follow-up:** How do you test both paths of a feature flag in your integration tests?*

07

Testing

10 questions 2 Junior 5 Mid 3 Senior

69 How do you write integration tests for an API that depends on a database and two external HTTP services?

SENIOR

Why interviewers ask this: Integration testing strategy. Shows production testing maturity.

GREAT ANSWER

I use `WebApplicationFactory` + Testcontainers + WireMock:

```csharp
public class ProductsApiTests : IClassFixture<ApiFixture>
{
    private readonly HttpClient _client;

    public ProductsApiTests(ApiFixture fixture)
    {
        _client = fixture.CreateClient();
    }

    [Fact]
    public async Task CreateProduct_ReturnsCreated()
    {
        // Arrange
        var request = new { Name = "Test", Price = 99.99 };

        // Act
        var response = await _client.PostAsJsonAsync("/api/products", request);

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.Created);
        var product = await response.Content.ReadFromJsonAsync<ProductResponse>();
        product!.Name.Should().Be("Test");
    }
}
```

The fixture:

```csharp
public class ApiFixture : WebApplicationFactory<Program>, IAsyncLifetime
{
    private readonly PostgreSqlContainer _postgres = new PostgreSqlBuilder().Build();
    private readonly WireMockServer _pricingApi = WireMockServer.Start();

    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            // Replace real DB with Testcontainers
            services.RemoveAll<DbContextOptions<AppDbContext>>();
            services.AddDbContext<AppDbContext>(o =>
                o.UseNpgsql(_postgres.GetConnectionString()));

            // Replace external service with WireMock
            services.Configure<PricingApiOptions>(o =>
                o.BaseUrl = _pricingApi.Url!);
        });
    }
    public async Task InitializeAsync()
    {
        await _postgres.StartAsync();
        _pricingApi.Given(Request.Create().WithPath("/api/prices"))
            .RespondWith(Response.Create().WithBody("[{\"sku\": \"A1\", \"price\": 1
0}]"));
    }
}
```

Why this approach: - Real database (Testcontainers) catches migration and query issues that SQLite/InMemory miss - WireMock for external services gives deterministic responses - `WebApplicationFactory` tests the full pipeline (middleware, filters, serialization) - Tests run in CI without external dependencies

> **Red Flag:** I mock the database with InMemory provider." — InMemory doesn't support transactions, constraints, or SQL-specific features.

> ***Follow-up:*** *How do you handle test data setup and cleanup between tests?*

70 What should you test with unit tests vs integration tests in an API project? MID

Why interviewers ask this: Test strategy. Tests whether someone over-tests or under-tests.

GREAT ANSWER

TEST TYPE	WHAT TO TEST	WHAT NOT TO TEST
Unit tests	Business logic, domain rules, validation, pure functions, calculators	Controller/endpoint wiring, DbContext queries, HTTP pipeline
Integration tests	Full request → response flow, database queries, auth pipeline, serialization	External services (use WireMock), UI

My distribution for a typical Web API: - **70% integration tests** — test from HTTP request to database and back - **20% unit tests** — domain logic, validators, complex calculations - **10% architecture tests** — verify conventions (all endpoints have auth, all DTOs are records, etc.)

What I DON'T unit test: - EF Core queries — they generate SQL. Unit testing LINQ that doesn't translate to SQL is testing the wrong thing. - Controllers/endpoints — the routing, binding, and serialization are framework code. Test them via integration tests. - DTOs and mapping — if mapping is trivial, an integration test covers it implicitly.

What I DO unit test:

```csharp
// Domain logic with rules
public class Order
{
    public void ApplyDiscount(decimal percentage)
    {
        if (percentage > 50) throw new BusinessRuleException("Max 50% discount");
        TotalAmount *= (1 - percentage / 100);
    }
}

// This is worth unit testing — pure business logic
[Fact]
public void ApplyDiscount_Over50Percent_ThrowsBusinessRule()
{
    var order = new Order { TotalAmount = 100 };
    var act = () => order.ApplyDiscount(60);
    act.Should().Throw<BusinessRuleException>();
}
```

⚠ **Red Flag:** I unit test everything, including database queries with the InMemory provider." — False confidence. InMemory behaves differently than real databases.

***Follow-up:** How do you decide if something is worth testing? What's your cost-benefit analysis?*

71 How do you test an endpoint that requires authentication in your integration tests?

MID

Why interviewers ask this: Practical testing challenge. Many devs skip auth in tests, which leaves security untested.

GREAT ANSWER

I create a test authentication handler that bypasses JWT validation but still sets up the ClaimsPrincipal:

```csharp
public class TestAuthHandler : AuthenticationHandler<AuthenticationSchemeOptions>
{
    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        var claims = new[]
        {
            new Claim(ClaimTypes.NameIdentifier, "test-user-id"),
            new Claim(ClaimTypes.Role, "Admin"),
        };
        var identity = new ClaimsIdentity(claims, "Test");
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, "Test");
        return Task.FromResult(AuthenticateResult.Success(ticket));
    }
}
```

Register it in the test fixture:

```csharp
builder.ConfigureServices(services =>
{
    services.AddAuthentication("Test")
        .AddScheme<AuthenticationSchemeOptions, TestAuthHandler>("Test", null);
});
```

For testing different roles/claims, I make the claims configurable:

```csharp
// Test as admin
_client.DefaultRequestHeaders.Add("X-Test-Role", "Admin");

// Test as regular user
_client.DefaultRequestHeaders.Add("X-Test-Role", "User");

// Test unauthorized
// Don't add the header — handler returns NoResult
```

Important: I also have tests that verify auth IS enforced:

```csharp
[Fact]
public async Task AdminEndpoint_WithoutAuth_Returns401()
{
    var client = _factory.CreateClient(); // No test auth
    var response = await client.GetAsync("/api/admin/users");
    response.StatusCode.Should().Be(HttpStatusCode.Unauthorized);
}
```

Red Flag: I remove `[Authorize]` in test environment." — Then you're not testing authorization at all.

Follow-up: *How would you test that a user can only access their own resources (resource-based authorization)?*

72 How do you test that your EF Core migrations actually work against a real database?

SENIOR

Why interviewers ask this: Migration testing prevents production deployment failures.

