BitSpire

EBOOK

MODERNE WEBENTWICKLUNG

Anatomie einer modernen ASP.NET Web-Anwendung

Moderne Ansätze der Web-Entwicklung, die oft in Vergessenheit geraten, und warum gute Architektur den Unterschied macht. Denkanstösse, keine Dogmen.

By
BitSpire

We <build> software that matters

INHALT

9 Kapitel

Inhalt

Ein praxisnaher Streifzug durch die **Anatomie einer modernen ASP.NET-Anwendung**, von der Architektur über Dependency Injection, Endpunkte und die Middleware-Pipeline bis zu Caching und Testing.

ÜBERSICHT

Was enthält dieses eBook

Dieses eBook zeigt moderne Ansätze der Web-Entwicklung auf, die im Alltag oft in Vergessenheit geraten. Es liefert Denkanstösse, keine Dogmen.

Warum ist gute Architektur wichtig?

01 **Weniger Komplexität**

Weniger technische Komplexität und kognitive Last, denn was nicht da ist, kann nicht vergessen gehen.

02 **Erweiterbarkeit**

Neue Anforderungen lassen sich einfacher realisieren, ohne bestehende Teile zu gefährden.

03 **Developer Experience**

Glückliche Entwickler haben mehr Zeit fürs Wesentliche.

ÜBER UNS

Hey, wir sind BitSpire ;)

Wir sind **.NET-Experten aus der Schweiz**, unter anderem Microsoft MVP und Open-Source-Contributor. Wir haben viele Jahre Erfahrung im Bau und in der Wartung von Schnittstellen, meist in Cloud-Umgebungen.

Von der Beratung von CTOs über Architektur bis hin zu Workshops im .NET-Bereich. Wir lieben Code, und noch viel mehr:
We <love> software that matters!

WAS WIR MACHEN

- Individualsoftware
- Beratung
- Architektur
- Workshops

Mit uns zusammenarbeiten?
Kein Problem, Sie erreichen uns jederzeit.

contact@bitspire.ch
bitspire.ch

ARCHITEKTUR

Was macht gute Architektur aus?

Gute Architektur ist kein Selbstzweck und schon gar keine Frage von möglichst vielen Layern oder Design Patterns. Sie zeigt sich daran, wie leicht sich ein Team damit bewegt: heute und in zwei Jahren.

Klare Verantwortlichkeiten

Jede Komponente weiss genau, wofür sie zuständig ist und wofür nicht. Separation of Concerns reduziert, was man gleichzeitig im Kopf behalten muss.

Lose Kopplung, hohe Kohäsion

Module lassen sich unabhängig ändern, testen und deployen, ohne dass an zehn anderen Stellen etwas kaputt geht.

Konsistenz statt Beliebigkeit

Wiederkehrende Probleme werden im ganzen Projekt gleich gelöst. Das senkt die Einstiegshürde für neue Teammitglieder massiv.

Testbarkeit von Anfang an

Lässt sich Business-Logik nicht isoliert testen, ist das meist ein Architektur-Problem, kein Test-Problem.

Einfachheit vor Cleverness

Die einfachste Lösung, die das Problem löst, schlägt langfristig fast immer die elegantere, komplexere Variante.

Bewusste Abhängigkeiten

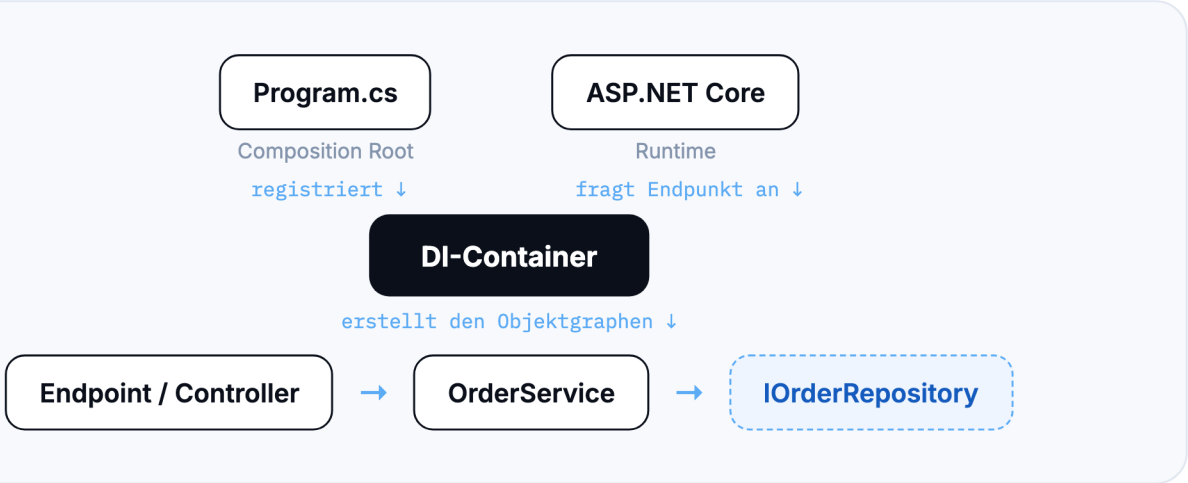
Man weiss, wo Grenzen liegen (z. B. zwischen Domäne, Infrastruktur und Web-Layer) und warum Abhängigkeiten in eine bestimmte Richtung zeigen.

