ChatGPT

گردآوری‌شده توسط بامداد طبری به کمک ChatGPT

بهترین شیوه‌ها در توسعه APIهای .NET 10

فرض کنید در یک پروژه فرضی قصد داریم یک **API وب** برای یک فروشگاه آنلاین با استفاده از .NET 10 ایجاد کنیم. برای توسعه‌دهندگان باتجربه، رعایت مجموعه‌ای از **بهترین شیوه‌ها (Best Practices)** می‌تواند تضمین کند که API ما **قابل‌اتکا، امن، کارا و قابل‌نگهداری** باشد. در ادامه با فرض این پروژه فروشگاه آنلاین، مهم‌ترین این شیوه‌ها را در قالب سناریوهای عملی بررسی می‌کنیم. این شیوه‌ها شامل مواردی همچون محدودسازی نرخ درخواست‌ها، تنظیم صحیح CORS، فشرده‌سازی پاسخ‌ها، نسخه‌بندی API، مدیریت سراسری خطاها، کش کردن خروجی، برنامه‌نویسی asynchronous، استفاده از تزریق وابستگی، مستندسازی خودکار API، لاگینگ مناسب، پیاده‌سازی احراز هویت JWT، اعتبارسنجی ورودی‌ها، نام‌گذاری درست Endpoints و چند توصیه دیگر است. بیایید هر کدام را با جزئیات مرور کنیم.

محدودسازی نرخ درخواست‌ها (Rate Limiting)

یکی از بهترین شیوه‌ها برای **محافظت از API**، محدود کردن نرخ یا تعداد درخواست‌های ورودی در یک بازه زمانی مشخص است. **Rate Limiting** می‌تواند جلوی سوءاستفاده کاربران یا ربات‌ها را بگیرد و از فشار بیش از حد به سرور جلوگیری کند [1]. در پروژه فرضی فروشگاه آنلاین، ممکن است بخواهیم هر **کاربر** حداکثر ۱۰۰ درخواست در دقیقه به API ارسال کند تا همهٔ کاربران به‌طور منصفانه از منابع بهره‌مند شوند و از حملات احتمالی (مثل DoS) جلوگیری شود [1]. مهم‌ترین دلایل پیاده‌سازی Rate Limiting عبارت‌اند از:

- **جلوگیری از سوءاستفاده و حملات:** محدودسازی نرخ می‌تواند از بمباران سرور توسط یک کاربر یا مهاجم جلوگیری کرده و ریسک حملات انکار سرویس (DoS) را کاهش دهد [1].
- **تضمین استفاده منصفانه:** با تعیین سقف درخواست، همه کاربران به شکل عادلانه به API دسترسی دارند و یک کاربر پرمصرف نمی‌تواند منابع سیستم را در انحصار گیرد [2].
- **حفظ پایداری و کارایی:** کنترل نرخ ورودی کمک می‌کند سرور از پس پردازش درخواست‌ها برآید و دچار اضافه‌بار نشود که این به **تجربه بهتر کاربری** و جلوگیری از افزایش هزینه‌ها (مثلاً در منابع ابری) منجر می‌شود [3].

در **ASP.NET Core** (نسخه‌های 7 به بعد)، یک میان‌افزار داخلی برای Rate Limiting فراهم شده است. این قابلیت از طریق فضای نام `Microsoft.AspNetCore.RateLimiting` در دسترس است و با تعریف policyهای مختلف می‌توان نرخ را در سطح‌های گوناگون محدود کرد [4]. برای مثال، می‌توان یک سیاست سراسری تنظیم کرد که به هر **کاربر** تأییدشده (Identity) حداکثر ۱۰۰ درخواست در دقیقه اجازه می‌دهد. پیاده‌سازی این سیاست به این صورت است که در زمان راه‌اندازی برنامه (مثلاً در `Program.cs`) سرویس Rate Limiter را اضافه کرده و policy مربوطه را تعریف می‌کنیم و سپس Middleware آن را فعال می‌کنیم [5] [6]:

```
// تنظیم محدودسازی نرخ: ۱۰۰ درخواست در دقیقه برای هر نام کاربری یا میزبان
builder.Services.AddRateLimiter(options =>
{
    options.GlobalLimiter = PartitionedRateLimiter.Create<HttpContext,
string>(httpContext =>
        RateLimitPartition.GetFixedWindowLimiter(
            partitionKey: httpContext.User.Identity?.Name ??
httpContext.Request.Headers.Host.ToString(),
```

```
            factory: _ => new FixedWindowRateLimiterOptions
            {
                PermitLimit = 100,
                Window = TimeSpan.FromMinutes(1),
                QueueLimit = 0
            }));
});

// ... سایر تنظیمات middlewareها
app.UseRateLimiter();
```

در مثال بالا، از یک **الگوی پنجره ثابت** (Fixed Window) استفاده شده که در هر پنجره ۱ دقیقه‌ای، حداکثر ۱۰۰ درخواست را می‌پذیرد. اگر کاربری بیش از این در یک دقیقه درخواست ارسال کند، Rate Limiting به طور خودکار درخواست‌های اضافی را رد کرده و معمولاً کد وضعیت ۴۲۹ (Too Many Requests) برمی‌گرداند. با این کار API ما در برابر ترافیک سنگین یا مخرب محافظت می‌شود [1] . به یاد داشته باشید که حتماً بعد از پیکربندی، `app.UseRateLimiter()` را در **Pipeline** فراخوانی کنید تا این میان‌افزار فعال شود [6] .

> **نکته:** Rate Limiting هرچند حملات DoS را تا حدی مهار می‌کند، اما راه‌حل کامل برای حملات **DDoS توزیع‌شده** نیست. برای حفاظت جامع‌تر، می‌توان از سرویس‌های ابری یا WAF بهره گرفت که ترافیک مخرب را در شبکه شناسایی و مسدود می‌کنند [7] [8] .

تنظیم CORS (اشتراک‌گذاری منابع بین مبدأها)

اگر کلاینت (مانند برنامه وب جاوااسکریپتی) روی دامنه یا پورت متفاوتی نسبت به API ما اجرا شود، لازم است **CORS (Cross-Origin Resource Sharing)** را به طور صحیح تنظیم کنیم. مرورگرها به صورت پیش‌فرض به صفحات وب اجازه نمی‌دهند به دامنه دیگری درخواست AJAX بفرستند مگر اینکه سرور مقصد صراحتاً اجازه دهد [9] . در پروژه فرضی ما، فرض کنید رابط کاربری فروشگاه بر روی دامنه `https://shop-client.com` اجرا می‌شود و API ما روی دامنه `https://api.myshop.com` است. برای اینکه مرورگر کلاینت بتواند به API درخواست بفرستد، باید روی API **سیاست CORS** تعریف کنیم که دامنه فرانت‌اند ما را مجاز بشمارد.

در ASP.NET Core، برای فعال‌سازی CORS ابتدا سرویس مربوطه را در `Program.cs` ثبت و سیاست‌های مجاز را تعریف می‌کنیم. سپس Middleware مربوط به CORS را در جای درست به Pipeline اضافه می‌کنیم [10] [11] . به عنوان نمونه، برای اجازه‌دادن به دامنه مشخصی (مثلاً دامنه فرانت‌اند ما) می‌توانیم چنین کدی اضافه کنیم:

```
var AllowedOrigin = "AllowedOriginPolicy";

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: AllowedOrigin, policy =>
    {
        policy.WithOrigins("https://shop-client.com") // دامنه مجاز
              .AllowAnyHeader()
              .AllowAnyMethod();
    });
});
```

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

// ... سایر middlewareها مانند app.UseRouting();

app.UseCors(AllowedOrigin);

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

در کد بالا، یک **Policy** به نام `"AllowedOriginPolicy"` تعریف شده که فقط اجازه می‌دهد مبدأ درخواست `https://shop-client.com` به API ما دسترسی داشته باشد و تمام هدرها و متدهای HTTP (GET, POST, ...) نیز مجاز هستند [12] [13]. سپس با `app.UseCors(AllowedOrigin)` این سیاست را اعمال می‌کنیم. توجه کنید که **ترتیب قرارگیری middlewareها بسیار مهم است**؛ طبق توصیه مایکروسافت، `UseCors` باید **بعد از** `UseRouting` و **قبل از** `UseAuthorization` فراخوانی شود تا به درستی عمل کند [14]. در غیر این صورت، ممکن است تنظیمات CORS اثر نکند.

با تنظیم صحیح CORS، مرورگر کلاینت ما در دامنه متفاوت می‌تواند درخواست‌های Ajax را به API ارسال کند و پاسخ دریافت نماید. توصیه می‌شود دامنه‌هایی که اجازه دسترسی دارند را محدود کنید (به‌جای `AllowAnyOrigin`) و فقط آن‌هایی را که نیاز است مجاز نمایید تا سطح امنیت افزایش یابد [15].

فشرده‌سازی پاسخ‌ها (Response Compression)

یکی از راه‌های بهبود کارایی API، **کاهش حجم داده‌های ارسالی در پاسخ** است. فشرده‌سازی پاسخ (مثلاً به صورت Gzip یا Brotli) موجب می‌شود حجم Payload کاهش یافته و سرعت انتقال و دریافت توسط کلاینت بیشتر شود [16]. در API فروشگاه آنلاین ما، تصور کنید اندپوینتی داریم که لیست محصولات (شامل ده‌ها یا صدها رکورد) را برمی‌گرداند. بدون فشرده‌سازی، این پاسخ ممکن است مثلاً ۵۰۰ کیلوبایت داده‌ی JSON باشد، اما با فعال کردن Gzip می‌توانیم اندازه آن را به میزان قابل توجهی کاهش دهیم (مثلاً به ۱۰۰ کیلوبایت)، که نتیجه آن **کاهش زمان دانلود و افزایش سرعت لود** در سمت کاربر است [16].

