

# WEBSERVICES MIT ASP.NET



- Sourcen mit Beispielen zum Skript finden sie unter [florianwachs/AspNetWebservicesCourse \(github.com\)](https://github.com/florianwachs/AspNetWebservicesCourse)

# ASP.NET UND .NET

- Offizielle Begrifflichkeiten
  - .NET (9)
  - ASP.NET
  - dotnet-cli

# ASP.NET UND .NET VERSIONEN

- Aktuell (3.2025):
  - .NET 9 STS, Net 10 ist in Preview
- November 2025
  - .NET 10 LTS

# WARUM ASP.NET?

# WARUM ASP.NET ?

- Kompletter Rewrite von ASP.NET (über 15 Jahre gewachsener Code)
- Cross-Plattform durch .NET (Core)
- Modular\*
- Open Source
- Wechsel vom „All-Inclusive“-Ansatz zu „Pick-What-You-Need“
- Zusammenführung von MVC und Web API was die Implementierung angeht
- Auflösen von Abhängigkeiten zu Windows-Komponenten wie den Internet Information Services
- Performance as a Feature

# ASP.NET

- Ab hier reden wir bei ASP.NET immer von der modernen Reimplementierung, nicht der Fullframework (4.8) Version

# ASP.NET EXTENSIBILITY

- Die Defaults verwenden (Implementiert über NuGet-Pakete)
- Von den bestehenden Klassen ableiten und Funktionalität anpassen
- Komplette Eigenimplementierung von Interfaces / abstrakten Klassen für maximale Anpassung



# ASP.NET INSTALLIEREN

- <https://dot.net>

Free. Cross-platform. Open source.

## Download .NET

For macOS

### .NET 9.0

Standard Term Support Recommended

[Download .NET SDK Arm64 \(Apple Silicon\)](#)

Version 9.0.3, released March 18, 2025

[All .NET 9.0 downloads](#) [.NET 9.0 runtime](#) [All .NET versions](#)

### .NET 8.0

Long Term Support

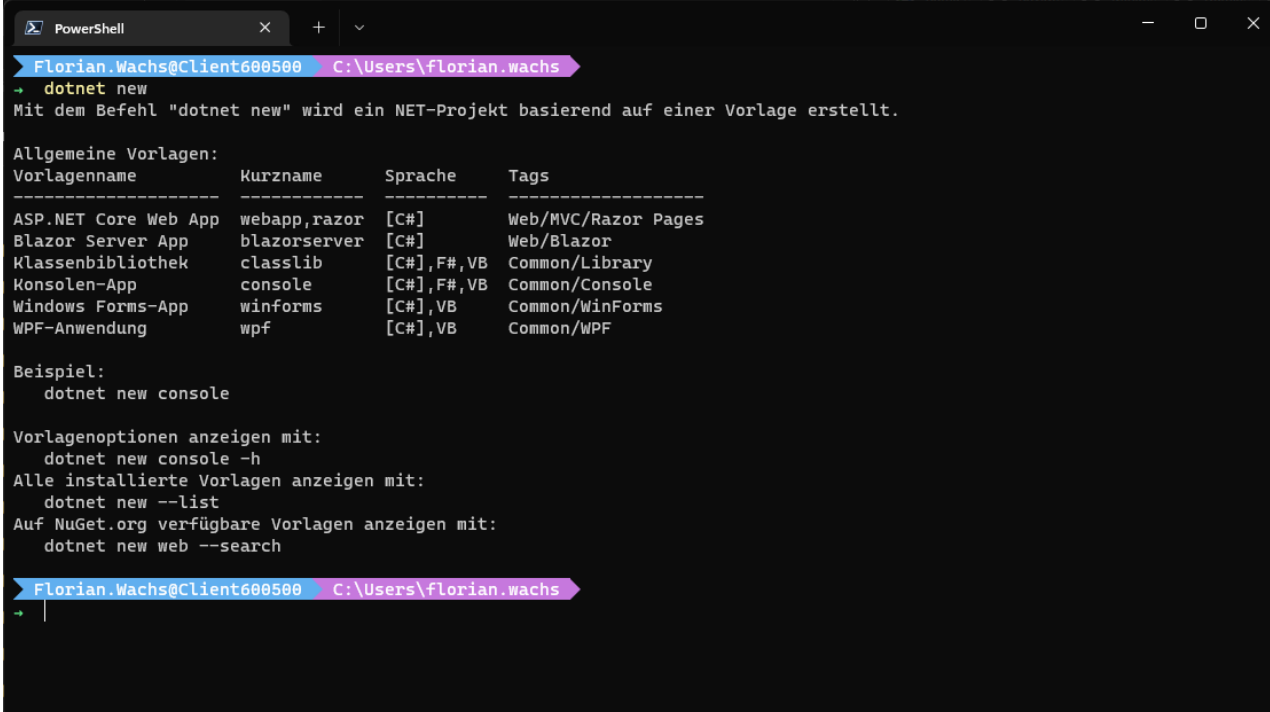
[Download .NET SDK Arm64 \(Apple Silicon\)](#)

Version 8.0.14, released March 11, 2025

[All .NET 8.0 downloads](#)