> Gerade im ASP.NET-Umfeld entscheidet sich vieles davon schon in der **Middleware-Pipeline**, dort, wo Requests durch die Anwendung fliessen und Cross-Cutting Concerns wie Logging, Auth oder Error Handling sauber (oder eben unsauber) organisiert werden.

DEPENDENCY INJECTION

Dependency Injection

Dependency Injection (DI) bedeutet: Eine Klasse erstellt ihre Abhängigkeiten nicht selbst, sondern erhält sie von aussen, normalerweise über den Konstruktor. **Inversion of Control** beschreibt das übergeordnete Prinzip: Die Objekterzeugung liegt beim Framework bzw. beim Container, nicht bei jeder einzelnen Klasse.



Der Container kennt die Zuordnung zwischen Abstraktion und Implementierung. Fordert ASP.NET Core einen Endpunkt an, löst er dessen Konstruktorparameter, und deren Abhängigkeiten, rekursiv auf. Der Domänencode kennt nur das, was er braucht, nicht dessen konkrete Erzeugung.

```
// Composition Root: hier werden konkrete Entscheidungen getroffen.
builder.Services.AddScoped<IOrderService, OrderService>();
builder.Services.AddScoped<IOrderRepository, SqlOrderRepository>();
// Domänencode: benötigt eine Fähigkeit, keine konkrete Datenbankklasse.
public sealed class OrderService(IOrderRepository orders) : IOrderService
{
    // ...
}
```

Das macht Abhängigkeiten sichtbar, austauschbar und testbar: Ein Test kann z. B. `IOrderRepository` durch einen Fake ersetzen, ohne `OrderService` zu ändern. Constructor Injection ist der Normalfall; `GetRequiredService` im Domänencode versteckt Abhängigkeiten und macht Klassen schwerer testbar.

Die passende Lifetime wählen

Lifetime	Instanz	Typischer Einsatz
`Transient`	Bei jedem Auflösen neu	Kleine, zustandslose Helfer
`Scoped`	Einmal pro HTTP-Request	Use Cases, Repositories, `DbContext`
`Singleton`	Einmal für die ganze App	Konfiguration, Caches, thread-sichere Dienste

> Ein Singleton darf nicht von einem Scoped Service abhängen: Es würde eine request-gebundene Instanz über deren Lebenszeit hinaus festhalten. Gerade `DbContext` ist deshalb in Web-Anwendungen üblicherweise Scoped.

DEPENDENCY INJECTION · SCOPED 04 / 09

Scoped: eine Arbeitseinheit pro Request

Innerhalb eines Requests erhalten alle Consumer dieselbe Instanz; mit dem Ende des Requests wird sie entsorgt. Der nächste Request beginnt mit einem frischen Scope und frischen Services.

Ein Request entspricht häufig einer zusammenhängenden fachlichen Operation, etwa „Bestellung anlegen“. Deshalb passt das **Unit-of-Work**-Pattern natürlich dazu: Services und Repositories eines Requests teilen sich einen `DbContext`, sammeln Änderungen in einer Arbeitseinheit und speichern sie gemeinsam. Ob daraus eine atomare Transaktion wird, entscheidet die konkrete Datenbankoperation; der Scope gibt die richtige fachliche und technische Grenze vor.

```
// AddDbContext registriert den Context standardmässig als Scoped.
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(connectionString));
// Beide erhalten im selben Request dieselbe AppDbContext-Instanz.
builder.Services.AddScoped<IOrderRepository, OrderRepository>();
builder.Services.AddScoped<IInvoiceRepository, InvoiceRepository>();
```

So entsteht keine versehentlich geteilte Datenbank-Session zwischen Requests, und der `DbContext` inklusive Change Tracker lebt exakt so lange wie die Operation, die ihn benötigt.