ASP.NET Core دارای Middleware داخلی برای **فشرده‌سازی پاسخ** است. برای فعال‌سازی آن، کافی است سرویس مربوطه را اضافه کرده و Middleware را در Pipeline قرار دهیم [17] [18]:

```
builder.Services.AddResponseCompression(); // ثبت سرویس‌های فشرده‌سازی (پیش‌فرض شامل Gzip/Brotli)
var app = builder.Build();
app.UseResponseCompression(); // فعال‌سازی میان‌افزار فشرده‌ساز پاسخ
```

تنها با همین چند خط، فشرده‌سازی برای MIME typeهای پیش‌فرض (متون JSON, HTML, CSS, JS و ...) فعال می‌شود [17]. کلاینت‌هایی که امکان دریافت محتوای فشرده را دارند با هدر `Accept-Encoding` درخواست خود را می‌فرستند و سرور نیز در صورت فعال بودن این middleware، پاسخ را با الگوریتم مناسب (مثلاً Brotli یا Gzip) فشرده کرده و با هدر `Content-Encoding` مربوطه ارسال می‌کند [19] [20]. مرورگر یا کلاینت HTTP نیز به طور خودکار این پاسخ را بازگشایی می‌کند.

به طور پیش‌فرض، ASP.NET Core اگر هیچ Provider خاصی اضافه نکنیم، هر دو الگوریتم **Brotli و Gzip** را فعال می‌کند و در مذاکره بین کلاینت و سرور، Brotli در اولویت قرار می‌گیرد (در صورتی که کلاینت پشتیبانی کند) [21] . ما می‌توانیم تنظیمات را سفارشی کنیم، مثلاً سطح فشرده‌سازی را تعیین کنیم یا یک الگوریتم را غیرفعال کنیم. به عنوان مثال:

```
builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true; // فعال‌سازی حتی برای درخواست‌های HTTPS (با احتیاط)
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
});
builder.Services.Configure<BrotliCompressionProviderOptions>(opts => opts.Level
= CompressionLevel.Fastest);
builder.Services.Configure<GzipCompressionProviderOptions>(opts => opts.Level =
CompressionLevel.SmallestSize);
```

با این تنظیمات، هر دو الگوریتم Brotli و Gzip اضافه شده‌اند و سطح فشرده‌سازی برای Brotli روی سریع‌ترین (کمترین فشار CPU) و برای Gzip روی بیشترین فشردگی (کوچک‌ترین حجم خروجی) تنظیم شده است [22] [23] .

> **توجه امنیتی:** به صورت پیش‌فرض فشرده‌سازی برای ترافیک HTTPS غیرفعال است زیرا حملاتی مانند **CRIME/BREACH** می‌توانند از ترکیب اطلاعات فشرده‌شده و رمزنگاری‌شده سوءاستفاده کنند [24] . اگر `EnableForHttps = true` را فعال می‌کنید، مراقب باشید که داده‌های حساس (مثل توکن‌ها) در خروجی‌های فشرده حضور نداشته باشند یا از **توکن‌های ضد-CSRF** استفاده کنید [24] . به هر حال، در بسیاری از APIهای عمومی فعال‌سازی فشرده‌سازی روی HTTPS رایج است تا کارایی بهتری حاصل شود اما ریسک‌های امنیتی باید در نظر گرفته شوند [25] .

نسخه‌بندی API (API Versioning)

با رشد و تغییر نیازمندی‌ها، ممکن است مجبور شویم در API خود **تغییرات Breaking** اعمال کنیم (مثلاً تغییر در ساختار داده‌های ورودی/خروجی یا قوانین بیزینسی). به‌منظور حفظ **پشتیبانی از نسخه‌های قدیمی** برای کلاینت‌هایی که هنوز از نسخه قبلی استفاده می‌کنند، بهترین روش معرفی **نسخه‌بندی API** است [26] [27] . نسخه‌بندی به ما اجازه می‌دهد چندین نسخه از API را به صورت همزمان ارائه دهیم؛ برای نمونه، نسخه 1.0 (قدیمی) و نسخه 2.0 (جدید) که در کنار هم کار می‌کنند و کلاینت می‌تواند مشخص کند با کدام نسخه کار می‌کند.

در پروژه فروشگاه آنلاین، فرض کنید نسخه v1 API شامل اندپوینت `GET /api/products` است که لیست محصولات را برمی‌گرداند. بعدها ممکن است تصمیم بگیریم در نسخه جدید این اندپوینت تغییر کند (مثلاً اطلاعات بیشتری برگرداند یا شیوه فیلتر نتایج عوض شود). با نسخه‌بندی، می‌توانیم یک اندپوینت جدید مثلاً در نسخه 2 ارائه کنیم بدون اینکه کاربران نسخه 1 دچار مشکل شوند. به عنوان مثال، URL نسخه 2 می‌تواند `GET /api/v2/products` باشد که تغییرات جدید را اعمال کرده است، در حالی که `GET /api/v1/products` همچنان رفتار قدیمی را حفظ می‌کند.

برای پیاده‌سازی نسخه‌بندی در ASP.NET Core، می‌توان از پکیج رسمی **Asp.Versioning** استفاده کرد. ابتدا باید NuGet مربوطه (`Asp.Versioning.Mvc` و همچنین `Asp.Versioning.ApiExplorer` برای ادغام با Swagger) را اضافه کنیم [28] . سپس در تنظیمات سرویس‌ها (Program.cs) نسخه‌بندی را فعال می‌کنیم:

```
builder.Services.AddApiVersioning(options =>
{
    options.DefaultApiVersion = new ApiVersion(1, 0);
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.ReportApiVersions = true;
    // تعیین روش‌های خواندن نسخه از درخواست:
    options.ApiVersionReader = ApiVersionReader.Combine(
        new UrlSegmentApiVersionReader(),       // خواندن نسخه از مسیر URL مثل /api/
v2/...
        new HeaderApiVersionReader("X-Version"),// خواندن نسخه از هدر سفارشی
        new QueryStringApiVersionReader("api-version") // خواندن نسخه از Query
String
    );
});
```

در کد بالا، نسخه پیش‌فرض API را 1.0 اعلام کرده‌ایم و مشخص کردیم اگر کلاینت نسخه را مشخص نکرد، همین نسخه پیش‌فرض در نظر گرفته شود [29]. همچنین `ReportApiVersions = true` باعث می‌شود در هدرهای پاسخ مشخص شود که چه نسخه‌هایی از API موجودند (این برای دیباگ و اطلاع‌رسانی مفید است و در محیط تولید می‌توان آن را غیرفعال کرد) [30]. روش‌های متعدد خواندن نسخه را نیز ترکیب کرده‌ایم: از قطعه مسیر URL، از Query String و از هدر مخصوص. (در عمل شما ممکن است فقط یکی از این روش‌ها را انتخاب کنید. **رایج‌ترین روش** نسخه‌بندی، گنجاندن شماره نسخه در URL است [31]، به عنوان مثال `/api/v2/products`.)

پس از تنظیم سرویس، می‌توانیم به کنترلرها و اکشن‌ها نسخه اختصاص دهیم. برای نمونه:

```
[ApiController]
[Route("api/v{version:apiVersion}/[controller]")]
[ApiVersion("1.0")]
public class ProductsController : ControllerBase
{
    [HttpGet]
    public IActionResult GetProductsV1() { ... }
}

[ApiController]
[Route("api/v{version:apiVersion}/[controller]")]
[ApiVersion("2.0")]
public class ProductsV2Controller : ControllerBase
{
    [HttpGet]
    public IActionResult GetProductsV2() { ... }
}
```

در این مثال، دو کنترلر جدا برای نسخه‌های 1 و 2 داریم که هر دو زیر مسیر `/api/v{version}/products` مپ می‌شوند اما هر کدام فقط به نسخه خاصی پاسخ می‌دهند. در روش دیگر می‌توان به جای چند کنترلر، با attributeهای `[MapToApiVersion("2.0")]` درون یک کنترلر، اکشن متفاوتی برای نسخه 2 تعریف کرد. در هر صورت، افزودن

نسخه به API تضمین می‌کند که **کلاینت‌های قدیمی** با نسخه قبلی کار کنند و همزمان **امکانات جدید** در نسخه بعدی عرضه شود بدون اینکه وقفه‌ای در سرویس‌رسانی ایجاد گردد [32] [33].

همچنین، نسخه‌بندی به مستندسازی API نیز کمک می‌کند؛ ابزار Swagger می‌تواند برای هر نسخه یک **صفحه مستندات جدا** تولید کند. (برای این کار باید Swagger را با ApiExplorer یکپارچه کرد که بعدتر در بخش مستندسازی توضیح داده می‌شود.)

مدیریت سراسری خطاها (Global Exception Handling)

در یک API حرفه‌ای، نباید اجازه دهیم **استثناهای کنترل‌نشده** باعث کرش کردن سرویس یا نشت جزئیات پیاده‌سازی به کاربر شوند. به جای آن، تمام خطاهای رخداده باید به شکل استاندارد و قابل پیش‌بینی مدیریت و گزارش شوند. **مدیریت سراسری استثناها** یعنی پیش‌بینی اینکه ممکن است در هر جای pipeline یا منطق برنامه خطایی رخ دهد و ما یک مکانیزم مرکزی برای گرفتن این خطاها و تولید پاسخ مناسب داشته باشیم [34] [35].