# DOTNET CLI

- dotnet new
  - Neues Projekt durch ein Template anlegen
- dotnet restore
  - .NET Pakete wiederherstellen
- dotnet package add
  - Neues NuGet-Paket hinzufügen
- dotnet run
  - Code ausführen
- dotnet test
  - Alle Unittests laufen lassen



```
PowerShell
Florian.Wachs@Client600500 C:\Users\florian.wachs
> dotnet new
Mit dem Befehl "dotnet new" wird ein NET-Projekt basierend auf einer Vorlage erstellt.

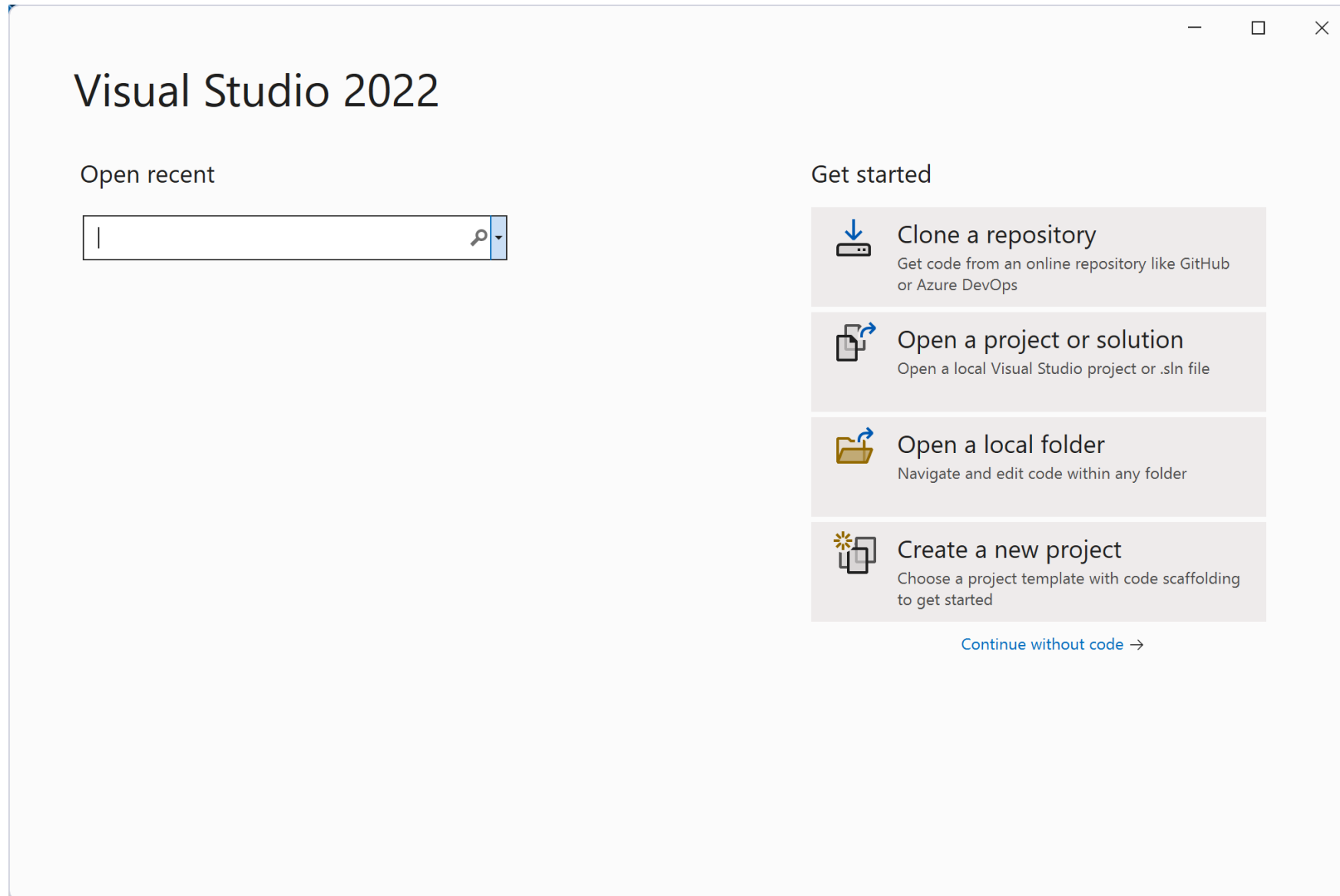
Allgemeine Vorlagen:
Vorlagenname      Kurzname      Sprache      Tags
-----
ASP.NET Core Web App webapp, razor [C#]         Web/MVC/Razor Pages