ENDPUNKTE

Controllers vs. Minimal APIs

Minimal APIs sind ein alternativer Weg, HTTP-Endpunkte in ASP.NET Core zu bauen. Sie verwenden dieselbe Hosting-, Routing-, Middleware- und DI-Infrastruktur wie Controller, bringen aber nicht automatisch den ganzen MVC-Unterbau mit. Ein Endpoint ist eine Funktion, die mit `MapGet`, `MapPost` und ähnlichen Methoden registriert wird.

	Controller	Minimal API
Aufbau	Klassen mit Actions und Attributen	Funktionen, direkt auf Routen abgebildet
Standard	MVC-Konventionen, Discovery, Action Filters, attributbasierte Metadaten	Kleiner Startpunkt; Features werden gezielt ergänzt
Abhängigkeiten	Über den Konstruktor des Controllers	Direkt als Parameter des Route Handlers
Passend für	Grosse, konventionsreiche APIs, etablierter MVC-Stil	Kleine bis mittlere APIs, spezialisierte Services, AOT-nahe Workloads

Minimal bedeutet nicht „ohne Architektur“. Es bedeutet: **nur die benötigten Bausteine standardmässig mitbringen**. Der funktionale Stil macht die Abhängigkeiten eines Endpoints besonders sichtbar:

```
app.MapGet("/orders/{id:guid}", async (
    Guid id,
    IOrderService orderService,
    CancellationToken cancellationToken) =>
{
    var order = await orderService.GetByIdAsync(id, cancellationToken);
    return order is null
        ? TypedResults.NotFound()
        : TypedResults.Ok(order);
});
```

TypedResults statt Results

`Results.Ok(...)` gibt nur das allgemeine Interface `IResult` zurück. `TypedResults.Ok(...)` gibt dagegen den konkreten Typ `Ok<T>` zurück. Das ist etwas ausführlicher, hat aber zwei Vorteile: Unit Tests können den konkreten Response-Typ direkt prüfen, und die OpenAPI-Generierung kennt Statuscode und Schema ohne zusätzliche `.Produces(...)`-Aufrufe.

Gibt ein Endpoint mehrere Resultate zurück, beschreibt ein Union Result genau diesen Vertrag:

```
private static async Task<Results<Ok<OrderDto>, NotFound>> GetById(
    Guid id,
    IOrderService orderService,
    CancellationToken cancellationToken)
{
    var order = await orderService.GetByIdAsync(id, cancellationToken);
    return order is null
        ? TypedResults.NotFound()
        : TypedResults.Ok(order);
}
```

> Der Compiler verhindert, dass versehentlich ein nicht deklarierter Response-Typ zurückgegeben wird. Die Signatur dokumentiert zugleich: Dieser Endpoint liefert entweder `200 OK` mit `OrderDto` oder `404 Not Found`. Mehr dazu in der **Microsoft-Dokumentation zu TypedResults**.

ENDPUNKTE · STRUKTUR 05 / 09

Gruppen und Extension Methods statt langer `Program.cs`

Auch Minimal APIs müssen nicht in einer unübersichtlichen `Program.cs` enden. `MapGroup` bündelt einen gemeinsamen Route-Prefix sowie Metadaten wie Tags oder Authorization. Eine Extension Method gibt dem Modul einen klaren Platz, ähnlich wie ein Controller, aber ohne dessen technischen Unterbau:

```
// Program.cs
app.MapOrderEndpoints();
public static class OrderEndpointExtensions
{
    public static IEndpointRouteBuilder MapOrderEndpoints(
        this IEndpointRouteBuilder endpoints)
    {
        var orders = endpoints.MapGroup("/orders")
            .WithTags("Orders")
            .RequireAuthorization();
        orders.MapGet("/{id:guid}", GetById)
            .WithName("GetOrderById");
        return endpoints;
    }
    // private static ... GetById( ... ) wie zuvor
}
```

> Beim Controller liegen gemeinsame Abhängigkeiten typischerweise im Konstruktor und gelten damit oft für alle Actions, auch wenn nur eine sie braucht. Das macht Controller nicht falsch; für komplexe APIs mit vielen gemeinsamen Konventionen bleiben sie eine sehr gute Wahl.

ENDPUNKTE · NATIVE AOT

Native AOT: wenn Startzeit und Speicher zählen

Bei **Native AOT** wird die Anwendung beim Veröffentlichen vorab in nativen Code für eine konkrete Zielplattform wie `linux-x64` kompiliert. Der JIT-Compiler wird zur Laufzeit nicht mehr benötigt; zusammen mit Trimming, bei dem nicht erreichbarer Code aus dem Deployment entfernt wird, kann das Startzeit und Memory Footprint deutlich reduzieren.