GREAT ANSWER

I use Testcontainers in a dedicated migration test:

CSHARP

```csharp
public class MigrationTests : IAsyncLifetime
{
    private readonly PostgreSqlContainer _postgres = new PostgreSqlBuilder().Build();

    [Fact]
    public async Task AllMigrations_ApplySuccessfully()
    {
        var options = new DbContextOptionsBuilder<AppDbContext>()
            .UseNpgsql(_postgres.GetConnectionString())
            .Options;

        await using var context = new AppDbContext(options);

        // Apply all migrations — this is what happens in production
        await context.Database.MigrateAsync();

        // Verify the schema is correct
        var canConnect = await context.Database.CanConnectAsync();
        canConnect.Should().BeTrue();

        // Optionally verify seed data
        var categories = await context.Categories.CountAsync();
        categories.Should().BeGreaterThan(0);
    }

    [Fact]
    public async Task Migrations_AreIdempotent()
    {
        // Apply migrations twice — should not throw
        await using var context = CreateContext();
        await context.Database.MigrateAsync();
        await context.Database.MigrateAsync(); // Second apply should be no-op
    }

    [Fact]
    public async Task PendingModelChanges_DoNotExist()
    {
        // Verify no model changes are missing a migration
        await using var context = CreateContext();
        await context.Database.MigrateAsync();
        var pending = context.Database.GetPendingMigrations();
        pending.Should().BeEmpty();
    }
}
```

This catches: - Migrations that fail due to SQL syntax errors - Migrations that work on SQLite but fail on PostgreSQL - Missing migrations (model changes without a corresponding migration) - Data seeding failures

> ⚠ **Red Flag:** I test migrations by running them in the staging environment." — Too late. Test them in CI.

> ***Follow-up:*** *How do you test a migration that involves data transformation — not just schema changes?*

73 What is snapshot testing (Verify) and when is it useful for API testing?

MID

Why interviewers ask this: Modern testing approach that reduces test maintenance.

GREAT ANSWER

Snapshot testing captures the output of a test and saves it to a file. On subsequent runs, it compares the output against the saved snapshot. If they differ, the test fails — and you review the diff.

Using the **Verify** library:

```csharp
[Fact]
public async Task GetProduct_ReturnsExpectedShape()
{
    var response = await _client.GetAsync("/api/products/1");
    var json = await response.Content.ReadAsStringAsync();

    await Verify(json); // First run: creates .verified.txt snapshot
                        // Next runs: compares against snapshot
}
```

The snapshot file (`GetProduct_ReturnsExpectedShape.verified.txt`):

```json
{
  "id": 1,
  "name": "Widget",
  "price": 29.99,
  "category": "Electronics",
  "createdAt": "DateTimeOffset_1" // Verify scrubs non-deterministic values
}
```

When it's useful: - API contract testing — catch unintended response shape changes - Serialization testing — verify JSON output matches expected format - Complex object comparison — instead of 20 assertions, one snapshot - OpenAPI document testing — snapshot the generated OpenAPI spec

When it's NOT useful: - Tests where the output changes frequently (timestamps, random IDs) - Simple assertions where explicit checks are clearer

Verify scrubs non-deterministic values automatically (DateTimes, Guids). You can also add custom scrubbers.

⚠ **Red Flag:** I've never heard of snapshot testing." — Fair for juniors, concerning for seniors.

Follow-up: *How do you handle snapshot updates when the change is intentional? What's the workflow?*

74 What is the AAA pattern in testing?

JUNIOR

Why interviewers ask this: Fundamental testing structure. Shows if someone writes organized tests.

GREAT ANSWER

Arrange, Act, Assert — three clear phases in every test:

CSHARP

```csharp
[Fact]
public async Task CreateProduct_WithValidData_ReturnsCreated()
{
    // Arrange — set up the test data and preconditions
    var request = new CreateProductRequest("Widget", 29.99m, "Electronics");

    // Act — perform the action being tested
    var response = await _client.PostAsJsonAsync("/api/products", request);

    // Assert — verify the result
    response.StatusCode.Should().Be(HttpStatusCode.Created);
    var product = await response.Content.ReadFromJsonAsync<ProductResponse>();
    product!.Name.Should().Be("Widget");
    product.Price.Should().Be(29.99m);
}
```

Why it matters: - **Readability** — anyone can understand the test's intent at a glance - **One action per test** — the Act section should be one or two lines. If it's more, you're testing too many things - **Clear failure diagnosis** — when a test fails, you know which phase went wrong

Common violations: - Asserting in the Arrange phase (checking preconditions that should be guaranteed) - Multiple Act phases in one test (testing two operations — split into two tests) - No clear separation (everything mashed together)

Red Flag: I just write code and check if it works at the end." — Unstructured tests are hard to maintain.

Follow-up: *When would you deviate from AAA? Are there cases where a different structure makes more sense?*

75 Your team has 500 integration tests and they take 15 minutes to run. How do you speed them up?

SENIOR

Why interviewers ask this: Test performance at scale. Shows practical experience with CI pipelines.

GREAT ANSWER

Multiple strategies, applied in order of impact:

1. Reuse the database container across tests:

```csharp
// DON'T: new container per test class
// DO: shared container with transaction rollback
public class ProductTests : IAsyncLifetime
{
    public async Task InitializeAsync()
    {
        await _db.Database.BeginTransactionAsync(); // Start transaction
    }
    public async Task DisposeAsync()
    {
        await _db.Database.RollbackTransactionAsync(); // Rollback — clean state
    }
}
```

2. Parallelize test collections:

```csharp
// xUnit v3 runs test classes in parallel by default
// But tests within a class run sequentially
// Group independent test classes into parallel collections
```

3. Use Respawn instead of recreating the database:

```csharp
private static readonly Respawner _respawner = Respawner.CreateAsync(connectionString).R
esult;

public async Task ResetDatabase()
{
    await _respawner.ResetAsync(connectionString); // Truncates tables, much faster than
recreating
}
```

4. Reduce unnecessary work: - Don't seed test data globally — each test creates only what it needs - Use `WebApplicationFactory` once per test collection, not per test - Disable logging in test configuration

5. Split into fast and slow suites: - Fast suite (< 2 min): core business logic, critical paths — runs on every commit - Full suite (15 min): everything — runs on PR merge

⚠ **Red Flag:** Switch to unit tests." — That changes what you're testing, not the speed.

Follow-up: *How do you handle test database state when tests run in parallel?*

76 How do you test an endpoint that calls an external HTTP API (like a payment gateway)?

MID

Why interviewers ask this: External dependency isolation. Core testing skill.

GREAT ANSWER

Never call the real payment gateway in tests. Use WireMock to create a fake HTTP server:

```csharp
private readonly WireMockServer _paymentApi = WireMockServer.Start();

[Fact]
public async Task ProcessPayment_Success_ReturnsOk()
{
    // Arrange — configure WireMock to return a successful response
    _paymentApi.Given(
        Request.Create()
            .WithPath("/api/charges")
            .WithBody(new JsonMatcher(new { amount = 100, currency = "USD" }))
            .UsingPost())
        .RespondWith(
            Response.Create()
                .WithStatusCode(200)
                .WithBody("""{"chargeId": "ch_123", "status": "succeeded"}"""));

    // Act
    var response = await _client.PostAsJsonAsync("/api/orders/1/pay",
        new { Amount = 100 });

    // Assert
    response.StatusCode.Should().Be(HttpStatusCode.Ok);
}

[Fact]
public async Task ProcessPayment_GatewayTimeout_ReturnsServiceUnavailable()
{
    // Test failure scenario
    _paymentApi.Given(Request.Create().WithPath("/api/charges").UsingPost())
        .RespondWith(Response.Create().WithDelay(TimeSpan.FromSeconds(30)));

    var response = await _client.PostAsJsonAsync("/api/orders/1/pay",
        new { Amount = 100 });

    response.StatusCode.Should().Be(HttpStatusCode.ServiceUnavailable);
}
```

Why WireMock over mocking IHttpClient: - Tests the full HTTP pipeline (serialization, headers, error handling) - You can test timeouts, slow responses, and network errors - The test reads like a real scenario, not implementation details

⚠ **Red Flag:** I mock the `HttpClient` with Moq." — You can't mock `HttpClient` directly (it's a concrete class), and mocking `HttpMessageHandler` tests implementation details.