Blazor Server App  blazorserver [C#]         Web/Blazor
Klassenbibliothek  classlib      [C#],F#,VB   Common/Library
Konsolen-App       console       [C#],F#,VB   Common/Console
Windows Forms-App winforms      [C#],VB      Common/WinForms
WPF-Anwendung      wpf           [C#],VB      Common/WPF

Beispiel:
dotnet new console

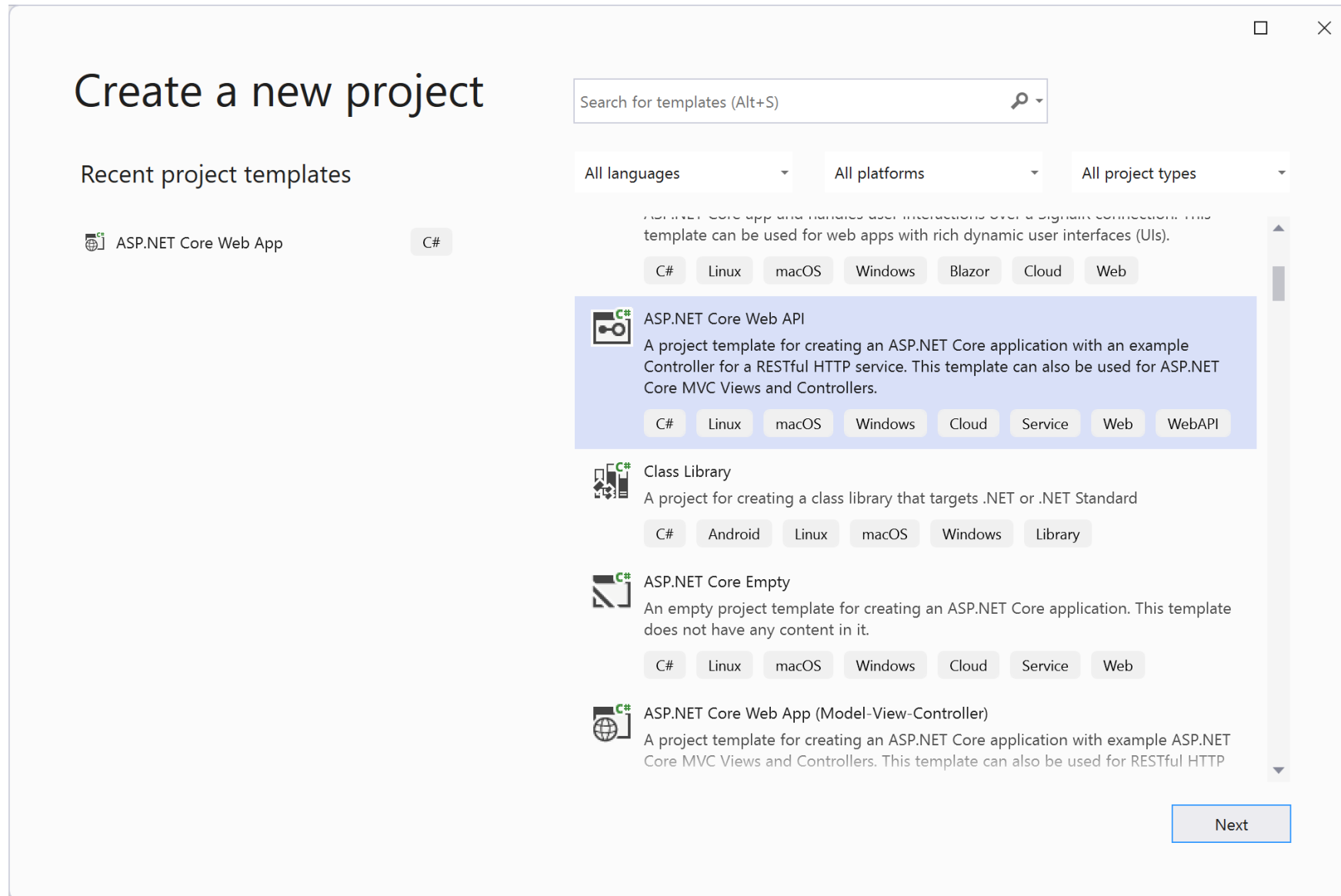
Vorlagenoptionen anzeigen mit:
dotnet new console -h
Alle installierte Vorlagen anzeigen mit:
dotnet new --list
Auf NuGet.org verfügbare Vorlagen anzeigen mit:
dotnet new web --search

Florian.Wachs@Client600500 C:\Users\florian.wachs
> |
```