PASST GUT

Cloud-native und Serverless-Workloads, AOT-taugliche Azure Functions, kleine Microservices mit vielen Instanzen oder häufigen Cold Starts.

PASST SCHLECHT

Runtime-Reflection und dynamische Codeerzeugung. MVC und damit Controller sind derzeit nicht vollständig AOT-kompatibel, da Discovery stark auf Reflection basiert.

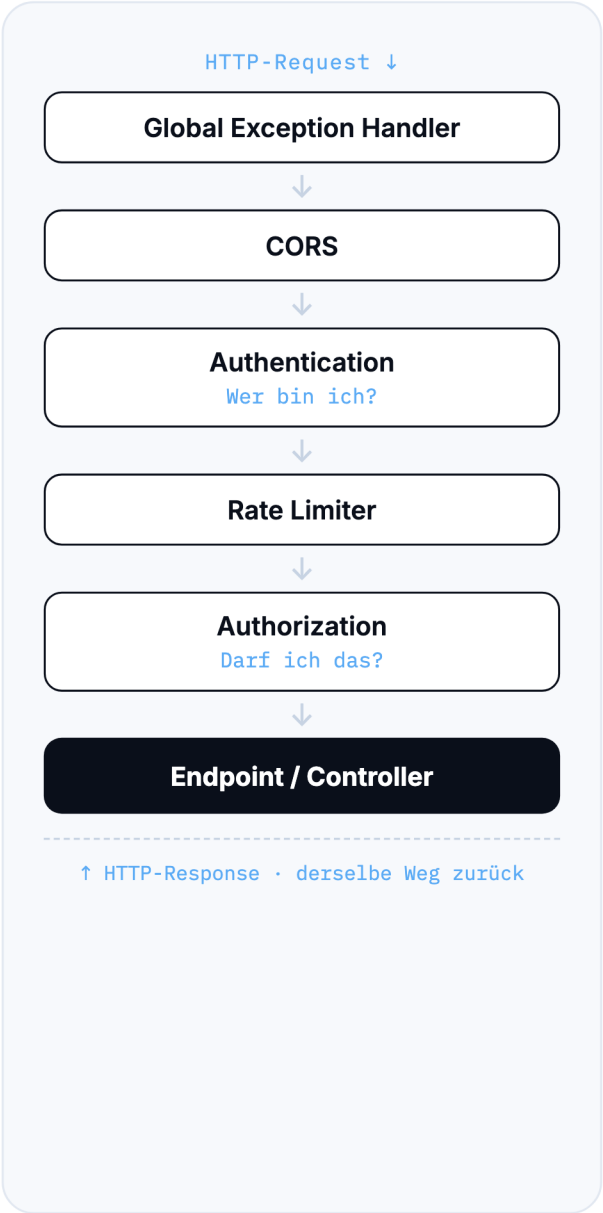
Minimal APIs sind deshalb der bevorzugte Ausgangspunkt für AOT, vorausgesetzt, auch die verwendeten Libraries und die JSON-Serialisierung sind AOT-kompatibel.

> Native AOT ist kein pauschales Architekturziel. Für eine langlebige, umfangreiche Business-Anwendung ohne Startzeitdruck kann der Aufwand den Nutzen übersteigen. Die tatsächliche Differenz sollte immer für die eigene Anwendung gemessen werden.

MIDDLEWARE

Pipelines & Middleware

Middleware sind kleine Bausteine um die eigentliche Fachlogik herum. Jeder Request durchläuft sie der Reihe nach; jede Middleware darf vor *und* nach der nächsten arbeiten, oder den Request frühzeitig beenden. Querschnittsthemen wie Sicherheit, Fehlerbehandlung und Protokollierung liegen so **einmal zentral** in der Pipeline statt wiederholt in jedem Controller.



Ein `return` ohne `next()` ist ein **Short Circuit**: nachfolgende Middleware und der Endpoint werden nicht mehr ausgeführt.

Das eignet sich z. B. für statische Dateien, eine abgewiesene Rate-Limit-Anfrage oder eine fehlende API-Key-Prüfung.

Typische Middleware

Middleware	Zentrale Aufgabe
Authentication	Ermittelt die Identität, etwa aus Cookie oder Bearer Token, und setzt `HttpContext.User`.
Authorization	Prüft Policies, Rollen oder Claims und gibt bei fehlender Berechtigung `403 Forbidden` zurück.
CORS	Erlaubt dem Browser gezielt Cross-Origin-Requests, etwa vom Frontend auf die API.
Global Exception Handler	Fängt unbehandelte Exceptions ab, protokolliert sie und liefert eine konsistente Problem-Response statt eines Stack Traces.
Rate Limiter	Schützt API und Ressourcen vor Überlastung oder Missbrauch, meist mit `429 Too Many Requests`.