> ***Follow-up:*** *How do you verify that your code sent the correct request to the external API? How do you check headers and body?*

77 What are architecture tests and why would you add them to your project? MID

Why interviewers ask this: Shows mature engineering practices beyond just feature testing.

GREAT ANSWER

Architecture tests verify that your code follows structural conventions automatically. They catch violations in CI instead of code review:

```csharp
using NetArchTest.Rules;

[Fact]
public void Endpoints_ShouldNot_DependOnInfrastructure()
{
    Types.InAssembly(typeof(Program).Assembly)
        .That().ResideInNamespace("Api.Endpoints")
        .ShouldNot().HaveDependencyOn("Infrastructure")
        .GetResult().IsSuccessful.Should().BeTrue();
}

[Fact]
public void AllEndpoints_ShouldRequireAuthorization()
{
    var endpoints = Types.InAssembly(typeof(Program).Assembly)
        .That().ImplementInterface(typeof(IEndpoint))
        .GetTypes();

    foreach (var endpoint in endpoints)
    {
        endpoint.GetMethods()
            .Should().Contain(m =>
                m.GetCustomAttributes<AuthorizeAttribute>().Any()
                || m.DeclaringType!.GetCustomAttributes<AuthorizeAttribute>().Any(),
                $"{endpoint.Name} must have [Authorize]");
    }
}

[Fact]
public void DomainLayer_ShouldNotReference_EntityFramework()
{
    Types.InAssembly(typeof(Order).Assembly)
        .ShouldNot().HaveDependencyOn("Microsoft.EntityFrameworkCore")
        .GetResult().IsSuccessful.Should().BeTrue();
}
```

What I test architecturally: - Layer dependency rules (domain doesn't reference infrastructure) - Naming conventions (handlers end with "Handler", DTOs end with "Request"/"Response") - Security (all endpoints have authorization attributes) - EF Core (all entities have configurations in `IEntityTypeConfiguration<T>`)

⚠ **Red Flag:** We enforce architecture in code reviews." — Manual review misses things. Automated tests don't.

Follow-up: *How do you handle exceptions to architecture rules? Not everything fits the convention.*

78 What testing library do you use in .NET and why? What assertion library do you prefer?

JUNIOR

Why interviewers ask this: Tests awareness of the .NET testing ecosystem.

GREAT ANSWER

xUnit v3 is my default: - Most popular in the .NET ecosystem - Constructor injection for test fixtures (DI-friendly) - Parallel test execution by default - Clean, convention-based (no `[TestClass]` attributes)

FluentAssertions for assertions:

```csharp
// Standard xUnit assertion — hard to read
Assert.Equal("Widget", product.Name);
Assert.True(product.Price > 0);

// FluentAssertions — reads like English
product.Name.Should().Be("Widget");
product.Price.Should().BePositive();
product.Categories.Should().Contain("Electronics");
response.StatusCode.Should().Be(HttpStatusCode.OK);
```

FluentAssertions gives much better error messages too. Instead of "Expected True, got False", you get "Expected product.Price to be positive, but found -5.00".

Other tools in my test stack: - **Testcontainers** — real databases in Docker for integration tests - **WireMock** — fake HTTP servers for external API testing - **Bogus** — generate realistic test data - **Verify** — snapshot testing - **Respawn** — fast database cleanup between tests

Red Flag: NUnit because that's what I learned." — Not wrong, but shows lack of ecosystem awareness. xUnit is the standard for new .NET projects.

***Follow-up:** What's new in xUnit v3 compared to v2?*

08

Production Readiness

10 questions 1 Junior 5 Mid 4 Senior

79 How do you implement structured logging in a .NET API? Why is it better than string-based logging?

SENIOR

Why interviewers ask this: Observability is critical in production. Structured logging is the foundation.

GREAT ANSWER

String-based logging:

```csharp
logger.LogInformation($"Order {orderId} created by user {userId} for ${amount}");
// Output: "Order 42 created by user john for $99.99"
// Can you search for all orders by user "john"? Only with regex. Painful.
```

Structured logging:

```csharp
logger.LogInformation("Order {OrderId} created by {UserId} for {Amount}",
    orderId, userId, amount);
// Stored as: { "OrderId": 42, "UserId": "john", "Amount": 99.99, "Message": "..." }
// Now you can query: WHERE UserId = 'john' AND Amount > 50
```

My setup with Serilog:

```csharp
builder.Host.UseSerilog((context, config) => config
    .ReadFrom.Configuration(context.Configuration)
    .Enrich.WithProperty("Application", "ProductApi")
    .Enrich.WithCorrelationId()
    .WriteTo.Console()
    .WriteTo.Seq("http://localhost:5341")); // Or Elasticsearch, Datadog, etc.

// Request logging middleware — one log per request instead of dozens
app.UseSerilogRequestLogging(options =>
{
    options.EnrichDiagnosticContext = (diagnosticContext, httpContext) =>
    {
        diagnosticContext.Set("UserId", httpContext.User.GetUserId());
        diagnosticContext.Set("RequestHost", httpContext.Request.Host.Value);
    };
});
```

Key practices: 1. **Use message templates, not string interpolation** — templates create searchable properties 2. **Enrich with context** — correlation ID, user ID, tenant ID on every log entry 3. **One request log line** — Serilog's `UseSerilogRequestLogging` replaces the noisy default logging 4. **Log levels matter** — `Information` for business events, `Warning` for recoverable issues, `Error` for failures 5. **Never log sensitive data** — mask PII, don't log request bodies with passwords

⚠ **Red Flag:** I use `Console.WriteLine` for debugging." — Not queryable, not structured, not production-ready.

> ***Follow-up:*** *How do you correlate logs across multiple services? What's a correlation ID and how do you propagate it?*

80 How do you implement health checks in a .NET API? What do you check? MID

Why interviewers ask this: Health checks are how load balancers and orchestrators know if your service is healthy.