# VISUAL STUDIO 2022



# VISUAL STUDIO 2022



# VISUAL STUDIO 2022

□ ×

## Configure your new project

ASP.NET Core Web App C# Linux macOS Windows Cloud Service Web

Project name

Location

...

Solution

Solution name ⓘ

☐ Place solution and project in the same directory

Back Next

# VISUAL STUDIO 2022

□ ×

## Additional information

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web WebAPI

Framework ⓘ

.NET 6.0 (Long-term support) ▾

Authentication type ⓘ

None ▾

☒ Configure for HTTPS ⓘ

☐ Enable Docker ⓘ

Docker OS ⓘ

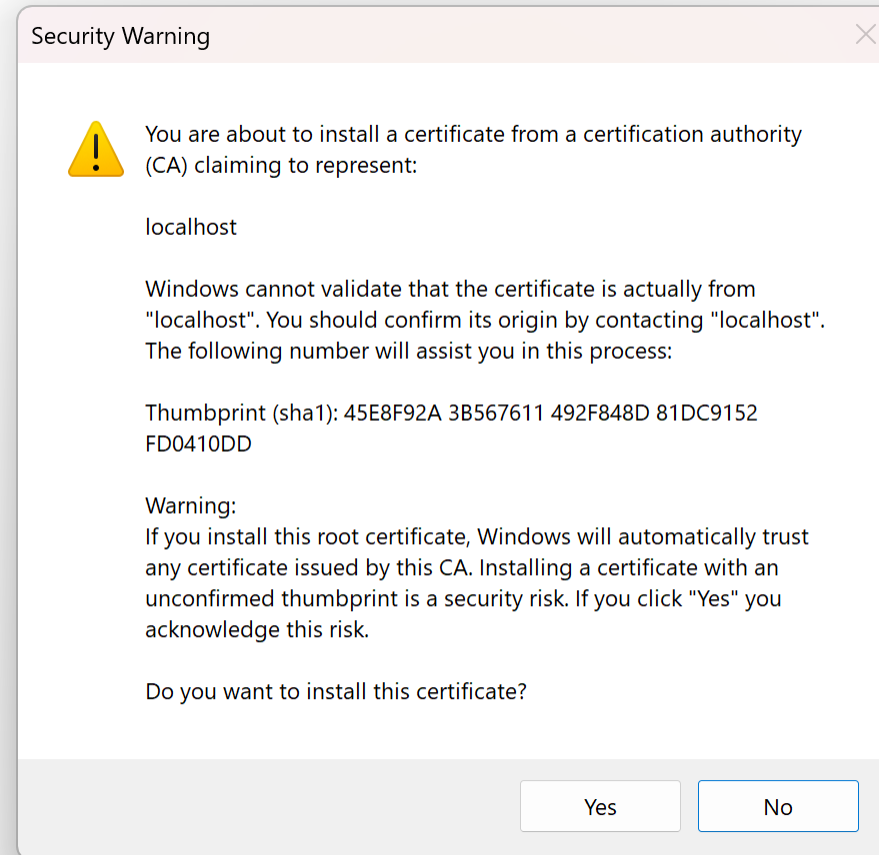
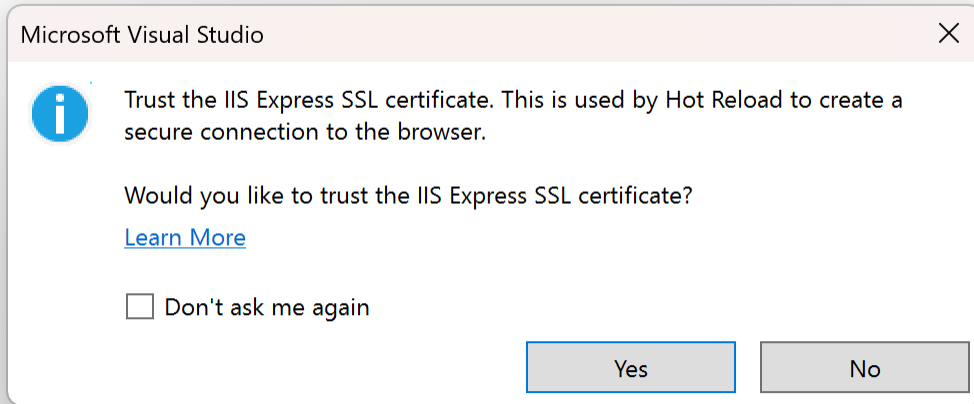
Linux ▾

☐ Use controllers (unchecked to use minimal APIs) ⓘ

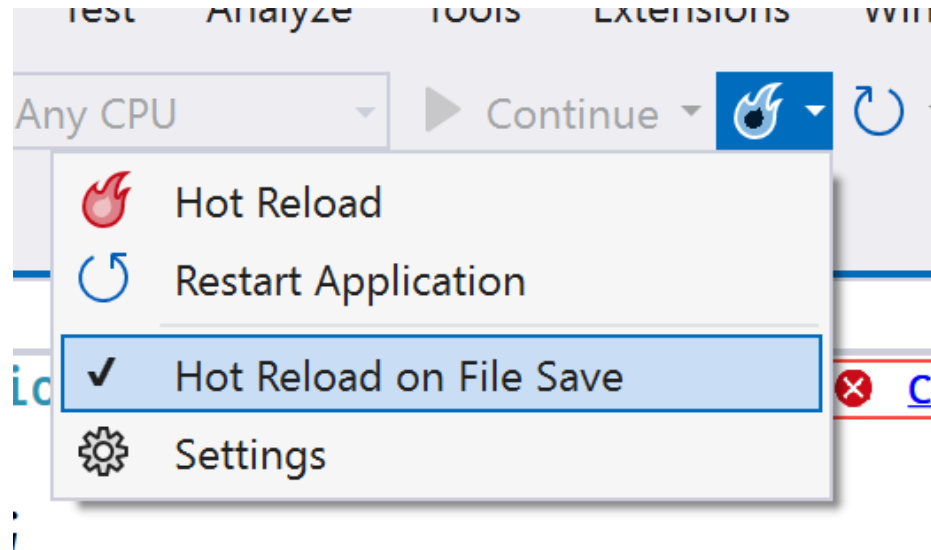
☒ Enable OpenAPI support ⓘ

Back Create

# ASP.NET DEV HTTPS ZERTIFIKAT



# .NET HOT RELOAD





# ASP.NET SDK

## SDK

<https://docs.microsoft.com/de-de/dotnet/core/tools/csproj#sdk-attribute>

Dient als „Starting Point“ für die Anwendung. Erlaubt die implizierte Einbindung zusätzlicher Pakete, Buildprozess-Modifikationen und vielen zusätzlichen Voreinstellungen