Authentication beantwortet **„Wer bin ich?“**, Authorization **„Darf ich das?“**. Deshalb kann Authorization niemals vor Authentication stehen. Ein zentraler Exception Handler gehört möglichst an den Anfang, damit er auch Fehler aus späteren Layern erfasst.

PIPELINES · ERROR HANDLING 06 / 09

Exceptions zentral in ProblemDetails übersetzen

Der `GlobalExceptionHandler` ist ein gutes Beispiel für Zentralisierung: Code wirft eine aussagekräftige Exception, aber **genau eine** Stelle entscheidet, wie daraus eine HTTP-Response wird. Controller und Minimal APIs bleiben dadurch frei von wiederholten `try`/`catch`-Blöcken und exponieren keine internen Exception-Meldungen.

`ProblemDetails` ist das etablierte, maschinenlesbare Format für Fehlerantworten zwischen HTTP-Systemen.

status	title	type	instance
HTTP-Statuscode	Kurzbeschreibung	Fehlerkategorie-URI	konkreter Pfad

Mit `Extensions` lassen sich sichere Zusatzinformationen wie eine Trace-ID ergänzen. `AddProblemDetails()` registriert den zentralen Writer, den der Handler verwendet.

> Der Handler ist ein **Singleton**. Er darf deshalb nicht direkt von Scoped Services wie einem `DbContext` abhängen. Erwartbare Fehler werden in passende `ProblemDetails` übersetzt; unbekannte Fehler werden geloggt, aber gegenüber dem Consumer bewusst allgemein gehalten.

PIPELINES · GLOBALEXCEPTIONHANDLER 06 / 09

```
public sealed class GlobalExceptionHandler(
    ILogger<GlobalExceptionHandler> logger,
    IProblemDetailsService problemDetails) : IExceptionHandler
{
    public async ValueTask<bool> TryHandleAsync(
        HttpContext httpContext,
        Exception exception,
        CancellationToken cancellationToken)
    {
        logger.LogError(exception, "Unhandled exception for {Path}",
            httpContext.Request.Path);
        var (status, title, type) = exception switch
        {
            OrderNotFoundException => (
                StatusCodes.Status404NotFound,
                "Order not found",
                "https://api.example.com/problems/order-not-found"),
            BusinessRuleException => (
                StatusCodes.Status422UnprocessableEntity,
                "Business rule violated",
                "https://api.example.com/problems/business-rule"),
            _ => (
                StatusCodes.Status500InternalServerError,
                "An unexpected error occurred",
                "https://api.example.com/problems/unexpected-error")
        };
        httpContext.Response.StatusCode = status;
        await problemDetails.WriteAsync(new ProblemDetailsContext
        {
            HttpContext = httpContext,
            ProblemDetails = new() { Status = status, Title = title,
                Type = type, Instance = httpContext.Request.Path.Value,
                Extensions = { ["traceId"] = httpContext.TraceIdentifier } }
        });
        return true;  // behandelt; keine weitere Response schreiben.
    }
}
```

```
builder.Services.AddProblemDetails();
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();
app.UseExceptionHandler();  // früh in der Pipeline
```