در سناریوی فروشگاه آنلاین، فرض کنید در حین پردازش یک سفارش خطایی در پایگاه‌داده یا یک NullReferenceException رخ می‌دهد. به جای اینکه برنامه با کد 500 بدون توضیح یا با یک traceback خام پاسخ دهد، ما می‌خواهیم یک **پاسخ خطای سازگار** (مثلاً یک JSON با جزئیات خطا) به کلاینت ارائه کنیم و لاگ مناسبی نیز در سرور ثبت شود. برای این منظور چند راه وجود دارد:

- پیاده‌سازی یک **Middleware سفارشی** که دور تمام درخواست‌ها یک بلوک try/catch قرار می‌دهد؛ هر خطای پرتاب‌شده را می‌گیرد، لاگ می‌کند و یک پاسخ خطا (مثلاً شامل **شیء ProblemDetails**) برمی‌گرداند [35] [36]. این روش در نسخه‌های قدیمی ASP.NET Core مرسوم بود.
- استفاده از **Middleware پیش‌فرض خطا** با `app.UseExceptionHandler`. این میان‌افزار به شما امکان می‌دهد یک مسیر جایگزین برای پردازش خطا تعیین کنید. برای نمونه، می‌توانید مشخص کنید اگر خطایی رخ داد، کنترل به مسیر `/error` منتقل شود و آن مسیر خودش یک پاسخ مناسب ایجاد کند [37] [38].

روش دوم را می‌توانیم در پروژه خود به‌کار بگیریم. ابتدا در زمان پیکربندی Pipeline، مشخص می‌کنیم که در حالت Production (غیرDevelopment) Middleware خطا فعال باشد:

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/error");
}
```

طبق کد بالا، در محیط Development ما صفحه خطای توسعه‌دهنده را داریم که جزئیات استثنا را نشان می‌دهد، اما در Production تمام خطاها به مسیر `/error` هدایت می‌شوند [39]. حال باید این مسیر را پیاده‌سازی کنیم. یک راه ساده، افزودن یک اکشن یا انتهای مسیر `/error` است که از متد کمکی `Problem()` برای تولید پاسخ خطا استفاده کند:

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error")]
    [ApiExplorerSettings(IgnoreApi = true)] // نمایش داده Swagger این مسیر در مستندات
```

```
نشود
    public IActionResult HandleError() => Problem();
}
```

متد `Problem()` به طور خودکار یک شیء **ProblemDetails** مطابق RFC 7807 تولید می‌کند و کد وضعیت مناسب (معمولاً 500 برای خطاهای بدون دسته‌بندی خاص) را برمی‌گرداند [40]. مزیت این روش این است که ساختار ثابتی برای خطاها ایجاد می‌کند. خروجی نمونه ممکن است شبیه زیر باشد:

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.6.1",
  "title": "Internal Server Error",
  "status": 500,
  "traceId": "00-...-00"
}
```

البته می‌توانیم ProblemDetails را شخصی‌سازی هم بکنیم (مثلاً یک شناسه خطا یا پیغام خاص اضافه کنیم)، اما به عنوان پیش‌فرض همین اطلاعات کلی کفایت می‌کند و از **نمایش جزییات حساس روی محیط Production جلوگیری می‌شود** [37] [40]. همچنین با `[ApiExplorerSettings(IgnoreApi=true)]` مطمئن شدیم این اندپوینت خطا در Swagger مستند نشود.

روش‌های پیشرفته‌تری نیز برای Global Exception Handling وجود دارد. برای مثال، در .NET 8 به بعد، یک قابلیت جدید به نام **IExceptionHandler** معرفی شده است که می‌توانیم یک کلاس سفارشی پیاده کنیم و به عنوان ExceptionHandler ثبت کنیم [41] [42]. این Handler می‌تواند انواع خاصی از استثناها را به صورت شرطی مدیریت کند. اما در بسیاری از موارد، همان روش middleware سفارشی یا `UseExceptionHandler` کفایت می‌کند. نکتهٔ مهم این است که با هر رویکردی، **تمام خروجی‌های خطا در API شکل یکنواختی داشته باشند** تا مصرف‌کنندگان API بتوانند به راحتی خطاها را پردازش کنند.

در کنار برگرداندن پاسخ مناسب، **ثبت لاگ خطا** نیز حیاتی است. اطمینان حاصل کنید که در لایهٔ global error handler، جزئیات Exception (پیام، استک‌ترِیس و ...) در لاگ‌های سمت سرور ذخیره شود تا بعداً بتوانید مشکلات را عیب‌یابی کنید [43]. نمونه‌ای از لاگ کردن خطا در Middleware سفارشی:

```
_logger.LogError(exception, "Exception occurred: {Message}", exception.Message);
```

در مثال بالا، متن و جزییات خطا در سطح Error در لاگ ذخیره می‌شود [43]. در ادامه بخش لاگینگ، بیشتر به این موضوع می‌پردازیم.

کش کردن پاسخ‌ها (Response Caching / Output Caching)

بسیاری از درخواست‌های به API، به‌خصوص درخواست‌های GET که داده‌های نسبتاً ثابت برمی‌گردانند، را می‌توان **کش کرد (Cache)** تا نیازی به اجرای دوباره منطق و کوئری پایگاه‌داده برای هر بار درخواست نباشد. کش کردن پاسخ‌ها یکی از موثرترین راه‌ها برای **افزایش کارایی و مقیاس‌پذیری** API است [44] [45]. فرض کنیم در API فروشگاه، یک اندپوینت برای دریافت لیست دسته‌بندی‌های محصولات داریم که به ندرت تغییر می‌کند. به جای اینکه هر بار به پایگاه‌داده وصل شویم، می‌توانیم نتیجه این اندپوینت را مثلاً به مدت ۱ ساعت کش کنیم. به این ترتیب، اگر ۱۰۰ کلاینت ظرف آن ساعت

درخواست این لیست را بدهند، فقط اولین درخواست واقعاً پردازش کامل می‌شود و ۹۹ تای بعدی سریعاً پاسخ را از کش دریافت می‌کنند که بسیار سریع و کم‌هزینه است.

ASP.NET Core از نسخه 7 به بعد، **Output Caching Middleware** داخلی ارائه کرده است (مشابه OutputCache قدیمی در ASP.NET MVC). برای استفاده از آن، ابتدا باید سرویس مربوطه را ثبت کنیم و سپس Middleware را در Pipeline فعال کنیم [45] [46]:

```
builder.Services.AddOutputCache();

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

فراخوانی `AddOutputCache` و `UseOutputCache` به تنهایی caching را فعال نمی‌کند، بلکه زیرساخت آن را آماده می‌سازد [47]. برای اینکه خروجی یک اندپوینت خاص کش شود، یا باید از **attribute** مخصوص `[OutputCache]` روی آن اکشن/مسیر استفاده کنیم یا در حالت مینیمال API، از extension method به نام `CacheOutput()` بهره ببریم [48].

برای برنامه‌های مبتنی بر کنترلر، ساده‌ترین راه این است که روی اکشن مربوطه `[OutputCache]` قرار دهیم. مثلاً:

```
[HttpGet]
[OutputCache(Duration = 60)] // کش کردن خروجی این اکشن به مدت ۶۰ ثانیه
public IEnumerable<Category> GetCategories() { ... }
```

با این کار، خروجی این متد GET برای ۶۰ ثانیه در سرور کش می‌شود و درخواست‌های بعدی در طی این بازه بدون اجرای مجدد کد، همان پاسخ قبلی را دریافت می‌کنند. زمانی که Duration منقضی شود، اولین درخواست بعدی دوباره نتیجه را تولید و در کش قرار می‌دهد.

همچنین می‌توان سیاست‌های کش پیشرفته‌تری تعریف کرد. مثلاً تعیین کرد که کش بر اساس پارامترهای query متغیر باشد یا برای کاربران احراز هویت‌شده cache جداگانه شود. Output Caching Middleware امکانات متعددی دارد (شامل Vary کردن بر اساس headerها یا queryها). در مثال ما، می‌توانیم **Policy** تعریف کنیم:

```
builder.Services.AddOutputCache(options =>
{
    options.AddPolicy("CategoryCache", policy =>
policy.Expire(TimeSpan.FromMinutes(10)));
});
```

و سپس در اکشن به جای Duration، از آن Policy استفاده کنیم:

```
[OutputCache(PolicyName = "CategoryCache")]
```

این Policy بالا تعیین می‌کند که هر پاسخ پس از ۱۰ دقیقه منقضی شود [49].

نکته مهم دیگر، **ترتیب** middlewareهای Cache نسبت به CORS و Authorization است. معمولاً OutputCache باید **بعد از** احراز هویت و CORS بیاید تا از تداخل با آنها جلوگیری شود [50] . (به عنوان مثال، محتوای کش‌شده شاید بسته به کاربر تغییر کند، لذا معمولاً محتوا را برای درخواست‌های عمومی یا GETهای بدون نیاز به کاربر کش می‌کنیم.)

تفاوت Response Caching و Output Caching: ASP.NET Core قبلاً نیز مفهومی به نام Response Caching داشت که مبتنی بر هدرهای HTTP کار می‌کرد (و برای پروکسی‌ها/مرورگرها کش را ممکن می‌ساخت). اما Output Caching که در اینجا بحث شد، **کش سمت سرور** است و کاملاً از سمت سرور پاسخ آماده را برمی‌گرداند. در نتیجه برای بهبود کارایی سرور بسیار موثرتر است.

در مجموع، استفاده از caching در اندپوینت‌های **پر مصرف و کم‌تغییر** توصیه می‌شود. البته باید دقت کنیم که اندپوینت‌های حساس به کاربر (مثل اطلاعات پروفایل که برای هر کاربر متفاوت است) یا داده‌های لحظه‌ای (مثل موجودی انبار که سریع تغییر می‌کند) را کش نکنیم یا حداقل **تنظیمات انقضای کوتاه** برایشان قرار دهیم که داده کهنه به کاربر برنگردد.