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

```
<PropertyGroup>
```

```
<TargetFramework>net6.0</TargetFramework>
```

```
<Nullable>enable</Nullable>
```

```
<ImplicitUsings>enable</ImplicitUsings>
```

```
</PropertyGroup>
```

```
<ItemGroup>
```

```
<PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
```

```
</ItemGroup>
```

```
</Project>
```

## TargetFramework

Steuert implizit welche Metapakete eingebunden werden

## Nuget-Packages

Durch Einbindung zusätzlicher Pakete kann die Funktionalität unserer Anwendung erweitert werden

# ASP.NET SDK

- Spezielles SDK das auf das „Share-Framework“ verweist
- Updates über .NET SDK / Runtime-Installer
- Zusätzliche Features wie Authentifizierung, GraphQL, gRPC, Entity Framework usw. werden über NuGet-Pakete eingebunden werden

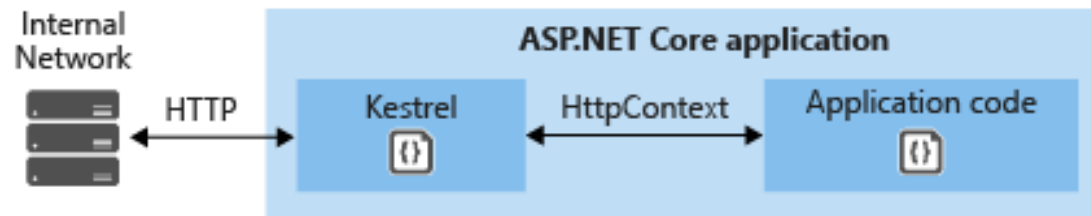
# **WAS STARTET, KONFIGURIERT UND REGELT DEN LEBENSZYKLUS UNSERES WEBSERVICES?**

# HOSTING

# HOSTING VON ASP.NET

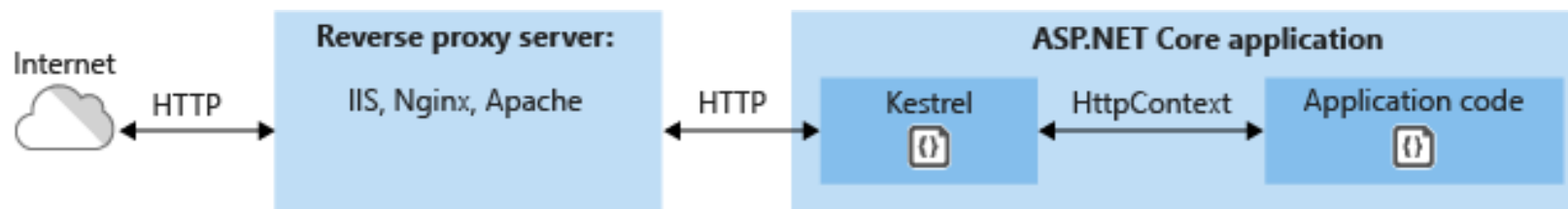
## Kestrel

Kestrel ist ein performanter Web Server, der auf allen Plattformen verfügbar ist auf denen .NET Core läuft



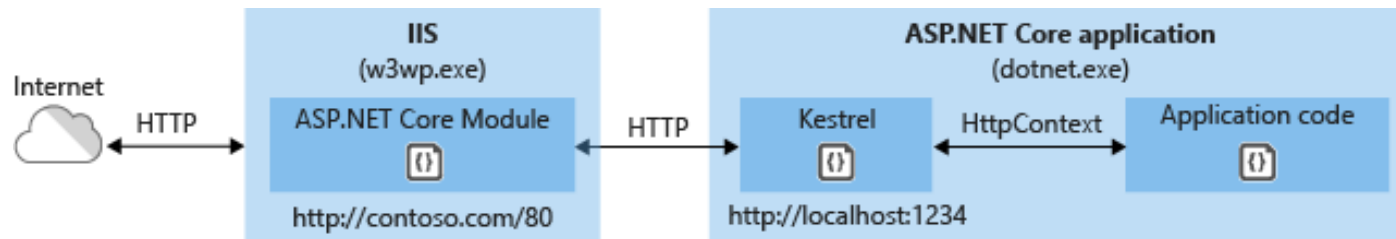
## Kestrel ohne Reverse Proxy

Seit .net core 3+ kann Kestrel auch ohne Reverse Proxy wie NGNIX oder IIS betrieben werden