GREAT ANSWER

```csharp
builder.Services.AddHealthChecks()
    .AddDbContextCheck<AppDbContext>("database")           // Can we reach the DB?
    .AddRedis(redisConnectionString, "redis")              // Can we reach Redis?
    .AddUrlGroup(new Uri("https://api.payment.com/health"),
        "payment-gateway");                                 // Is the payment API up?

app.MapHealthChecks("/health", new HealthCheckOptions
{
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});

// Separate liveness vs readiness
app.MapHealthChecks("/health/live", new HealthCheckOptions
{
    Predicate = _ => false // Just checks if the app is running
});

app.MapHealthChecks("/health/ready", new HealthCheckOptions
{
    Predicate = check => check.Tags.Contains("ready") // Checks dependencies
});
```

Liveness vs Readiness: - **Liveness** (`/health/live`) — "Is the process alive?" If it fails, Kubernetes restarts the pod. - **Readiness** (`/health/ready`) — "Can it handle requests?" If it fails, the load balancer stops sending traffic. The database might be down, but the app is still alive.

What I check: - Database connectivity (required) - Cache availability (Redis/memory) - External service health (payment gateway, email service) - Disk space (if writing logs/files locally) - Memory usage (custom health check for memory pressure)

What I DON'T check in health endpoints: - Business logic correctness - Expensive operations (don't run a full DB query, just check connectivity) - Authentication (health endpoints should be unauthenticated for load balancers)

Red Flag: I just check if the app returns 200 on any endpoint." — That tells you the app is running, not that it can serve traffic.

***Follow-up:** How do you configure Kubernetes liveness and readiness probes for a .NET API?*

81 How do you containerize a .NET API with Docker? Walk me through your Dockerfile.

MID

Why interviewers ask this: Docker knowledge is essential for modern deployment.

GREAT ANSWER

Multi-stage build — build in one stage, run in another:

```dockerfile
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
WORKDIR /src
COPY *.csproj .
RUN dotnet restore
COPY . .
RUN dotnet publish -c Release -o /app/publish

# Runtime stage
FROM mcr.microsoft.com/dotnet/aspnet:10.0-noble-chiseled AS runtime
WORKDIR /app
COPY --from=build /app/publish .
USER $APP_UID
EXPOSE 8080
ENTRYPOINT ["dotnet", "MyApi.dll"]
```

Key decisions:

1. 1. **Multi-stage build** — the SDK image is ~700MB. The runtime image is ~100MB. The chiseled image is ~30MB. Only ship what you need.
1. 2. **Chiseled images** (`-noble-chiseled`) — distroless Ubuntu. No shell, no package manager, no attack surface. Perfect for APIs.
1. 3. **Non-root user** (`USER $APP_UID`) — never run as root in production.
1. 4. **Layer caching** — `COPY *.csproj` and `dotnet restore` before copying source code. NuGet packages are cached unless the `.csproj` changes.
1. 5. **Port 8080** — .NET 8+ defaults to port 8080 in containers (not 80).

.dockerignore — exclude `bin/`, `obj/`, `.git/`, and test projects from the build context.

Alternative — no Dockerfile at all:

```bash
dotnet publish /t:PublishContainer -p ContainerRepository=myapp
```

.NET SDK can generate container images natively. Great for simple apps.

⚠ **Red Flag:** I copy the entire solution into the container and run `dotnet run`." — Development mode, not optimized, includes SDK.

Follow-up: *How do you handle health checks in Docker Compose? How does Docker know if your container is healthy?*

82 Your API is deployed to Kubernetes with 3 replicas. How do you handle graceful shutdown when a pod is terminated?

SENIOR

Why interviewers ask this: Production deployment. Tests understanding of container orchestration.

GREAT ANSWER

When Kubernetes terminates a pod, it sends `SIGTERM`. The app needs to: 1. Stop accepting new requests 2. Finish processing in-flight requests 3. Clean up resources (close DB connections, flush logs) 4. Exit

.NET handles this via `IHostApplicationLifetime`:

```csharp
app.Lifetime.ApplicationStopping.Register(() =>
{
    logger.LogInformation("Shutting down — finishing in-flight requests");
    // Signal health check to return unhealthy
    // Stop background services
});

app.Lifetime.ApplicationStopped.Register(() =>
{
    logger.LogInformation("Shutdown complete");
    Log.CloseAndFlush(); // Flush Serilog
});
```

Kubernetes configuration:

```yaml
spec:
  terminationGracePeriodSeconds: 30  # Time before SIGKILL
  containers:
    - name: api
      lifecycle:
        preStop:
          exec:
            command: ["sleep", "5"]  # Allow load balancer to remove the pod
```

The critical detail: There's a race condition between Kubernetes removing the pod from the service (load balancer) and sending SIGTERM. The `preStop` sleep ensures the load balancer stops sending traffic BEFORE the app starts shutting down.

Configure shutdown timeout in .NET:

```csharp
builder.Services.Configure<HostOptions>(options =>
    options.ShutdownTimeout = TimeSpan.FromSeconds(25)); // Must be less than terminatio
nGracePeriodSeconds
```

⚠ **Red Flag:** The app just stops when Kubernetes kills it." — In-flight requests get 502 errors.

> ***Follow-up:*** *How do long-running background services (like message consumers) handle graceful shutdown?*

83 How do you configure CI/CD for a .NET API using GitHub Actions?

MID

Why interviewers ask this: DevOps maturity. Most developers need CI/CD knowledge.

GREAT ANSWER

YAML

```yaml
name: CI/CD
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  build-test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_PASSWORD: test
        ports: ['5432:5432']
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '10.0.x'

      - name: Restore
        run: dotnet restore

      - name: Build
        run: dotnet build --no-restore -c Release

      - name: Test
        run: dotnet test --no-build -c Release --logger "trx"
        env:
          ConnectionStrings__Default: "Host=localhost;Database=test;Username=postgres;Pa
ssword=test"

      - name: Publish
        if: github.ref == 'refs/heads/main'
        run: dotnet publish src/Api -c Release -o ./publish

      - name: Build & Push Docker Image
        if: github.ref == 'refs/heads/main'
        run: |
          docker build -t myregistry/api:${{ github.sha }} .
          docker push myregistry/api:${{ github.sha }}
```

Key practices: 1. **Services block** for database — real PostgreSQL for integration tests 2. **Build once, test once, publish once** — don't rebuild between steps 3. **Only deploy from main** — PR branches get build + test, not deploy 4. **Tag with commit SHA** — every image is traceable to a commit 5. **Separate deploy step** — build and deploy are different jobs. Deploy requires manual approval for production.

> ⚠ **Red Flag:** We deploy by SSH-ing into the server and running `git pull`." — No CI, no tests, no rollback capability.

> ***Follow-up:*** *How do you handle database migrations in the CI/CD pipeline?*

84 How do you monitor your API in production? What metrics and alerts do you set up?

SENIOR

Why interviewers ask this: Observability. The difference between "it works on my machine" and "it works in production."