PIPELINES · REIHENFOLGE

Die Reihenfolge ist Teil der Architektur

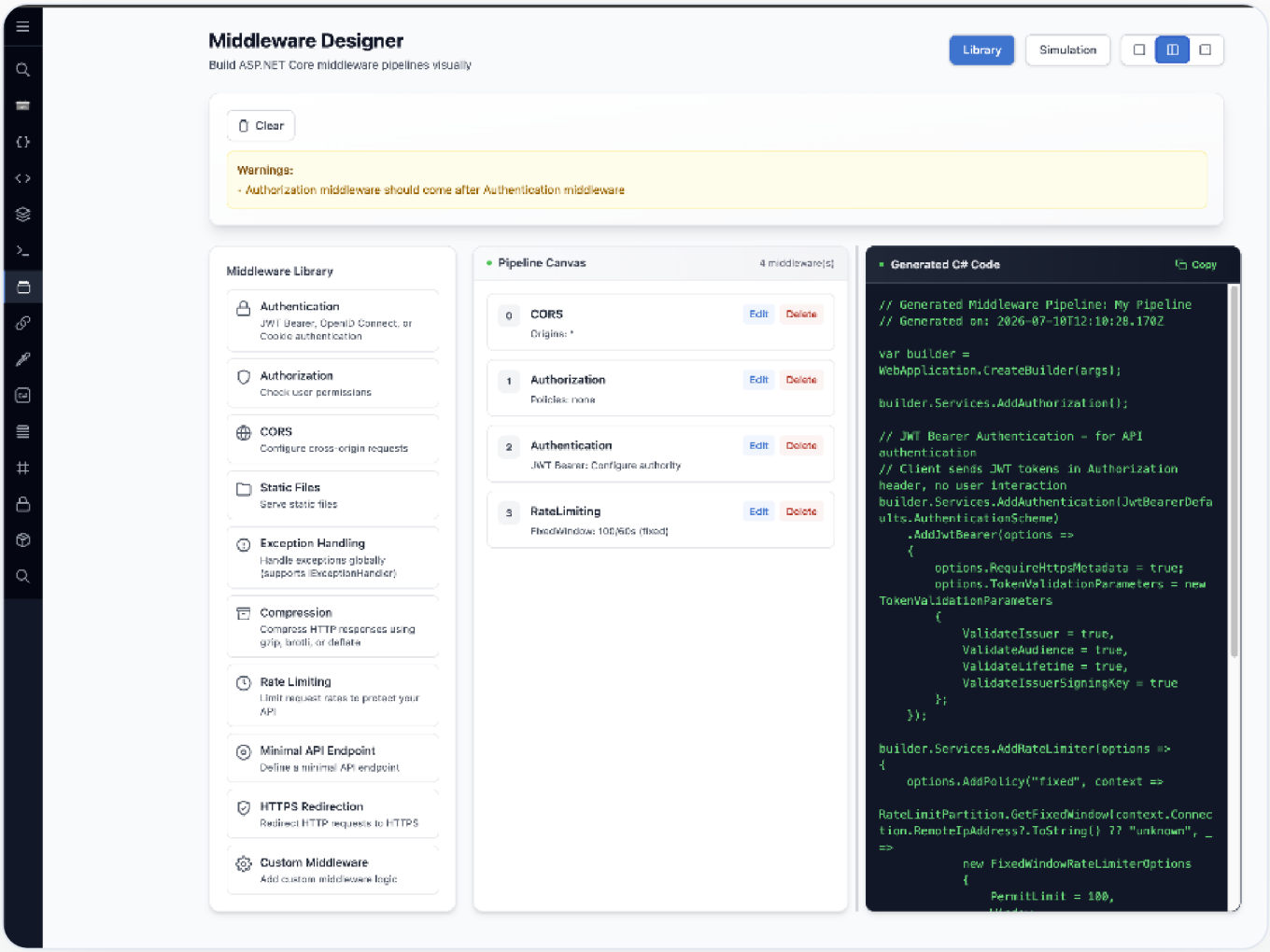
```
builder.Services.AddProblemDetails();
builder.Services.AddExceptionHandler<GlobalExceptionHandler>();
builder.Services.AddCors(options =>
    options.AddPolicy("frontend", policy => policy
        .WithOrigins("https://app.example.com")
        .AllowAnyHeader()
        .AllowAnyMethod()));
builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
builder.Services.AddRateLimiter(options =>
    options.AddFixedWindowLimiter("api", limiter =>
    {
        limiter.PermitLimit = 100;
        limiter.Window = TimeSpan.FromMinutes(1);
    }));
var app = builder.Build();
app.UseExceptionHandler();
app.UseHttpsRedirection();
app.UseRouting();
app.UseCors("frontend");
app.UseAuthentication();
app.UseRateLimiter();  // nach Routing + nach Auth
app.UseAuthorization();
app.MapControllers();
```

Die konkrete Reihenfolge hängt von der Anwendung ab, die Invarianten nicht: **CORS → Authentication → Authorization**. Endpoint-spezifische Rate Limits benötigen `UseRouting`; ein Limiter pro Benutzer braucht zuvor die Authentication.

PIPELINES · WERKZEUG 06 / 09

Interaktiver Middleware Designer

Unser **interaktiver Middleware Designer** macht den Durchlauf sichtbar und warnt bei problematischen Reihenfolgen, etwa Authorization vor Authentication.



→ Selbst ausprobieren: **toolbox.bitspire.ch/middleware-designer**

PIPELINES · RATE LIMITING 06 / 09

Rate Limiting: vier Strategien

Variante	Funktionsweise	Geeignet für
Fixed Window	Feste Zahl Requests je Zeitfenster; zum nächsten Fenster wird zurückgesetzt.	Einfache API-Limits, z. B. 100/Minute.
Sliding Window	Teilt das Fenster in Segmente; abgelaufene Requests werden schrittweise freigegeben.	Gleichmässigere Limits ohne Burst an Fenstergrenzen.
Token Bucket	Ein Bucket enthält regelmässig nachgefüllte Tokens; ein Request verbraucht ein Token.	Kontrollierte Bursts bei begrenzter Rate.
Concurrency	Begrenzt gleichzeitig laufende Requests, nicht Requests pro Zeitraum.	Teure Reports, Exporte oder externe Aufrufe.