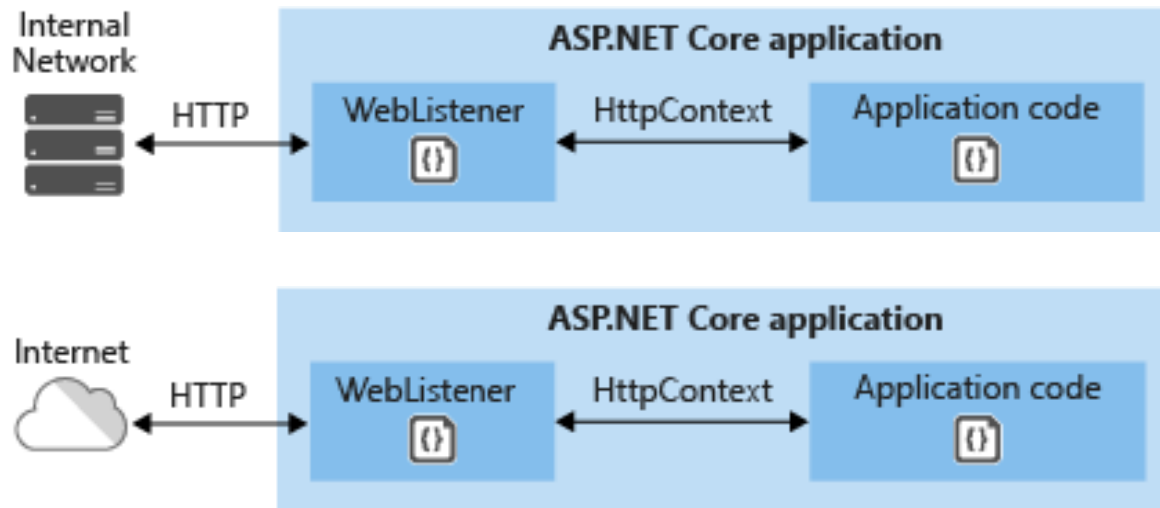
# HOSTING VON ASP.NET

ASP.NET Module (ANCM) ist ein natives IIS Module welches den Traffic an die ASP.NET Applikation weitergibt und auch wieder zurück