استفاده از async/await و جلوگیری از بلاک کردن

پلتفرم .NET از مدل asynchronous پشتیبانی کامل دارد و در ASP.NET Core به صورت طراحی‌شده این امکان هست که با استفاده از async/await **صدها یا هزاران درخواست همزمان** را بدون وقفه مدیریت کنیم [51] . بنابراین یک بهترین شیوه کلیدی این است که در تمام بخش‌های I/O-محور (دسترسی به پایگاه‌داده، فراخوانی APIهای دیگر، خواندن/نوشتن فایل، و ...) از متدهای **غیرهمزمان (async)** استفاده کنیم و از بلاک‌کردن Threadها پرهیز کنیم [52] . بلاک شدن Thread (مثلاً با متدهای synchronous طولانی‌مدت) باعث می‌شود Thread Pool اشباع شود و در نهایت سرویس‌دهی به سایر درخواست‌ها کند یا متوقف شود [53] .

برای مثال، در API فروشگاه آنلاین هر جا به دیتابیس کوئری می‌زنیم (مثلاً با Entity Framework)، حتماً از متدهای async مثل `ToListAsync()` یا `SaveChangesAsync()` استفاده می‌کنیم. یا وقتی منتظر پاسخ یک سرویس خارجی هستیم، مثلاً ارسال یک درخواست پرداخت به درگاه بانک، آن را await می‌کنیم به جای اینکه با .Result منتظر بمانیم. بدین شکل، Thread مربوطه می‌تواند آزاد شود و درخواست‌های دیگر را پردازش کند تا زمانی که پاسخ آماده شود [52] .

برخی **ضد-الگوها** وجود دارند که باید از آنها اجتناب کرد [54] :

- هرگز با فراخوانی `.Wait()` یا `.Result` روی Taskهای async منتظر نشوید. این کار رشته جاری را بلاک کرده و مزیت Async را از بین می‌برد و می‌تواند منجر به بن‌بست یا ThreadPool Starvation شود [55] .
- از ایجاد Thread دستی یا `Task.Run` بلا فاصله قبل از await خودداری کنید. در محیط ASP.NET Core کد ما **خودش روی ThreadPool** اجرا می‌شود، ایجاد Task جدید فقط سربار اضافی دارد مگر کار واقعاً CPU-bound سنگینی باشد که باید جدا شود [56] . در کل برای عملیات I/O لازم نیست با Task.Run آن را غیرهمزمان کنید، کتابخانه‌های EF، HttpClient و ... خودشان متد async دارند.
- تا حد امکان متدهای Controller و تمام زنجیره فراخوانی را async تعریف کنید تا مزیت end-to-end Async محقق شود [57] . یعنی متد اکشن کنترلر به جای `IActionResult` از نوع `Task<IActionResult>` باشد و داخل آن از await استفاده شود. این یک الگوی رایج است که تمام Stack غیرهمزمان باشد.

با رعایت این موارد، API ما می‌تواند **همزمانی** بالایی را تحمل کند و بهینه از منابع استفاده نماید. برعکس، اگر جایی Threadها را بلاک کنیم (مثلاً صبر روی یک قفل یا عملیات کند)، تأثیر منفی مستقیمی بر کارایی تحت بار زیاد خواهد داشت [53] .

همچنین توصیه می‌شود در متدهای Async در صورت امکان از **CancellationToken** استفاده کنیم. ASP.NET Core به طور خودکار یک CancellationToken مرتبط با درخواست کاربر به متدهای کنترلر async تزریق می‌کند (اگر در امضای متد بگنجانید). با این کار، اگر کاربر اتصال را قطع کرد یا درخواستش Timeout شد، عملیات در حال اجرا می‌تواند کنسل شود و منابع آزاد شوند.

تزریق وابستگی (Dependency Injection) و معماری ماژولار

فریم‌ورک ASP.NET Core به طور درونی بر پایه **تزریق وابستگی (DI)** بنا شده است. کنترلرهای ما،Middlewareها و سرویس‌های داخلی از کانتینر DI تغذیه می‌شوند. استفاده صحیح از DI چندین مزیت بزرگ دارد: **جداسازی‌concerns**، کاهش coupling، امکان **تست واحد آسان‌تر** و تنظیمات پیکربندی منعطف.

در پروژه فروشگاه آنلاین، ما منطق‌های مختلفی خواهیم داشت (مثل سرویس مدیریت سفارش، سرویس پرداخت، سرویس ایمیل و ...). بهترین شیوه این است که این منطق‌ها را در کلاس‌های جدا (مثلاً کلاس‌های سرویس) پیاده کنیم و این کلاس‌ها را از طریق واسطهایشان (interfaces) به کنترلرها تزریق کنیم. برای مثال:

```
public interface IOrderService { void PlaceOrder(Order order); }
public class OrderService : IOrderService
{
    private readonly IPaymentGateway _payment;
    public OrderService(IPaymentGateway payment) { _payment = payment; }
    public void PlaceOrder(Order order)
    {
        // پردازش سفارش
        _payment.Process(order.Total);
        // ...
    }
}
```

سپس در `Program.cs` این سرویس‌ها را به کانتینر DI معرفی می‌کنیم:

```
builder.Services.AddScoped<IOrderService, OrderService>();
builder.Services.AddScoped<IPaymentGateway, PaymentGateway>();
```

از این پس، هر جا `IOrderService` نیاز باشد (مثلاً در سازنده یک کنترلر)، خود فریم‌ورک یک نمونه از OrderService را ایجاد و تزریق می‌کند. کنترلر نمونه‌وار:

```
[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase
{
    private readonly IOrderService _orderService;
    public OrdersController(IOrderService orderService)
    {
```

```
        _orderService = orderService;
    }

    [HttpPost]
    public IActionResult CreateOrder(OrderDto dto)
    {
        // تبدیل DTO به مدل حوزه (Domain) و اعتبارسنجی...
        _orderService.PlaceOrder(order);
        return Ok();
    }
}
```

بدین ترتیب کنترلر ما از جزئیات پردازش سفارش ایزوله است و فقط با اینترفیس کار می‌کند. **مزیت‌ها:** اگر بعدها منطق PlaceOrder تغییر کند یا پیاده‌سازی دیگری اضافه شود (مثلاً OrderServiceV2)، کنترلر بدون تغییر باقی می‌ماند. یا اگر بخواهیم این کنترلر را تست کنیم، می‌توانیم یک `Mock` از IOrderService به آن تزریق کنیم و رفتارهای مختلف را شبیه‌سازی کنیم.

چند **اصل مهم در طراحی با DI** [58] :

- حتی‌الامکان از **ایجاد مستقیم اشیاء وابسته با** `new` در داخل کلاس‌ها خودداری کنید [59] . این کار وابستگی سفت‌وسخت ایجاد می‌کند. به جای آن، وابستگی‌ها را از بیرون (کانتینر) دریافت کنید. به عنوان مثال، در OrderService اگر PaymentGateway وابسته است، آن را تزریق کردیم به جای اینکه داخل PlaceOrder شی جدید از PaymentGateway بسازیم.
- از **کلاس‌ها یا داده‌های سراسری (global state)** پرهیز کنید. به عنوان جایگزین، اگر نیاز به حالت مشترک در کل برنامه دارید، می‌توانید از Singletonها در DI استفاده کنید (مثلاً یک Singleton برای کش یا لاگر) [58] . اما دقت کنید Singleton باید **بدون وضعیت mutable** (یا Thread-safe) باشد چون بین همه درخواست‌ها مشترک است.
- اصل **تک وظیفه‌ای (SRP)** را رعایت کنید. اگر کلاسی بیش از حد وابستگی دارد (مثلاً می‌بینید ۵-۶ سرویس مختلف تزریق می‌کند)، احتمالاً آن کلاس چند مسئولیت دارد و بهتر است به چند بخش کوچک‌تر تفکیک شود [60] .
- **Lifetime مناسب** برای سرویس‌ها انتخاب کنید:
- سرویس‌های سبک و بی‌حالت را Transient ثبت کنید (`AddTransient`).
- سرویس‌های وابسته به **سشن وب یا واحد کاری هر درخواست** (مثل DbContext در EF Core) را Scoped ثبت کنید (`AddScoped`)، که یعنی در طول یک درخواست وب یک نمونه مشترک خواهد بود.
- سرویس‌هایی که می‌خواهید در تمام برنامه **تنها یک نمونه** داشته باشند (مثل Logger یا HttpClientFactory یا MemoryCache) را Singleton ثبت کنید (`AddSingleton`).

هر سرویس بر اساس ماهیت خود یکی از این سه مدل Lifetime را خواهد داشت. انتخاب Lifetime صحیح هم از نظر کارایی (تعداد دفعات ساخت شیء) مهم است، هم از نظر صحت عملکرد (مثلاً اگر DbContext را Singleton کنید بین درخواست‌ها اشتراکی می‌شود که غلط است).

از دیگر مزایای DI در ASP.NET Core این است که بسیاری از قابلیت‌های داخلی به صورت extension method فراهم شده‌اند. برای مثال با `builder.Services.AddControllers()` یا `AddSwaggerGen()` و غیره، کل گروهی از سرویس‌های موردنیاز به یکباره ثبت می‌شوند [61] [62] . شما نیز می‌توانید برای ماژول‌های پروژه خود extension method تعریف کنید تا سرویس‌های مربوط به آن ماژول را دسته‌ای به DI اضافه کند و Program خواناتر شود [63] [64] .

به طور خلاصه، **تزریق وابستگی** قلب معماری قابل نگهداری در ASP.NET Core است. با بهره‌گیری از آن، اجزای سیستم شما کم‌تر به هم وابسته‌اند و تست و توسعه آن‌ها ساده‌تر خواهد بود.

مستندسازی خودکار API (OpenAPI/Swagger Documentation)

هر چه یک API کامل و کارا بنویسیم، اگر **مستندسازی** خوبی نداشته باشد، استفاده از آن برای دیگران دشوار خواهد بود. یکی از مزایای دنیای .NET Core، وجود ابزار قدرتمندی به نام **Swagger** (و استاندارد OpenAPI) است که می‌تواند به شکل **خودکار** مستندات API شما را تولید کند و حتی یک رابط کاربری تعاملی (Swagger UI) برای تست اندپوینت‌ها فراهم کند [65].