GREAT ANSWER

I use the three pillars of observability: **Logs, Metrics, Traces.**

Metrics I monitor (using OpenTelemetry):

METRIC	ALERT THRESHOLD	WHY
Request duration (p95)	> 500ms	Latency regression
Error rate (5xx)	> 1%	Something is broken
Request rate	Drop > 50% in 5min	Traffic anomaly or outage
Database query time (p95)	> 200ms	Slow queries
Memory usage	> 80% of limit	Memory leak
CPU usage	> 70% sustained	Scaling needed
Active DB connections	> 80% of pool	Connection leak

Setup with OpenTelemetry:

CSHARP

```csharp
builder.Services.AddOpenTelemetry()
    .WithMetrics(metrics =>
    {
        metrics.AddAspNetCoreInstrumentation()
               .AddHttpClientInstrumentation()
               .AddRuntimeInstrumentation()
               .AddOtlpExporter();
    })
    .WithTracing(tracing =>
    {
        tracing.AddAspNetCoreInstrumentation()
               .AddHttpClientInstrumentation()
               .AddEntityFrameworkCoreInstrumentation()
               .AddOtlpExporter();
    });
```

Custom business metrics:

CSHARP

```csharp
var orderCounter = meter.CreateCounter<int>("orders.created");
var orderDuration = meter.CreateHistogram<double>("orders.processing_duration_ms");

// In handler
orderCounter.Add(1, new KeyValuePair<string, object?>("payment_method", "stripe"));
```

Alerts I set up on day one: - Error rate spike → PagerDuty - P95 latency above SLA → Slack warning - Health check failure → PagerDuty - Memory trending upward (leak detection) → Slack

⚠ **Red Flag:** We check the logs when users report issues." — Reactive, not proactive. You should know about problems before users do.

Follow-up: *How do distributed traces help you debug a slow request that touches multiple services?*

85 What's the difference between `ILogger<T>` and `ILoggerFactory`? When do you use each?

MID

Why interviewers ask this: Logging fundamentals. Shows if someone understands the logging abstraction.

GREAT ANSWER

`ILogger<T>` is a typed logger scoped to a category (the class name):

```csharp
public class ProductService(ILogger<ProductService> logger)
{
    public void DoWork()
    {
        logger.LogInformation("Processing..."); // Category: "MyApp.ProductService"
    }
}
```

`ILoggerFactory` creates loggers dynamically:

```csharp
public class GenericProcessor(ILoggerFactory loggerFactory)
{
    public void Process(string jobType)
    {
        var logger = loggerFactory.CreateLogger($"Processor.{jobType}");
        logger.LogInformation("Processing {JobType}", jobType);
        // Category: "Processor.OrderSync" — dynamic based on job type
    }
}
```

When to use which: - `ILogger<T>` — 99% of the time. Standard approach for service classes. - `ILoggerFactory` — when you need dynamic categories (background job processors, plugin systems, generic classes where `T` isn't meaningful).

Don't inject `ILogger` (non-generic) — it uses an empty category. Your logs lose context about which class generated them.

⚠ **Red Flag:** I inject `ILogger` without the type parameter." — Loses category context.

***Follow-up:** How do log scopes work and when would you use `logger.BeginScope()`?*

86 How do you handle configuration for different environments (dev, staging, production)? MID

Why interviewers ask this: Configuration management. Fundamental for deployment.

GREAT ANSWER

.NET's configuration system layers sources with precedence:

```
appsettings.json (base)
  ↓ overrides
appsettings.{Environment}.json (environment-specific)
  ↓ overrides
Environment variables
  ↓ overrides
User secrets (dev only)
  ↓ overrides
Command-line args
```

My approach:

`appsettings.json` — defaults that work everywhere:

```json
{
  "Logging": { "LogLevel": { "Default": "Information" } },
  "ConnectionStrings": { "Default": "" },
  "Cache": { "ExpirationMinutes": 30 }
}
```

`appsettings.Development.json` — local dev settings (committed):

```json
{
  "Logging": { "LogLevel": { "Default": "Debug" } },
  "ConnectionStrings": { "Default": "Host=localhost;Database=myapp_dev" }
}
```

Production secrets — NEVER in JSON files. Use environment variables or a secret manager:

```bash
# Docker / Kubernetes
ConnectionStrings__Default=Host=prod-db;Database=myapp;Password=***
```

Key practices: 1. `ASPNETCORE_ENVIRONMENT` controls which `appsettings.{env}.json` loads 2. Use Options pattern with validation:

```csharp
builder.Services.AddOptions<DatabaseOptions>()
    .BindConfiguration("Database")
    .ValidateDataAnnotations()
    .ValidateOnStart(); // Fail fast if config is missing
```

1. 3. Never use `if (env.IsDevelopment())` for business logic — only for dev tooling (Swagger, detailed errors).

⚠ **Red Flag:** I have different `appsettings.json` files and deploy the right one." — That's manual and error-prone.

💬 ***Follow-up:*** *How does `ValidateOnStart()` work? What happens if required configuration is missing?*

87 What is `Program.cs` in a .NET API and what does it configure?

JUNIOR

Why interviewers ask this: Entry point understanding. Tests if someone knows how an API boots up.

GREAT ANSWER

`Program.cs` is the application's entry point. It configures everything the API needs:

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

// 1. SERVICE REGISTRATION — configure DI container
builder.Services.AddDbContext<AppDbContext>(o => o.UseNpgsql(connectionString));
builder.Services.AddScoped<IProductService, ProductService>();
builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
builder.Services.AddOpenApi();

var app = builder.Build();

// 2. MIDDLEWARE PIPELINE — configure request processing order
app.UseExceptionHandler();
app.UseCors();
app.UseAuthentication();
app.UseAuthorization();

// 3. ENDPOINT MAPPING — define routes
app.MapGet("/api/products", (IProductService service) => service.GetAllAsync());
app.MapPost("/api/products", (CreateProductRequest req) => ...);

app.Run();
```

Two phases: 1. `builder` phase — register services, configure logging, add authentication schemes. Nothing is running yet. 2. `app` phase — configure middleware pipeline and map endpoints. The order of middleware registration matters.

Since .NET 6: No more `Startup.cs`. Everything is in `Program.cs` with top-level statements. The `partial class Program` is implicit — useful for `WebApplicationFactory` in tests:

```csharp
// In test project
public class ApiFixture : WebApplicationFactory<Program> { }
```

Red Flag: It's where the `Main` method is." — Technically true but shows no understanding of what it configures.

***Follow-up:** How would you organize `Program.cs` as it grows beyond 100 lines? Extension methods?*

88 You deploy a new version and error rates spike. How do you handle rollback?

SENIOR

Why interviewers ask this: Production incident management. Tests operational maturity.

GREAT ANSWER

Immediate response (first 5 minutes):

1. 1. **Confirm the spike correlates with the deployment** — check deployment timestamp vs error start time
2. 2. **Assess blast radius** — is it all endpoints or specific ones? All users or a subset?
3. 3. **Rollback decision:** If error rate > 5% or affecting payments/auth, rollback immediately. Don't debug in production.

Rollback strategies (by deployment type):

Container/Kubernetes:

```bash
kubectl rollout undo deployment/api # Instant rollback to previous version
```

Azure App Service:

```bash
az webapp deployment slot swap --slot staging --target-slot production # Swap back
```

Database migrations — the tricky part: - If the new version added columns but didn't remove any → rollback is safe, old code ignores new columns - If the migration dropped or renamed columns → you need a forward-fix migration, not a rollback - **This is why I always make migrations backward-compatible:** add new columns first, deploy new code, then remove old columns in a later release

Prevention: - **Canary deployments** — route 5% of traffic to the new version first - **Feature flags** — deploy the code but keep new features disabled - **Blue-green deployments** — run both versions, switch traffic atomically

Post-incident: Write a blameless post-mortem. What happened, why the tests didn't catch it, how to prevent it.

⚠ **Red Flag:** We fix forward — we never rollback." — Sometimes rollback is the fastest way to stop the bleeding.

***Follow-up:** How do you make database migrations backward-compatible for zero-downtime deployments?*

09

Modern .NET & C# Features

12 questions 1 Junior 7 Mid 4 Senior

89 What are primary constructors in C# 12+ and how do they change how you write services?

MID

Why interviewers ask this: Modern C# awareness. Shows if someone stays current.

GREAT ANSWER

Primary constructors let you declare constructor parameters directly on the class:

```csharp
// Before — verbose
public class ProductService
{
    private readonly AppDbContext _db;
    private readonly ILogger<ProductService> _logger;

    public ProductService(AppDbContext db, ILogger<ProductService> logger)
    {
        _db = db;
        _logger = logger;
    }
}

// After — primary constructor
public class ProductService(AppDbContext db, ILogger<ProductService> logger)
{
    public async Task<Product?> GetAsync(int id)
    {
        logger.LogInformation("Getting product {Id}", id);
        return await db.Products.FindAsync(id);
    }
}
```

Key behaviors: - Parameters are captured as fields (compiler-generated) - They're mutable (unlike `readonly` fields) — but in practice, you don't reassign DI dependencies - Available throughout the class, not just the constructor - Works with `class`, `struct`, and `record`

When NOT to use: - When you need validation or logic in the constructor - When you need `readonly` enforcement (the parameter is mutable) - When multiple constructors with different signatures are needed

My take: I use primary constructors for all DI-injected services. Less boilerplate, same behavior. The mutability concern is theoretical — I've never seen someone accidentally reassign a `DbContext` parameter.

Red Flag: I haven't upgraded to C# 12 yet." — C# 12 shipped with .NET 8. It's been available for 2+ years.

***Follow-up:** How do primary constructors interact with inheritance? Can a derived class use the base class's primary constructor parameters?*

90 What are records in C# and when would you use them instead of classes for your DTOs?

MID

Why interviewers ask this: Modern C# data modeling. DTOs are used in every API.

GREAT ANSWER

Records provide value-based equality, immutability by default, and concise syntax:

```csharp
// Record — immutable, value equality, built-in ToString
public record ProductResponse(int Id, string Name, decimal Price);

// Equivalent class would need:
// - Constructor with all properties
// - Override Equals and GetHashCode
// - Override ToString
// - Implement IEquatable<T>
// That's ~30 lines vs 1 line.
```

When I use records: - **DTOs** (request/response models) — immutable, value equality for testing:

```csharp
var expected = new ProductResponse(1, "Widget", 29.99m);
var actual = await GetProductAsync(1);
actual.Should().Be(expected); // Value equality works!
```

- - **Value objects** in domain models
- - **Configuration options** (readonly settings)

When I use classes: - **Entities** — EF Core works better with mutable classes (change tracking) - **Services** — they have behavior, not just data - **When mutation is needed** — records require `with` expressions for changes:

```csharp
var updated = product with { Price = 39.99m }; // Creates a new instance
```

`record class` **vs** `record struct`**:** - `record class` (default) — reference type, heap allocated - `record struct` — value type, stack allocated, good for small DTOs in hot paths

Red Flag: Records are the same as classes but shorter." — Misses value equality, the key differentiator.

Follow-up: *How do records interact with JSON serialization? Any gotchas with `System.Text.Json`?*

91 What are collection expressions in C# 12 and how do they simplify your code?

JUNIOR

Why interviewers ask this: Modern C# syntax awareness.

GREAT ANSWER

Collection expressions provide a uniform syntax for creating collections:

```csharp
// Before — different syntax for each type
var list = new List<int> { 1, 2, 3 };
var array = new int[] { 1, 2, 3 };
int[] array2 = new[] { 1, 2, 3 };
var span = stackalloc int[] { 1, 2, 3 };

// After — one syntax for all
List<int> list = [1, 2, 3];
int[] array = [1, 2, 3];
Span<int> span = [1, 2, 3];
ImmutableArray<int> immutable = [1, 2, 3];
```

The spread operator (`..`) is the game-changer:

```csharp
int[] first = [1, 2, 3];
int[] second = [4, 5, 6];
int[] combined = [..first, ..second, 7, 8]; // [1, 2, 3, 4, 5, 6, 7, 8]
```

Empty collections without allocation:

```csharp
// Before
return Array.Empty<Product>();
return Enumerable.Empty<Product>();

// After
return [];  // Compiler picks the most efficient representation
```

Practical use in APIs:

```csharp
// Combining validation errors
List<string> errors = [..nameErrors, ..priceErrors, ..categoryErrors];

// Initializing seed data
List<Category> categories = [
    new() { Name = "Electronics" },
    new() { Name = "Clothing" },
];
```

⚠ **Red Flag:** I still use new `List<int> { ... }`." — Not wrong, but shows unfamiliarity with modern C#.

***Follow-up:** Can collection expressions create custom collection types? What's the `CollectionBuilder` attribute?*

92 What is `Span<T>` and `Memory<T>`? When would you use them in an API? SENIOR

Why interviewers ask this: Performance optimization. Senior-level memory management.

GREAT ANSWER

`Span<T>` is a stack-allocated view over contiguous memory (arrays, strings, native memory). It avoids allocations by slicing existing memory instead of copying:

```csharp
// Allocating — creates new string objects
string input = "2026-03-15";
string year = input.Substring(0, 4);   // Allocates new string "2026"
string month = input.Substring(5, 2);  // Allocates new string "03"

// Non-allocating — views into the same memory
ReadOnlySpan<char> span = input.AsSpan();
ReadOnlySpan<char> year = span[..4];   // No allocation — points to same memory
ReadOnlySpan<char> month = span[5..7]; // No allocation
```

When I use them in APIs:

1. 1. **Parsing request data without allocations:**

```csharp
// Parse a comma-separated query parameter
ReadOnlySpan<char> ids = request.Query["ids"].AsSpan();
foreach (var range in ids.Split(','))
{
    var id = int.Parse(ids[range]); // No string allocations
}
```

1. 2. **Processing file uploads** — slice byte arrays without copying

1. 3. **High-throughput endpoints** — when benchmarks show string allocation pressure

`Span<T>` **vs** `Memory<T>`**:** - `Span<T>` — stack-only, can't be stored in fields, can't be used in async methods - `Memory<T>` — heap-safe, can cross async boundaries

When NOT to use: - Regular CRUD endpoints — the performance gain is negligible for typical API workloads - When readability suffers more than the performance helps

My rule: Use `Span<T>` when benchmarks show that string/array allocations are a bottleneck. Don't prematurely optimize.

⚠ **Red Flag:** I use Span everywhere for performance." — Premature optimization. Most API bottlenecks are I/O, not allocations.

***Follow-up:** Can you use Span<T> in async methods? Why or why not?*

93 What is pattern matching in C# and how do you use it in API code?

MID

Why interviewers ask this: Modern C# feature used for clean control flow.

GREAT ANSWER

Pattern matching lets you test values against patterns and extract data:

```csharp
// Type pattern — replace if/else with switch expression
public IResult HandleResult<T>(Result<T> result) => result switch
{
    { IsSuccess: true, Value: var data } => Results.Ok(data),
    { Error: NotFoundError } => Results.NotFound(),
    { Error: ValidationError err } => Results.BadRequest(err.Errors),
    { Error: var err } => Results.Problem(err.Message),
};

// List pattern — validate array contents
static bool IsValidCoordinate(double[] coords) => coords is [>= -180, >= -90];

// Property pattern — readable conditions
if (request is { PageSize: > 100 })
    return Results.BadRequest("Max page size is 100");

// Relational pattern
string GetPriceCategory(decimal price) => price switch
{
    < 10 => "Budget",
    >= 10 and < 50 => "Mid-range",
    >= 50 and < 200 => "Premium",
    _ => "Luxury"
};
```

Most practical API uses:

1. 1. **Result/error mapping** (shown above) — clean error handling without if/else chains
2. 2. **Request validation** — readable guard clauses
3. 3. **Status mapping** — convert domain states to HTTP responses
4. 4. **Feature branching** — match on configuration or feature flags

⚠ **Red Flag:** I use `is` for null checks only." — Missing the power of pattern matching.

Follow-up: *What's the `switch` expression vs the `switch` statement? When would you use each?*

94 What's the difference between `record`, `class`, and `struct` in C#? When would you use each in an API project?

MID

Why interviewers ask this: Fundamental type system understanding.

GREAT ANSWER

FEATURE	CLASS	RECORD CLASS	STRUCT	RECORD STRUCT
Allocation	Heap	Heap	Stack	Stack
Equality	Reference	Value	Value	Value
Mutable	Yes	Immutable (default)	Yes	Immutable (default)
Inheritance	Yes	Yes	No	No
Nullable	Reference type	Reference type	Value type	Value type

In my API projects:

- - `class` — EF Core entities (need mutability for change tracking), services, handlers

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

- - `record` — DTOs, events, value objects

```csharp
public record CreateProductRequest(string Name, decimal Price);
public record ProductCreatedEvent(int ProductId, DateTime CreatedAt);
```

- - `struct` — small value types in hot paths (coordinates, money, strongly-typed IDs)

```csharp
public readonly record struct ProductId(int Value);
public readonly record struct Money(decimal Amount, string Currency);
```

When `struct` wins: When you're creating millions of small objects and want to avoid GC pressure. For typical API work, `class` and `record` are fine — don't micro-optimize.

⚠ **Red Flag:** I use classes for everything." — Missing records for DTOs is a lot of unnecessary boilerplate.

Follow-up: *What does `readonly record struct` give you over `record struct`?*

95 How does `IAsyncEnumerable<T>` work and when would you use it in an API? MID

Why interviewers ask this: Streaming data handling. Modern async pattern.

GREAT ANSWER

`IAsyncEnumerable<T>` returns items one at a time as they become available, instead of loading everything into memory:

```csharp
// Loads ALL products into memory first, then returns
app.MapGet("/products", async (AppDbContext db) =>
    await db.Products.ToListAsync()); // Materializes entire collection

// Streams products one at a time — constant memory usage
app.MapGet("/products/stream", (AppDbContext db) =>
    db.Products.AsAsyncEnumerable()); // Yields as DB cursor reads
```

When to use:

1. 1. **Large dataset exports** (CSV, JSON streaming):

```csharp
app.MapGet("/api/export", async (AppDbContext db, HttpContext http) =>
{
    http.Response.ContentType = "application/json";
    await foreach (var product in db.Products.AsNoTracking().AsAsyncEnumerable())
    {
        await JsonSerializer.SerializeAsync(http.Response.Body, product);
    }
});
```

1. 2. **Server-Sent Events (SSE)** — streaming updates to clients

1. 3. **Processing large files** — read and process line by line

When NOT to use: - Small datasets (< 1000 records) — the overhead of streaming isn't worth it - When you need the total count (streaming doesn't know the count upfront) - When you need to sort or filter the complete dataset

ASP.NET Core support: Returning `IAsyncEnumerable<T>` from a minimal API endpoint automatically streams the response as a JSON array. The first `[` is written immediately, and items are flushed as they arrive.

⚠ **Red Flag:** I always use `ToListAsync()` because it's simpler." — Fine for small datasets, problematic for large ones.

***Follow-up:** How does `IAsyncEnumerable` interact with EF Core's change tracker?*

96 What is source generation in .NET and how do you use it for better API performance?

SENIOR

Why interviewers ask this: Advanced .NET optimization. Shows deep framework knowledge.

GREAT ANSWER

Source generators produce C# code at compile time, eliminating runtime reflection. Three key uses in APIs:

1. System.Text.Json source generation — faster serialization:

```csharp
[JsonSerializable(typeof(ProductResponse))]
[JsonSerializable(typeof(List<ProductResponse>))]
internal partial class AppJsonContext : JsonSerializerContext;

// Use in minimal APIs
builder.Services.ConfigureHttpJsonOptions(options =>
    options.SerializerOptions.TypeInfoResolverChain.Insert(0, AppJsonContext.Default));
```

Result: 2-3x faster serialization, no reflection, AOT-compatible.

2. Logging source generation — structured logging without boxing:

```csharp
public static partial class LogMessages
{
    [LoggerMessage(Level = LogLevel.Information,
        Message = "Processing order {OrderId} for {Amount}")]
    public static partial void OrderProcessing(this ILogger logger, int orderId, decimal
amount);
}

// Usage — no boxing, no string interpolation at runtime
logger.OrderProcessing(42, 99.99m);
```

3. Regex source generation — compile-time regex:

```csharp
[GeneratedRegex(@"^[a-zA-Z0-9+_.-]+@[a-zA-Z0-9.-]+$")]
private static partial Regex EmailRegex();
```

Why it matters: - **AOT (Ahead of Time) compilation** — source generators make your code AOT-compatible by removing reflection - **Startup performance** — no reflection-based initialization - **Trimming** — the linker can remove unused code because dependencies are explicit

Red Flag: Source generators are only for library authors." — They're increasingly important for application developers, especially with AOT.

Follow-up: *What's the difference between source generators and Roslyn analyzers?*

97 What are minimal APIs and why would you choose them over controllers for a new project?

MID

Why interviewers ask this: Modern .NET API development. The default since .NET 6.

GREAT ANSWER

Minimal APIs define endpoints directly in `Program.cs` (or extension methods) without the ceremony of controller classes:

```csharp
// Minimal API
app.MapGet("/api/products/{id}", async (int id, AppDbContext db) =>
    await db.Products.FindAsync(id) is Product p
        ? Results.Ok(p)
        : Results.NotFound());

// Equivalent Controller
[ApiController]
[Route("api/products")]
public class ProductsController(AppDbContext db) : ControllerBase
{
    [HttpGet("{id}")]
    public async Task<IActionResult> Get(int id)
    {
        var product = await db.Products.FindAsync(id);
        return product is null ? NotFound() : Ok(product);
    }
}
```

Why I choose minimal APIs for new projects:

ASPECT	MINIMAL APIS	CONTROLLERS
Boilerplate	Low — just the handler	High — class, attributes, inheritance
Performance	Slightly faster (no reflection)	Reflection-based routing
AOT support	Full	Limited
Parameter binding	Explicit	Convention-based (implicit)
Testability	Inject handler directly	Need `WebApplicationFactory`
OpenAPI	`TypedResults` gives compile-time metadata	Attributes required

When I'd still use controllers: - Large teams familiar with MVC patterns - When you need model binding features that minimal APIs lack - Migration of existing controller-based projects (don't rewrite just to rewrite)

Red Flag: Controllers are always better because they organize code." — Organization comes from file structure, not the controller pattern.

***Follow-up:** How do you organize minimal APIs as the project grows? What's the `MapGroup` approach?*

98 What's new in .NET 10 that impacts how you build Web APIs?

SENIOR

Why interviewers ask this: Staying current. Shows continuous learning.

GREAT ANSWER

Key .NET 10 features for API developers:

1. HybridCache improvements — built-in stampede protection, L1+L2 caching (memory + distributed), simpler API than `IDistributedCache`.

2. OpenAPI built-in — `AddOpenApi()` and `MapOpenApi()` are built into the framework. No more Swashbuckle (which is unmaintained). Transformers for customization:

```csharp
builder.Services.AddOpenApi(options =>
    options.AddDocumentTransformer<SecuritySchemeTransformer>());
```

3. `ExecuteUpdateAsync` / `ExecuteDeleteAsync` improvements — more complex bulk operations supported.

4. Built-in rate limiting — sliding window, token bucket, fixed window, concurrency. No third-party library needed.

5. Improved minimal API parameter binding — better support for complex types from query strings.

6. EF Core 10 — always matches .NET major version. Improved LINQ translation, better JSON column support, compiled models improvements.

7. Container publishing — `dotnet publish /t:PublishContainer` without a Dockerfile.

What I've adopted: - HybridCache replacing our custom caching layer - Built-in OpenAPI replacing Swashbuckle - Collection expressions and primary constructors everywhere - `TimeProvider` for testable time-dependent code

⚠ **Red Flag:** I'm still on .NET 6 LTS." — .NET 6 is end of life. Staying current matters for security and features.

***Follow-up:** How do you plan a .NET version upgrade for a production application? What's your migration checklist?*

99 What is the `TimeProvider` abstraction and why should you use it? MID

Why interviewers ask this: Testability of time-dependent code. A practical modern .NET feature.

GREAT ANSWER

`TimeProvider` abstracts `DateTime.UtcNow` and `DateTimeOffset.UtcNow` so you can control time in tests:

```csharp
// Production code — inject TimeProvider
public class DiscountService(TimeProvider time)
{
    public bool IsBlackFriday()
    {
        var now = time.GetUtcNow();
        return now.Month == 11 && now.Day >= 25 && now.Day <= 30;
    }
}

// Registration
builder.Services.AddSingleton(TimeProvider.System); // Real time in production
```

Testing with `FakeTimeProvider`:

```csharp
[Fact]
public void IsBlackFriday_OnNovember25_ReturnsTrue()
{
    var fakeTime = new FakeTimeProvider(new DateTimeOffset(2026, 11, 25, 0, 0, 0, TimeSp
an.Zero));
    var service = new DiscountService(fakeTime);

    service.IsBlackFriday().Should().BeTrue();
}

[Fact]
public void IsBlackFriday_OnDecember1_ReturnsFalse()
{
    var fakeTime = new FakeTimeProvider(new DateTimeOffset(2026, 12, 1, 0, 0, 0, TimeSpa
n.Zero));
    var service = new DiscountService(fakeTime);

    service.IsBlackFriday().Should().BeFalse();
}
```

Why this matters: Before `TimeProvider`, people used `DateTime.UtcNow` directly, making tests non-deterministic (tests pass on Black Friday, fail on other days), or they built custom `IClock` interfaces.

`TimeProvider` also provides `CreateTimer()` for testable timers and `GetTimestamp()` for testable performance measurement.

Red Flag: I use `DateTime.Now` everywhere." — Not testable, and `.Now` uses local time which causes timezone bugs.

Follow-up: *What's the difference between `DateTime.UtcNow` and `DateTimeOffset.UtcNow`? Which should you use in APIs?*

100 What's the `field` keyword in C# 14 and how does it change property definitions?

SENIOR

Why interviewers ask this: Cutting-edge C# feature. Shows if someone follows language evolution.

GREAT ANSWER

The `field` keyword gives you access to the auto-generated backing field inside a property, without manually declaring it:

```csharp
// Before — had to declare a backing field manually for validation
private string _name = string.Empty;
public string Name
{
    get => _name;
    set => _name = value ?? throw new ArgumentNullException(nameof(value));
}

// After (C# 14) — field keyword references the compiler-generated backing field
public string Name
{
    get;
    set => field = value ?? throw new ArgumentNullException(nameof(value));
}
```

Practical uses in API development:

1. 1. **Lazy initialization:**

```csharp
public string FullName
{
    get => field ??= $"{FirstName} {LastName}";
}
```

1. 2. **Validation in properties:**

```csharp
public decimal Price
{
    get;
    set => field = value >= 0 ? value : throw new ArgumentException("Price must be posit
ive");
}
```

1. 3. **Change tracking / notification:**

```csharp
public string Status
{
    get;
    set
    {
        if (field != value)
        {
                field = value;
                OnStatusChanged();
            }
        }
    }
```

Why it matters: Eliminates the boilerplate of declaring backing fields for properties that need custom logic in the getter or setter, while still using auto-property syntax.

Red Flag: I haven't heard of the `field` keyword." — Fair, it's brand new in C# 14 / .NET 10. But senior devs should follow language previews.

Follow-up: *How does `field` interact with `required` and `init` property accessors?*

Want More .NET Content?

Join 6,500+ .NET developers who get weekly tips, benchmarks, and production-tested patterns every Tuesday.

codewithmukesh.com/newsletter

Free Resources

.NET Web API Zero to Hero Course
codewithmukesh.com/courses/dotnet-webapi-zero-to-hero

Claude Code for .NET Developers
codewithmukesh.com/courses/claude-code-for-dotnet-developers

140+ .NET Articles & Guides
codewithmukesh.com/blog