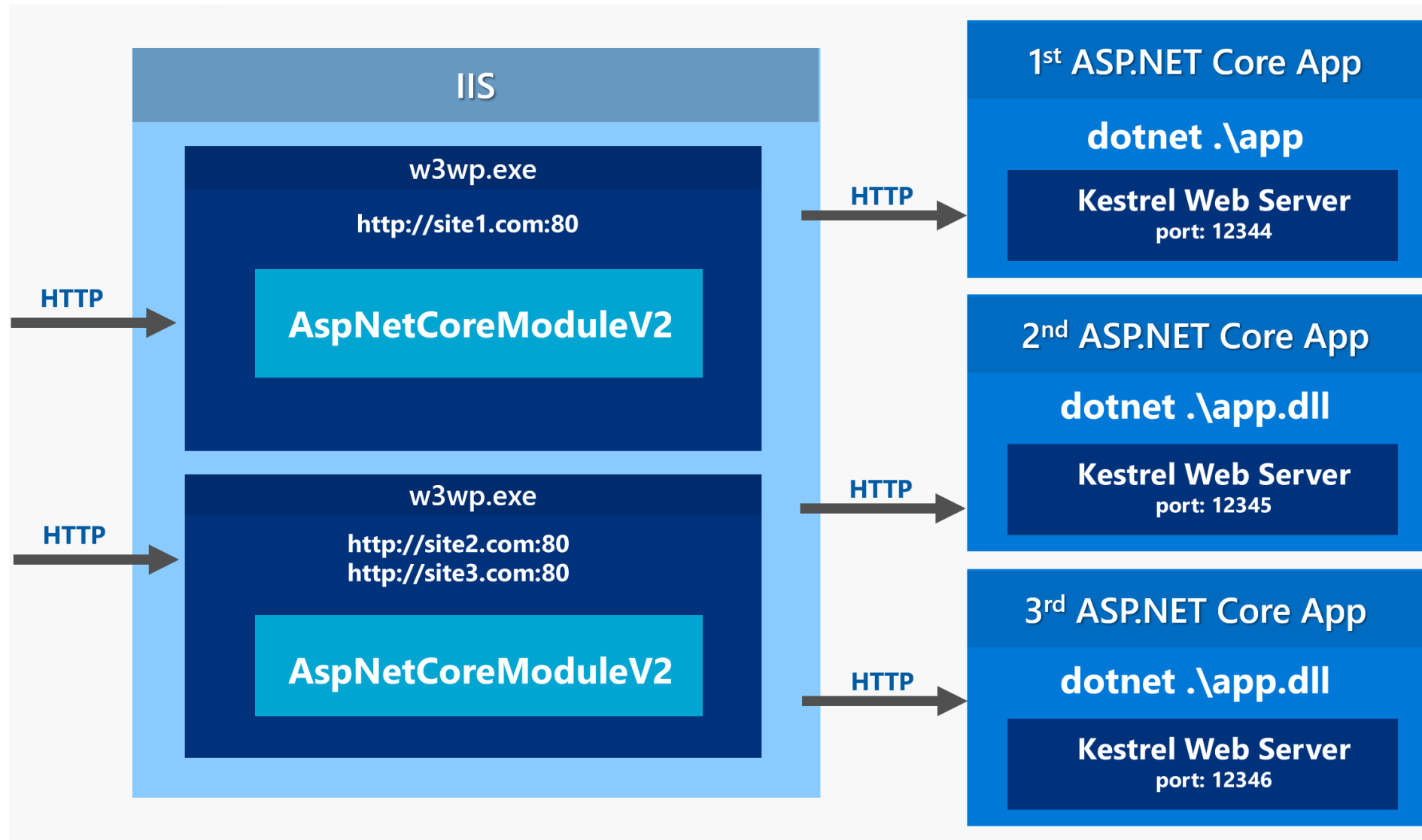
# HOSTING VON ASP.NET

WebListener ist ein Web Server der direkt auf dem Http.Sys kernel mode driver von Windows aufsetzt. Kann als Alternative zu Kestrel gesehen werden, läuft aber nur auf Windows > 7 und Windows Server > 2008 R2



Ist eine gute Lösung wenn keine Features vom IIS benötigt werden

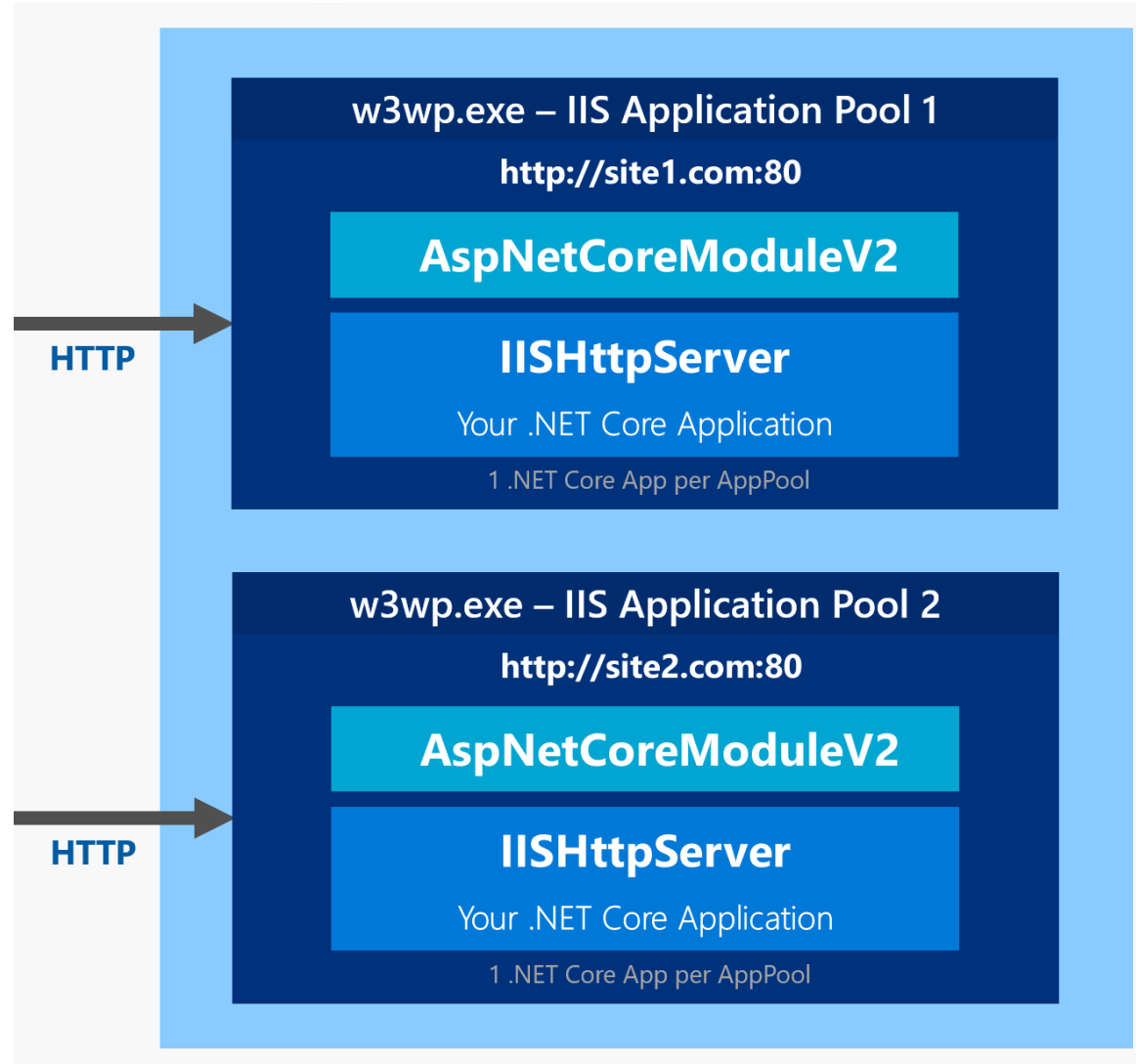
# ASP.NET OUT- VS IN-PROCESS HOSTING



<https://weblog.west-wind.com/posts/2019/Mar/16/ASPNET-Core-Hosting-on-IIS-with-ASPNET-Core-22>