در پروژه فروشگاه آنلاین، احتمالاً API ما توسط برنامه‌نویسان فرانت‌اند یا توسعه‌دهندگان موبایل مورد استفاده قرار می‌گیرد. ما می‌توانیم با افزودن Swagger، یک صفحه وب داشته باشیم که تمام مسیرهای API (به تفکیک نسخه‌ها)، ورودی و خروجی هر کدام، و روش فراخوانی آن‌ها را نشان دهد. این کار باعث صرفه‌جویی در زمان تیم‌ها و کاهش سوءتفاهم‌ها می‌شود.

در تصویر بالا نمونه‌ای از رابط Swagger UI را مشاهده می‌کنید که برای نسخه‌های مختلف API اطلاعات را نمایش می‌دهد. این رابط به توسعه‌دهندگان اجازه می‌دهد به راحتی **اندپوینت‌ها را آزمایش** کنند، پارامترهای ورودی را تغییر دهند و نتایج را ببینند.

راه‌اندازی Swagger در ASP.NET Core ساده است. ابتدا پکیج `Swashbuckle.AspNetCore` را از NuGet نصب می‌کنیم. سپس در `Program.cs` اضافه می‌کنیم [66]:

```
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

سطر `AddEndpointsApiExplorer` برای پشتیبانی از Minimal APIها است و در صورت استفاده از کنترلرها الزامی نیست اما ضرری هم ندارد. سپس پس از ساختن app، Middlewareهای Swagger را اضافه می‌کنیم:

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

در بالا تعیین کرده‌ایم که Swagger فقط در محیط Development فعال باشد [65] (زیرا معمولاً مستندات را روی محیط Production عمومی نمی‌گذارند یا حداقل پشت احراز هویت قرار می‌دهند). `UseSwagger()` خروجی JSON (فرمت OpenAPI) را در مسیر مثلاً `/swagger/v1/swagger.json` فراهم می‌کند و `UseSwaggerUI()` رابط گرافیکی زیبا در مسیر `/swagger` را فراهم می‌کند که توسعه‌دهندگان می‌توانند با مرورگر مشاهده کنند [65].

Swagger به طور خودکار براساس **اشیاء مسیرها، اکشن‌ها و مدل‌های دیتا** مستندات را می‌سازد. اما بهتر است برای خوانایی بیشتر، از **کامنت‌های XML** برای توضیح هر اندپوینت و مدل نیز استفاده کنیم. با افزودن تنظیمات `<GenerateDocumentationFile>true</GenerateDocumentationFile>` در فایل csproj پروژه [67] و فعال

کردن Include نظرات XML در Swagger (تنظیمات SwaggerGen)، هر کامنت سه‌خطی (`///`) که بالای اکشن‌ها و کلاس‌های مدل نوشته‌ایم در مستندات نمایش داده خواهد شد [68] . مثلاً اگر بالای متد کنترلر `GetProducts` توضیحی بنویسیم که چه می‌کند، در Swagger UI زیر عنوان آن متد نمایش داده می‌شود.

علاوه بر این، می‌توانیم اطلاعات کلی API مانند عنوان، توضیحات، ورژن و اطلاعات تماس را نیز در Swagger تنظیم کنیم تا صفحه مستندات حرفه‌ای‌تر به نظر برسد [68] [69] . نمونه:

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "MyShop API",
        Version = "v1",
        Description = "API فروشگاه آنلاین ما",
        Contact = new OpenApiContact { Name = "Team X", Email =
"support@myshop.com" }
    });
});
```

با این کار، در بالای صفحه Swagger UI عنوان و توضیحات API نمایش داده می‌شود که به فهم بهتر مخاطبان کمک می‌کند.

نتیجه استفاده از Swagger/OpenAPI این است که در زمان بسیار کوتاه، یک **مستندات زنده** برای API خود داریم که همیشه به‌روز است (چون مستقیماً از کد تولید می‌شود) و تیم‌های دیگر می‌توانند بدون نیاز به ماژول‌های اضافی، خودشان قابلیت‌های API را کشف کنند. در عین حال، ما نیز می‌توانیم Swagger UI را برای تست‌های دستی سریع اندپوینت‌هایمان استفاده کنیم.

> 🔒 **نکته امنیتی:** مستندات Swagger حاوی اطلاعات دقیقی از API شماست (مسیرها، مدل‌ها و شاید نمونه‌ها). بهتر است این مستندات را روی محیط Production محدود کنید یا پشت احراز هویت قرار دهید تا فقط افراد مجاز بتوانند ببینند. راه دیگر این است که Swagger را تنها در محیط‌های توسعه و تست فعال کنید و در محیط عملیاتی غیرفعال باشد [65] .

لاگینگ و نظارت بر رویدادها (Logging)

ثبت وقایع (Logging) بخش مهم دیگری از یک API حرفه‌ای است. لاگ‌ها چشم شما درون اجرای برنامه در محیط عملیاتی هستند و به کمک آن‌ها می‌توانید رفتار سیستم را مانیتور و مشکلات را ردیابی کنید. ASP.NET Core یک سازوکار logging داخلی بر پایه واسط `ILogger` ارائه می‌کند که با DI قابل استفاده است [70] [71] . بهترین شیوه این است که در هر کلاس کلیدی (مثلاً کنترلرها، سرویس‌های اصلی) یک نمونه `ILogger<T>` تزریق کنید و رویدادهای مهم را لاگ نمایید.

در سناریوی فروشگاه، رویدادهای مهم شامل مواردی مثل: دریافت یک درخواست جدید، ثبت شدن یک سفارش، وقوع یک خطای غیرمنتظره، اقدام یک کاربر غیرمجاز برای دسترسی و ... هستند. باید تعیین کنید هر نوع رویداد در چه **سطحی** (Log Level) ثبت شود:

- رویدادهای معمول سیستم (مثل دریافت درخواست یا انجام یک عمل Business) را در سطح **Information** یا **Debug** ثبت کنید.
- رویدادهای مشکوک یا هشداردهنده (مثلاً تلاش لاگین ناموفق یا پاسخ خیلی کند) را در سطح **Warning** ثبت کنید.
- رویدادهای خطا (Exceptions) را در سطح **Error** یا **Critical** ثبت کنید، همراه با جزئیات Exception.

به عنوان نمونه، در `OrdersController` وقتی سفارش جدیدی با موفقیت ثبت شد، می‌توانیم چنین لاگی داشته باشیم:

```
_logger.LogInformation("Order {OrderId} placed by User {User}", order.Id,
userId);
```

این لاگ با سطح Information ذخیره می‌شود (که معمولاً در محیط Production هم فعال است) و از **template** استفاده کردیم تا داده‌هایی مثل OrderId و UserId را به‌صورت ساخت‌یافته در لاگ قرار دهیم. استفاده از پیام‌های قالب‌دار (به جای کانکات کردن استرینگ) توصیه می‌شود چون سیستم Logging می‌تواند این‌ها را به صورت ساختاری (Structured Logging) پردازش کند.

در موارد خطا هم همان‌طور که در بخش قبل ذکر شد، حتماً Exception را با `_logger.LogError(ex, "...")` ثبت کنید [43] . ILogger به طور خودکار استک‌ترِیس Exception و پیام آن را در خروجی می‌آورد. شما می‌توانید یک پیام کاربرپسند نیز اضافه کنید. برای مثال:

```
catch(Exception ex)
{
    _logger.LogError(ex, "خطای غیرمنتظره در ثبت سفارش کاربر {UserId}", userId);
    throw; // تبدیل کند HTTP خطا آن را به پاسخ Middleware مجدداً پرتاب می‌کنیم تا
}
```

نکته مهم دیگر، **عدم ثبت اطلاعات بسیار حساس** در لاگ‌هاست. هرچند لاگینگ برای Debug عالی است، اما نباید مثلاً پسورد کاربران یا توکن‌های JWT را به صورت متن‌واضح در لاگ‌ها بنویسید. این موارد را ماسک کنید یا کلاً لاگ نکنید تا اگر لاگ به بیرون درز کرد، مشکلی ایجاد نشود.

ASP.NET Core Logging به صورت پیش‌فرض خروجی Console را فراهم می‌کند (و در IIS هم به EventLog می‌فرستد). اما شما می‌توانید از کتابخانه‌های شخص ثالث مثل **Serilog** یا **NLog** استفاده کنید تا لاگ‌ها را به فایل، دیتابیس یا سیستم‌های مرکزی لاگ (مانند Elasticsearch/Kibana یا Seq) بفرستید. برای یک سیستم در مقیاس تولید، معمولاً یک **زیرساخت لاگ مرکزی** وجود دارد که لاگ‌های تمامی سرویس‌ها را جمع‌آوری می‌کند و شما می‌توانید با جستجو و فیلتر، مشکلات را در آن پیدا کنید.

به صورت خلاصه، **هر رویداد مهم یا خطا را لاگ کنید** و سطح مناسب را به آن اختصاص دهید. این کار در کنار مانیتورینگ سلامت سرویس (که در ادامه می‌آید) به شما امکان می‌دهد سیستم را حتی پس از استقرار نیز زیر نظر داشته باشید و به سرعت به مشکلات واکنش نشان دهید.

احراز هویت و مجوزدهی با JWT

بیشتر APIهای امروزی نیازمند نوعی **امنیت** هستند؛ بدین معنی که همه کاربران یا سرویس‌ها در سطح دسترسی نتوانند آزادانه به تمام اندپوینت‌ها دسترسی داشته باشند. یکی از رایج‌ترین روش‌های **احراز هویت Stateless** در APIها، استفاده از **JSON Web Token (JWT)** است. در این روش، کاربر پس از یک‌بار login (مثلاً ارسال نام‌کاربری/رمزعبور به یک اندپوینت توکن)، یک توکن JWT دریافت می‌کند. این توکن حاوی اطلاعات پایه‌ای درباره کاربر (Claims) و زمان انقضا است و توسط سرور امضا شده است. سپس کاربر در درخواست‌های بعدی این توکن را در هدر Authorization می‌فرستد (`Authorization: Bearer <JWT>`). سرور می‌تواند بدون نیاز به نگهداری Session سمت خود، با اعتبارسنجی امضای JWT و بررسی Claimsهای آن، کاربر را احراز هویت و سطح دسترسی‌اش را تعیین کند [72] [73].