Policies gelten global oder gezielt pro Endpoint, nützlich, wenn ein teurer Export strenger limitiert wird als ein normaler Lesezugriff:

```
app.MapGet("/reports/export", ExportReport)
   .RequireRateLimiting("api");
```

Eigene Middleware mit `app.Use`

Für kleine, lokale Regeln genügt `app.Use`. Die Middleware kann den Request weiterreichen oder ihn mit einem `return` selbst beantworten:

```
public static class ApiKeyMiddlewareExtensions
{
    public static IApplicationBuilder UseRequiredApiKey(this IApplicationBuilder app) =>
        app.Use(async (context, next) =>
        {
            if (!context.Request.Headers.ContainsKey("X-Api-Key"))
            {
                context.Response.StatusCode = StatusCodes.Status401Unauthorized;
                await context.Response.WriteAsync("Missing API key");
                return;  // Short Circuit
            }
            await next(context);
        });
}
```

Extension Methods halten `Program.cs` übersichtlich

Registrierungen gehören als sprechende Extension Methods nahe zu ihrem Modul, so bleibt der Composition Root lesbar:

```
builder.Services
    .AddApplication()
    .AddInfrastructure(builder.Configuration);
public static class ApplicationServiceCollectionExtensions
{
    public static IServiceCollection AddApplication(this IServiceCollection services)
    {
        services.AddScoped<IOrderService, OrderService>();
        return services;
    }
}
```

CACHING

Caching: schnell, aber bewusst

Caching ist einer der wirksamsten Beschleuniger einer Web-Anwendung: Eine teure Datenbankabfrage oder ein externer HTTP-Aufruf wird nicht für jeden Request wiederholt. Es ersetzt aber keine gute Abfrage und braucht eine fachliche Entscheidung: **Welche Daten dürfen wie lange veraltet sein, und wer invalidiert sie nach einer Änderung?**

L1 · In-Memory

Sehr schnell, gilt aber nur für eine einzelne App-Instanz.

+

L2 · Distributed

Optional, z. B. Redis. Teilt Daten zwischen Instanzen und überlebt einen Prozessneustart.

`HybridCache` vereint beide Ebenen. Ist kein `IDistributedCache` konfiguriert, bleibt es trotzdem als lokaler Cache nützlich.

```
builder.Services.AddStackExchangeRedisCache(options =>
    options.Configuration = builder.Configuration.GetConnectionString("Redis"));
builder.Services.AddHybridCache(options =>
    options.DefaultEntryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromMinutes(5),
        LocalCacheExpiration = TimeSpan.FromMinutes(1)
    });
```

CACHING · LESEN

Lesen mit `GetOrCreateAsync`

Wir cachen dabei ein **DTO**, nicht eine von EF Core getrackte Entity:

```
public sealed class ProductQuery(HybridCache cache, AppDbContext db)
{
    public ValueTask<ProductSummary> GetExistingAsync(
        Guid id, CancellationToken cancellationToken) =>
        cache.GetOrCreateAsync(
            $"products:{id}",
            async ct => await db.Products
                .Where(product => product.Id == id)
                .Select(product => new ProductSummary(
                    product.Id, product.Name, product.Price))
                .SingleAsync(ct),
            cancellationToken: cancellationToken);
}
```

> `GetOrCreateAsync` beschreibt den ganzen Ablauf: erst L1, dann gegebenenfalls L2 und erst bei einem Cache Miss die tatsächliche Datenquelle.

CACHING · RESILIENZ

Stampede Protection

Läuft ein beliebter Cache-Eintrag ab, können sonst hunderte gleichzeitige Requests dieselbe teure Abfrage starten, ein **Cache Stampede**. `HybridCache` sorgt dafür, dass für einen Key nur ein gleichzeitiger Aufruf die Factory ausführt; die übrigen warten auf dessen Resultat.

Nach einer Änderung müssen betroffene Einträge explizit entfernt oder über Tags invalidiert werden. Ein Cache-Key gehört daher in dieselbe fachliche Verantwortung wie der Schreibvorgang, nicht als verstreute String-Konstante in irgendwelche Controller. Eine ausführlichere Einführung bietet unser Artikel **MemoryCache, DistributedCache und HybridCache**.