در پروژه فروشگاه آنلاین، فرض کنید برخی اندپوینت‌ها عمومی هستند (مثلاً دریافت لیست محصولات)، اما اندپوینت ثبت سفارش یا مشاهده سفارش‌های کاربر باید تنها در صورتی قابل استفاده باشد که کاربر **وارد سیستم شده** و JWT معتبر داشته باشد. بنابراین ما مکانیزم JWT را برای API پیاده می‌کنیم:

ابتدا نیاز است JWT Bearer Authentication را در ASP.NET Core تنظیم کنیم. در فایل تنظیمات (appsettings.json یا محیط) یک **کلید امنیتی** برای امضای JWT (اگر از امضای متقارن HS256 استفاده می‌کنیم) یا تنظیمات گواهی (برای RSA) مشخص می‌کنیم. برای سادگی، فرض کنیم یک Secret key داریم. سپس در `Program.cs`:

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidIssuer = "MyShopAuthServer",
            ValidateAudience = true,
            ValidAudience = "MyShopAPI",
            ValidateLifetime = true,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("SuperSecretKey12345"))
        };
    });
```

در اینجا ما مشخص کردیم که توکن JWT باید Issuer (صادرکننده) برابر "MyShopAuthServer" و Audience (مصرف‌کننده) برابر "MyShopAPI" داشته باشد و همچنین تاریخ انقضا بررسی شود. کلید امضای متقارن را نیز از یک رشته ثابت (نمونه) تولید کردیم. در عمل بهتر است این کلید را از پیکربندی بخوانید و در کد ثابت ننویسید. پس از این تنظیمات، Middleware احراز هویت را هم باید اضافه کنیم:

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

از این پس، هر درخواست به اندپوینت‌های محافظت‌شده JWT ابتدا توسط Middleware احراز هویت بررسی می‌شود. اگر هدر Authorization با Bearer token موجود باشد، سعی می‌کند آن را اعتبارسنجی کند [74]. در صورت موفقیت، یک

Identity در HttpContext User قرار می‌گیرد که حاوی Claimsهای کاربر است و درخواست به کنترلر می‌رسد. اگر توکن موجود نباشد یا نامعتبر باشد، پاسخ 401 Unauthorized برگردانده می‌شود [74] [75] .

برای علامت‌گذاری اینکه کدام اندپوینت‌ها نیاز به احراز هویت دارند، می‌توانیم روی کنترلر یا اکشن‌ها از `[Authorize]` استفاده کنیم. مثلاً:

```
[Authorize]
[HttpPost("orders")]
public IActionResult PlaceOrder(OrderDto dto) { ... }
```

با این attribute، اطمینان می‌یابیم که فقط کاربرانی که JWT معتبر در هدر دارند می‌توانند این اندپوینت را صدا بزنند. اگر فردی بدون JWT یا با JWT نامعتبر تلاش کند، فریم‌ورک قبل از ورود به اکشن 401 برمی‌گرداند. همچنین در صورتی که کاربر احراز هویت شده ولی به نقش/سیاست موردنیاز دسترسی نداشته باشد، پاسخ 403 Forbidden دریافت می‌کند [76] .

نکات تکمیلی در مورد JWT:

- **زمان انقضا:** برای JWTها حتماً زمان انقضا (Expiry) کوتاه در نظر بگیرید (مثلاً ۱۵ دقیقه یا ۱ ساعت) [77] . این باعث می‌شود اگر توکن کسی لو رفت، دائمی نباشد. برای تجربه کاربری بهتر می‌توان از **Refresh Token**های بلندمدت به همراه توکن‌های کوتاه عمر استفاده کرد.
- **انتقال امن:** JWT حتماً باید روی کانال HTTPS منتقل شود تا شنود نشود. همچنین می‌توان برای امنیت بیشتر از **JWT امضاشده + رمزگذاری‌شده** استفاده کرد (JWE) اما معمولاً امضاشده کفایت می‌کند.
- **اندازه JWT:** از آنجا که JWT در هر درخواست ارسال می‌شود، خیلی آن را بزرگ نکنید و Claimهای غیرضروری داخلش نگذارید (مثلاً اطلاعات پروفایل را در JWT قرار ندهید، بهتر است فقط UserId و نقش‌ها و موارد ساده باشد).
- **Authorization پیشرفته:** می‌توانیم از **Policyها و Roleها** در ASP.NET Core استفاده کنیم تا دسترسی به برخی اندپوینت‌ها را محدودتر کنیم. برای مثال، `[Authorize(Roles="Admin")]` یا تعریف Policy که مثلاً کاربر حتماً ایمیلش تأیید شده باشد. این موارد با JWT هم کار می‌کنند (Claimsهای داخل JWT را تطبیق می‌دهند).

در کل، JWT یک روش انعطاف‌پذیر و سبک برای امن‌سازی API است و در سناریوهای موبایل و SPA که Stateful بودن ممکن نیست، بسیار جوابگوست. با پیاده‌سازی صحیح آن، اطمینان حاصل می‌کنیم که فقط کاربران مجاز قادر به انجام عملیات حساس (مثل ایجاد سفارش یا دیدن اطلاعات شخصی) هستند و API ما در برابر دسترسی غیرمجاز محافظت می‌شود.

اعتبارسنجی ورودی‌ها (Input Validation)

همواره باید ورودی‌هایی که از کلاینت دریافت می‌کنیم را **بررسی و اعتبارسنجی** کنیم. هرگز نباید به داده‌ای که کاربر می‌فرستد بدون وارسی اعتماد کنیم، چرا که ممکن است ناقص، اشتباه یا حتی مخرب باشد. در ASP.NET Core، اگر کنترلرهای ما با attribute `[ApiController]` علامت‌گذاری شده باشند، چارچوب به صورت خودکار بسیاری از اعتبارسنجی‌های ساده را انجام می‌دهد و در صورت نامعتبر بودن ورودی، یک پاسخ خطای ۴۰۰ Bad Request برمی‌گرداند [78] .

برای بهره‌مندی از این قابلیت، کافی است **Data Annotation** های مناسب را روی مدل‌های DTO یا بایندشونده تعریف کنیم. برای مثال، در پروژه فروشگاه یک مدل `RegisterUserDto` داریم برای ثبت‌نام کاربر که باید ایمیل و پسورد معتبری داشته باشد. این مدل را می‌توانیم به شکل زیر تزئین کنیم:

```
public class RegisterUserDto
{
    [Required(ErrorMessage = "ایمیل الزامی است")]
    [EmailAddress(ErrorMessage = "فرمت ایمیل نادرست است")]
    public string Email { get; set; }

    [Required]
    [StringLength(100, MinimumLength = 6, ErrorMessage = "پسورد باید حداقل ۶ کاراکتر
باشد")]
    public string Password { get; set; }

    [Compare("Password", ErrorMessage = "پسوردها مطابقت ندارند")]
    public string ConfirmPassword { get; set; }
}
```

با attributes بالا، فریم‌ورک به‌طور خودکار در زمان مدل‌بایندینگ بررسی می‌کند که فیلد Email و Password خالی نباشند (`[Required]`)، Email فرمتی شبیه ایمیل داشته باشد و طول Password در محدوده مشخص باشد [79] [80]. اگر هر یک از این قواعد نقض شود، قبل از آنکه اکشن کنترلر اجرا شود، فریم‌ورک یک پاسخ 400 Bad Request همراه با جزئیات خطا (لیست پیام‌های اعتبارسنجی) برمی‌گرداند [78]. این رفتار پیش‌فرض `[ApiController]` است که ModelState را چک کرده و خطاها را به کلاینت گزارش می‌کند، بنابراین نیازی نیست ما دستی `ModelState.IsValid` را بررسی کنیم (بر خلاف MVC سنتی) [81].

خروجی خطای 400 پیش‌فرض شبیه زیر خواهد بود:

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "...",
  "errors": {
    "Email": [ "فرمت ایمیل نادرست است" ],
    "Password": [ "پسورد باید حداقل ۶ کاراکتر باشد" ]
  }
}
```

همان‌طور که می‌بینید، جزئیات تمام خطاهای اعتبارسنجی در بخش `errors` آمده است. این ساختار نیز یک نوع ProblemDetails است و کار کلاینت را برای نمایش یا پردازش خطاها آسان می‌کند.

uuuuuuu (The user text included some presumably invalid or nonsensical part here, possibly my guess; I'll remove that as it's not needed, maybe some leftover.)

البته DataAnnotations برای اعتبارسنجی‌های ساده مناسب است. اگر نیاز به **اعتبارسنجی پیچیده‌تر** (مثلاً وابسته به پایگاه‌داده یا منطق تجاری) باشد، می‌توان از روش‌های دیگری استفاده کرد: مانند کتابخانه FluentValidation (که با ASP.NET Core ادغام می‌شود) یا نوشتن attributeهای سفارشی. اما همچنان توصیه می‌شود خطاهای اعتبارسنجی را با کد 400 به کلاینت اعلام کنید. می‌توانید پیام‌های خطا را بومی‌سازی کرده یا کدهای خطای خاص خود را در پاسخ ارسال کنید تا کلاینت به‌صورت برنامه‌نویسی آن‌ها را رسیدگی کند.

از منظر امنیتی، **اعتبارسنجی ورودی نخستین دیوار دفاعی** در برابر حملاتی مثل SQL Injection یا XSS نیز هست. اگر شما اطمینان حاصل کنید که ورودی‌ها در قالب و دامنه مورد انتظار هستند، احتمال کمتری دارد داده مخربی به لایه‌های پایین‌دست (مثل ORM یا مرورگر) راه یابد. البته برای Injectionها حتماً از پارامترهای امن در کوئری‌ها و برای XSS خروجی امن (مثلاً استفاده از امکانات Encoding در Razor یا Frontend) نیز باید استفاده شود، اما بحث ما اینجا بر روی API است که معمولاً داده خام می‌دهد.

خلاصه این که: **همیشه اعتبارسنجی** ! هر مدلی که از کاربر می‌گیرید را صحت‌سنجی کنید و در صورت خطا، پیام مناسب با ۴۰۰ به او بدهید. این کار هم سرویس شما را از داده‌های ناسازگار محافظت می‌کند، هم تجربه توسعه‌دهنده API را بهتر می‌کند چون دقیق می‌داند چه اشتباهی رخ داده است.

نام‌گذاری صحیح Endpoints و رعایت اصول RESTful

طراحی **رابط‌های RESTful** علاوه بر بحث‌های فنی، یک جنبه مهم **طراحی URIها و قراردادها** را نیز شامل می‌شود. بهترین شیوه این است که API ما **خوانا و شهودی** باشد، به‌طوری‌که از روی مسیر و نوع درخواست بتوان حدس زد چه عملی انجام می‌شود. برای رسیدن به این هدف، باید از نام‌گذاری‌های استاندارد استفاده کنیم [82] [83] :

- **استفاده از اسامی (اسم) به جای افعال در URL**: مسیرهای API باید تا حد امکان بر اساس **منابع (Resources)** نام‌گذاری شوند، نه بر اساس عملیات. برای مثال، به جای اینکه مسیر ثبت سفارش را `/api/CreateOrder` بگذاریم، بهتر است از اسم جمع استفاده کنیم: `/api/orders` و سپس از متد HTTP POST برای ایجاد استفاده کنیم [84] [85] . به‌طور کلی، فعل عملیات در خود متد HTTP مستتر است: GET برای خواندن، POST برای ایجاد، PUT/PATCH برای به‌روزرسانی، DELETE برای حذف. بنابراین URIها می‌توانند فقط معرف منابع باشند.
- **استفاده از اسم جمع برای مجموعه‌ها**: هنگامی که یک resource قابل شمارش داریم، شکل جمع آن را در مسیر استفاده کنیم. برای مثال `/api/products` نشان‌دهنده مجموعه همه محصولات است [86] ، و `/api/products/123` یک مورد منفرد از آن مجموعه (محصول با شناسه 123) را نمایش می‌دهد. این شیوه در تمام APIهای معروف دیده می‌شود و کاربران را سردرگم نخواهد کرد.
- **سازمان‌دهی سلسله‌مراتبی روابط**: اگر بین منابع رابطه والد/فرزند هست، می‌توان آن را در URI منعکس کرد. مثلاً در فروشگاه ما، سفارش‌ها وابسته به کاربر هستند. ممکن است بخواهیم اندپوینتی داشته باشیم که تمام سفارش‌های یک کاربر خاص را بدهد؛ می‌توان طراحی کرد: `/api/users/{userid}/orders` [87] . این نشان می‌دهد orders زیرمجموعه‌ای از یک user است. البته باید در طراحی این روابط زیاده‌روی نکرد (حداکثر 2 یا 3 سطح، مثلاً Resource/Resource/Resource). اگر تو در توی خیلی پیچیده شود، بهتر است از query parameters استفاده کنیم یا روابط را جداگانه هندل کنیم [88] [89] . اما یک سطح تو در تو (مثل مثال بالا) کاملاً قابل قبول و فهم است.
- **استفاده از حروف کوچک و خط تیره**: URIهای RESTful عرفاً با حروف کوچک هستند و اگر چند کلمه باشند با خط تیره (`-`) جدا می‌شوند نه underscore یا camelCase. مثلاً ترجیحاً `/api/user-orders` نسبت به `/api/UserOrders`. این موضوع روی کارکرد تأثیری ندارد اما یک رسم معمول است که به خوانایی کمک می‌کند [90] .
- **عدم encode اطلاعات غیرضروری در مسیر**: گاهی دیده می‌شود برخی APIها ورژن یا فرمت یا کلید API را در مسیر می‌آورند که توصیه نمی‌شود. برای نسخه از الگوی قبل (`/api/v1/...`) استفاده کنید. برای فرمت

(JSON/XML) از header `Accept` استفاده شود نه دو مسیر جدا. برای کلید API هم از query string یا header استفاده کنید، نه اینکه جزئی از path باشد.

- **برگرداندن Status Codeهای مناسب:** این هم جزئی از قرارداد RESTful است. مثلاً وقتی یک منبع جدید با POST ایجاد می‌شود، **کد 201 Created** و در هدر Location آدرس منبع جدید ارسال شود. یا وقتی منبعی وجود ندارد، 404 Not Found. برای درخواست‌های نامعتبر 400 و برای دسترسی غیرمجاز 401/403 (همان‌طور که قبلاً بحث شد) [91] [76] . کلاینت‌ها بر اساس این Status codeها رفتار مناسبی نشان می‌دهند؛ مثلاً اگر 201 بگیرند می‌دانند چیزی ساخته شده. یا 204 No Content برای حذف موفق شاید مناسب باشد چون بادی‌ای ندارد. رعایت این جزییات نشان‌دهنده توجه شما به استانداردها و کمک به سازگاری کلاینت‌هاست.

در مثال پروژه ما، ما اندپوینت‌های اصلی را این‌گونه طراحی می‌کنیم:

- `GET /api/products` – دریافت فهرست محصولات (ممکن است با کوئری‌استرینگ فیلتر یا صفحه‌بندی هم پشتیبانی شود).
- `GET /api/products/{id}` – دریافت جزئیات محصول با شناسه مشخص.
- `POST /api/orders` – ثبت یک سفارش جدید (نیازمند احراز هویت).
- `GET /api/orders/{id}` – دریافت جزئیات یک سفارش (فقط در صورت مالک بودن یا ادمین بودن).
- `GET /api/users/{id}/orders` – دریافت همه سفارش‌های کاربری با شناسه مشخص (احتمالاً برای ادمین یا برای خود کاربر با {id} = me).
- و ...

همان‌طور که می‌بینید، همه اسم‌ها جمع و انگلیسی هستند و هیچ فعلی مثل `Get` یا `Create` در مسیرها نیامده است. این سادگی باعث می‌شود URIها تقریبا **خودتوصیف‌گر** باشند. بهره گرفتن از اصول معماری REST بدین معناست که از امکانات استاندارد پروتکل HTTP حداکثر استفاده را ببریم و نیاز به ابداع قراردادهای جدید نباشد.

سایر نکات و بهترین شیوه‌های تکمیلی

علاوه بر موارد مهمی که تا الان پوشش دادیم، چند نکته دیگر نیز وجود دارد که رعایت آن‌ها به بهبود کیفیت API ما کمک می‌کند:

- **استفاده از Health Check:** برای سرویس‌های وب حرفه‌ای، داشتن یک اندپوینت سلامت (معمولاً `/health` یا `/healthz`) توصیه می‌شود. این اندپوینت می‌تواند وضعیت اجزای حیاتی سیستم را گزارش دهد (مثل اتصال به دیتابیس، در دسترس بودن سرویس‌های وابسته). پکیج Microsoft.Extensions.Diagnostics.HealthChecks امکانات زیادی برای این کار فراهم می‌کند. با `builder.Services.AddHealthChecks()` می‌توانید HealthCheckهای دلخواه (مثلاً پینگ دیتابیس) را ثبت کنید و سپس با `app.MapHealthChecks("/health")` یک مسیر برای گزارش سلامتی سیستم ایجاد کنید. ابزارهای ارکستری چون Kubernetes می‌توانند به طور دوره‌ای این مسیر را فراخوانی کنند و اگر پاسخی حاکی از خرابی بود، اقدام مقتضی (مثل ری‌استارت کردن کانتینر) انجام دهند.
- **مدیریت پیکربندی و Secrets:** بهتر است تمامی **رشته‌های اتصال، کلیدهای راز و تنظیمات متغیر** را در appsettings.json یا منابع امن (مثل Azure Key Vault یا Secret Manager داکر) نگهداری کنید و آن‌ها را در کد هاردکد نکنید. .NET Core قابلیت **Options Pattern** را ارائه می‌کند که می‌توانید تنظیمات را به صورت کلاس strongly-typed در DI تزریق کنید (`IOptions<T>`). این شیوه، مدیریت تنظیمات را انعطاف‌پذیر و تغییر آن‌ها را بدون نیاز به کامپایل مجدد ممکن می‌کند.
- **استفاده صحیح از HttpClient:** اگر API ما نیاز دارد به سرویس‌های خارجی HTTP فراخوانی کند (مثلاً می‌خواهیم در ثبت سفارش، یک درخواست به سرویس بانکی یا سرویس ارسال ایمیل بیرونی بفرستیم)، حتماً از **IHttpClientFactory** استفاده کنیم. ساخت دستی HttpClient برای هر درخواست یک ضد الگو است که می‌تواند منجر به مصرف بی‌رویه سوکت‌ها و مشکلات کارایی شود [92] . به جای آن،

`builder.Services.AddHttpClient()` را استفاده کنید که HttpClientها را در پشت صحنه **مدیریت و بهینه‌سازی** می‌کند (Connection Pooling، تنظیم Timeout و ...). طبق راهنمای رسمی، یا از یک HttpClient سراسری با تنظیم زمان lifetime استفاده کنید یا از IHttpClientFactory برای ایجاد HttpClientهای موقت بهره ببرید [93] . این کار از مشکل اتمام پورت‌های TCP در اثر باز و بسته کردن مکرر اتصالات جلوگیری می‌کند [94] و همچنین اگر DNS تغییری داشت، HttpClientFactory با expire کردن Handlerها، مورد جدید را resolve می‌کند [95] [92] .