WARUM WIR FUSIONCACHE MÖGEN

Wir bei BitSpire sind grosse Fans von **FusionCache**. Das Projekt ist nicht von Microsoft, aber eine praxiserprobte Referenz für das Hybrid-Cache-Pattern und kann zugleich als Implementierung der `HybridCache`-Abstraktion eingesetzt werden.

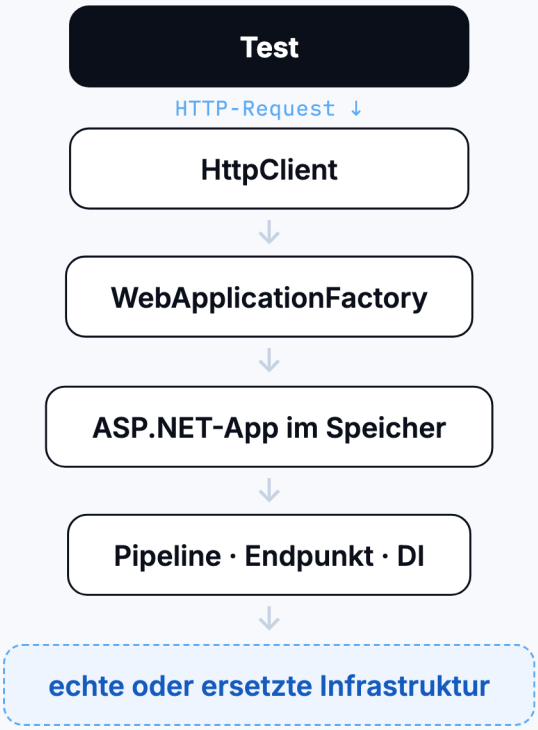
- Verteilte Stampede Protection
- Backplane
- Fail-Safe
- Timeouts
- Eager Refresh

TESTING

Testing

Für Web-APIs sind Tests am wertvollsten, wenn sie sich wie ein **echter Consumer** verhalten: Sie senden HTTP-Requests und prüfen die sichtbare Antwort, nicht interne Methoden oder Domain-Objekte.

`WebApplicationFactory<Program>` startet die ASP.NET-Anwendung dafür im Speicher. Pipeline, Dependency Injection, Konfiguration sowie Serialisierung laufen mit, nur die Infrastruktur kann gezielt ersetzt werden.



↑ `HTTP-Response` nimmt denselben Weg zurück zum Test

Das bringt drei wesentliche Vorteile:

01 **Echter Consumer.** Der Test verwendet nur den öffentlichen HTTP-Vertrag. Ändern sich Route, JSON-Feld oder Statuscode, fällt der Test wie bei einem echten API-Consumer fehl.

02 **Grosse Aussagekraft.** Ein Test prüft den ganzen Weg: Routing, Middleware, Binding, Validierung, DI und Serialisierung, und wird zur lebendigen Dokumentation eines Features.

03 **Richtiges TDD.** Zuerst Request und erwartete Response formulieren, *bevor* Produktionscode existiert. Der Test ist zunächst rot; erst dann wird genau genug implementiert, um ihn grün zu machen.

```
[Fact]
public async Task Create_order_returns_201()
{
    using var client = _factory.CreateClient();
    var response = await client.PostAsJsonAsync("/orders", new { sc-camel-product-id
= "keyboard" });
    Assert.Equal(HttpStatusCode.Created, response.StatusCode);
}
```

Der Test kennt bewusst kein internes `CreateOrderCommand`. Er bleibt damit ein zuverlässiger Vertragstest, auch wenn die Implementierung intern refaktoriert wird.

> Für Abhängigkeiten gibt es zwei Stufen: **Fakes** für fremde oder langsame Systeme und **echte, kurzlebige Infrastruktur** für die Datenbank. Mit **Testcontainers** startet der Test z. B. einen isolierten SQL-Server-Container und deckt Abfragen, Migrationen und Provider-Unterschiede realistisch ab, ganz ohne „Mocking“ des Datenbankverhaltens.

ZUM SCHLUSS

Lust auf mehr?

Moderne Web-Anwendungen haben noch viele weitere Facetten: Background Services, resiliente Kommunikation mit externen Systemen, saubere HTTP-Clients, Observability und mehr.

Jedes Projekt ist individuell, und genau deshalb lohnt sich ein Gespräch über die passende Architektur statt einer Lösung von der Stange.

Ihr Projekt mit uns besprechen?

Schreiben Sie uns, wir freuen uns auf den Austausch.

contact@bitspire.ch
bitspire.ch