- **نسخه‌بندی و Deployment بدون قطعی:** هنگام انتشار نسخه‌های جدید API، حتماً نسخه قبلی را تا مدتی حفظ کنید تا کلاینت‌ها فرصت به‌روزرسانی داشته باشند. به عبارتی **Backward Compatibility** را رعایت کنید مگر اینکه به دلایل تجاری مجبور شوید نسخه قبل را از کار بیندازید. در آن صورت، از قبل به مصرف‌کنندگان اطلاع‌رسانی کنید. این یک بهترین practice در سطح مدیریت چرخه عمر API است.
- **استفاده از Status Codeهای معنادار:** ما در طول مقاله به این مورد اشاره کردیم اما تأکید مجدد می‌کنیم که همیشه متناسب با نتیجه عملیات، کد وضعیت صحیح برگردانید. برخی برنامه‌نویسان عادت دارند همیشه 200 OK بدهند و خطاها را در payload مشخص کنند؛ این روش RESTful نیست. بهتر است از امکانات پروتکل بهره ببرید و مثلاً 404 را واقعاً برای "یافت نشد" بفرستید، 400 را برای "درخواست نادرست"، 401 برای "احراز هویت نشده"، 403 برای "ممنوع/مجوز ناکافی"، 500 برای "خطای سرور" و ... . این کار خوانایی API را برای همه افزایش می‌دهد [91] [76] .
- **طراحی برای مقیاس‌پذیری:** هر زمان که امکان دارد، کارها را **غیرهمزمان و ناپیوسته** کنید. مثلاً اگر ثبت سفارش شامل اطلاع‌رسانی ایمیل به کاربر است، شاید بهتر باشد این ایمیل را به صف پیام (مثلاً Azure Service Bus یا RabbitMQ) بفرستید تا به صورت جداگانه ارسال شود و لازم نباشد کاربر منتظر ارسال ایمیل بماند [96] . استفاده از **پیام‌محوری و صف** یک الگوی پیشرفته‌تر است که در سیستم‌های بزرگ بسیار کاربرد دارد (مفهومی مثل eventual consistency).
- **نظارت بر Performance:** حتماً در محیط‌های Production، نظارتی بر عملکرد API داشته باشید. استفاده از **Application Insights** یا ابزارهای APM دیگر کمک می‌کند متوجه شوید کدام اندپوینت‌ها کند هستند یا خطای زیادی دارند. این اطلاعات برای بهبود مستمر API ارزشمند است.
- **Logging توزیع‌شده و Trace Id:** زمانی که معماری شما فراتر از یک سرویس برود (مثلاً چند میکروسرویس داشته باشید)، مهم است که درخواست کاربر را در کل سیستم بتوانید ردیابی کنید. ASP.NET Core به طور پیش‌فرض از طریق Middlewareها یک `Trace-Id` در هر درخواست ایجاد می‌کند (یا اگر از بیرون آمد پاس می‌دهد) [97] . مطمئن شوید این TraceId را در همه لاگ‌هایتان چاپ می‌کنید (خوشبختانه ILogger این کار را اگر تنظیم کنید به صورت خودکار انجام می‌دهد). با این کار اگر یک خطا در سفارش رخ داد که به دو سرویس دیگر هم درخواست زده، می‌توانید با جستجوی TraceId در لاگ‌های همه سرویس‌ها، زنجیره کامل رویدادها را بیابید.
- **نام‌گذاری و سازمان‌دهی کد:** این شاید مستقیماً به API مرتبط نباشد، اما برای تجربه توسعه‌دهندگان نگهدارنده API مهم است. پروژه را به لایه‌های منطقی تقسیم کنید (Controllers، Services، Repositories، Models، DTOs و غیره). نام‌گذاری کلاس‌ها و متدها را گویا انتخاب کنید. کامنت‌گذاری جایی که پیچیدگی وجود دارد فراموش نشود. رعایت استاندارد کدنویسی (Code Style) و استفاده از آنالیزرها نیز توصیه می‌شود تا کد تمیزتری داشته باشید.

در پایان، ترکیب این بهترین شیوه‌ها باعث می‌شود API شما **ماندگار و قابل ارتقاء** باشد. یک توسعه‌دهنده باتجربه با دیدن چنین APIیی متوجه انسجام و کیفیت آن خواهد شد. همچنین کاربران API (چه همکاران داخلی تیم شما و چه توسعه‌دهندگان خارجی) تجربه بهتری خواهند داشت چرا که API شما قابل فهم‌تر، قابل اتکاتر و قابل پیش‌بینی‌تر است.

با پیشرفت پلتفرم .NET و مخصوصاً نسخه‌های جدید (.NET 9، .NET 8 و به زودی .NET 10)، ابزارها و امکانات بیشتری برای بهبود APIها در اختیار ما قرار می‌گیرد. اما اصول بنیادی معماری خوب و بهترین شیوه‌های ذکر شده، مستقل از نسخه، معتبر هستند. امیدواریم با به‌کارگیری این موارد در پروژه فرضی فروشگاه آنلاین و هر پروژه واقعی دیگری، APIهایی بسازید که از هر لحاظ «درجه یک» باشند!

منابع و مراجع:

1. "Arvin Kahbazi, Maarten Balliauw, Rick Anderson, "Rate limiting middleware in ASP.NET Core", *Microsoft Learn*, 2025 [5] [98]
2. "Rick Anderson, Kirk Larkin, "Enable Cross-Origin Requests (CORS) in ASP.NET Core", *Microsoft Learn*, 2024 [12] [14]
3. "Response compression in ASP.NET Core", *Microsoft Learn*, 2024 [17] [22]
4. "Ricardo Peres, "ASP.NET Core API Versioning", *Development With A Dot*, 2024 [29] [30]
5. "Milan Jovanović, "Global Error Handling in ASP.NET Core 8", *milanjovanovic.tech blog*, 2023 [42] [43]
6. "Tom Dykstra, "Output caching middleware in ASP.NET Core", *Microsoft Learn*, 2024 [45] [49]
7. "Mike Rousos, "ASP.NET Core Best Practices", *Microsoft Learn*, 2024 [53] [54]
8. "Dependency injection guidelines", *Microsoft Learn*, 2024 [58]
9. "Getting Started with Swashbuckle and ASP.NET Core", *Microsoft Learn*, 2023 [65] [68]
10. "Handle errors in ASP.NET Core web APIs", *Microsoft Learn*, 2024 [38] [40]
11. "Configure JWT bearer authentication in ASP.NET Core", *Microsoft Learn*, 2024 [74] [75]
12. "Model validation in ASP.NET Core MVC", *Microsoft Learn*, 2024 [79] [81]
13. "Web API design best practices", *Microsoft Learn (Azure Architecture Center)*, 2022 [84] [86]
14. "HttpClient Guidelines for .NET", *Microsoft Learn*, 2025 [94] [99]

Rate limiting middleware in ASP.NET Core | Microsoft Learn [1] [2] [3] [4] [5] [6] [7] [8] [98]
https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit?view=aspnetcore-9.0

Enable Cross-Origin Requests (CORS) in ASP.NET Core | Microsoft Learn [9] [10] [11] [12] [13] [14] [15]
https://learn.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-9.0

Response compression in ASP.NET Core | Microsoft Learn [16] [17] [18] [19] [20] [21] [22] [23] [24] [25]
https://learn.microsoft.com/en-us/aspnet/core/performance/response-compression?view=aspnetcore-9.0

Development With A Dot - ASP.NET Core API Versioning [26] [27] [28] [29] [30] [31] [32] [33]
/https://prod-static-asp-blogs.azurewebsites.net/ricardoperes/asp-net-core-api-versioning

Global Error Handling in ASP.NET Core 8 [34] [35] [36] [41] [42] [43]
https://www.milanjovanovic.tech/blog/global-error-handling-in-aspnetcore-8

Handle errors in ASP.NET Core controller-based web APIs | Microsoft Learn [37] [38] [39] [40] [97]
https://learn.microsoft.com/en-us/aspnet/core/web-api/handle-errors?view=aspnetcore-9.0

Using ASP.NET Core Output Caching - ServiceStack [44]
https://servicestack.net/posts/redis-outputcache

Output caching middleware in ASP.NET Core | Microsoft Learn [45] [46] [47] [48] [49] [50]
https://learn.microsoft.com/en-us/aspnet/core/performance/caching/output?view=aspnetcore-9.0

ASP.NET Core Best Practices | Microsoft Learn [51] [52] [53] [54] [55] [56] [57] [96]
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/best-practices?view=aspnetcore-9.0

Dependency injection guidelines - .NET | Microsoft Learn [58] [59] [60]
https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-guidelines

Dependency injection in ASP.NET Core | Microsoft Learn [61] [62] [63] [64] [70] [71]
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0

Get started with Swashbuckle and ASP.NET Core | Microsoft Learn [69] [68] [67] [66] [65]
https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-8.0

Configure JWT bearer authentication in ASP.NET Core | Microsoft Learn [91] [77] [76] [75] [74] [73] [72]
?https://learn.microsoft.com/en-us/aspnet/core/security/authentication/configure-jwt-bearer-authentication
view=aspnetcore-9.0

Model validation in ASP.NET Core MVC | Microsoft Learn [81] [80] [79] [78]
https://learn.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-9.0

Web API Design Best Practices - Azure Architecture Center | Microsoft Learn [89] [88] [87] [86] [85] [84] [83] [82]
https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

Top REST API URL naming convention standards | TheServerSide [90]
https://www.theserverside.com/video/Top-REST-API-URL-naming-convention-standards

HttpClient guidelines for .NET - .NET | Microsoft Learn [99] [95] [94] [93] [92]
https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/http/httpclient-guidelines