# ASP.NET OUT- VS IN-PROCESS HOSTING



<https://weblog.west-wind.com/posts/2019/Mar/16/ASPNET-Core-Hosting-on-IIS-with-ASPNET-Core-22>

## ASP.NET IN-PROCESS HOSTING VORTEILE

- Kann per Konfiguration aktiviert / deaktiviert werden
- Performance: Deutlich höherer Durchsatz (aktuell bis zu 2x)
- Verwendet nicht Kestrel sondern eine IISHttpServer-Implementierung, welche native IIS-Objekte nutzt

# KESTREL

# KESTREL

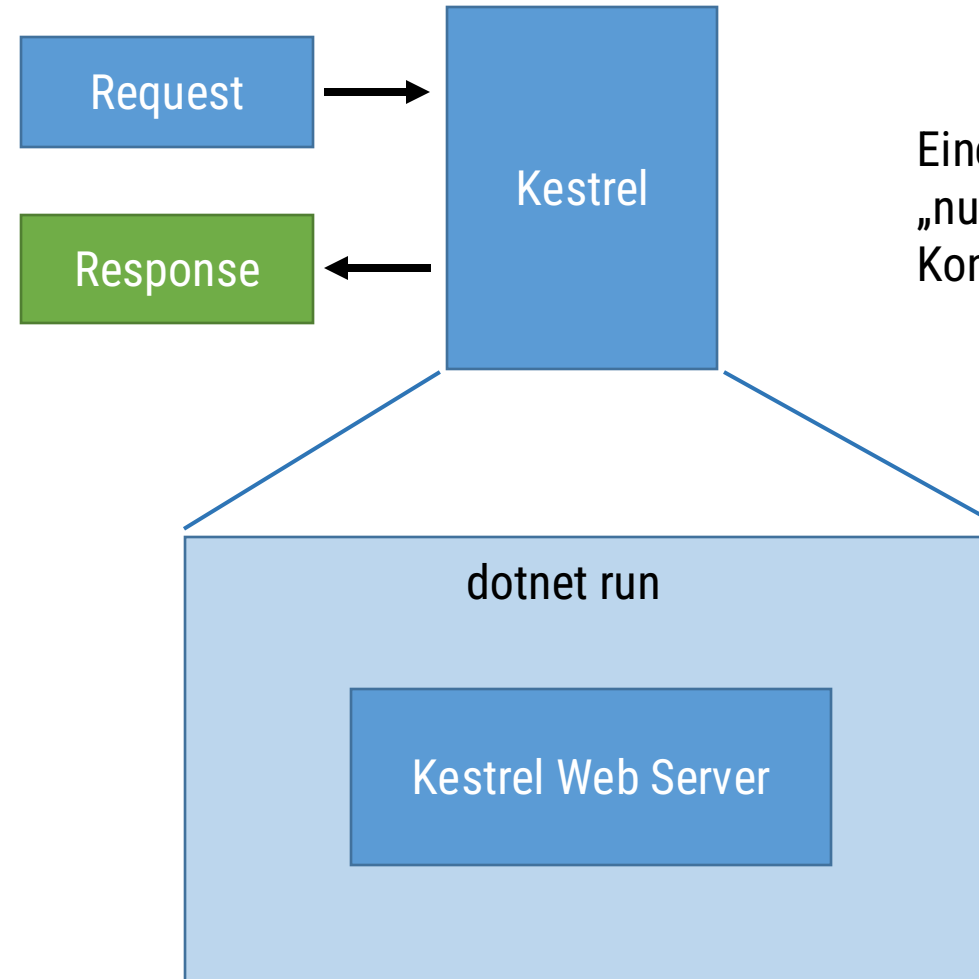
- Ab 3.0 ist Kestrel für die Verwendung ohne Reverse Proxy freigegeben.
- Implementiert in .NET
- Hochperformant
- Aktuell (2024-03) gibt es einen auf Kestrel basierender Reverse Proxy (YARP) welcher intern bei Microsoft für Hochlast-Services wie Azure AD Gateway verwendet wird

# WIE WIRD DAS ASP.NET HOSTING KONFIGURIERT?

# HOST

- In Asp.Net ist “Host” ein abstraktes Konzept um .NET Code bereitzustellen
- Ein Host stellt grundlegende Infrastruktur bereit wie
  - Logging
  - Dependency Injection
  - Configuration
  - Lifecycle Management
- Ein Host kann z.B. ein Windows Service, Linux System Daemon, WebHost sein
- Asp.Net nutzt einen WebHost

# HOST



Eine ASP.NET Anwendung ist im Prinzip „nur“ eine vom Kestrel-Server verwaltete Konsolenanwendung

# HOST

In .NET 6 wurde die Konfiguration in eine einzige Datei **Program.cs** zusammengeführt.

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

Die **CreateBuilder()**-Methode erzeugt ein Konfigurationsobjekt (Builder-Pattern) mit dessen Eigenschaften das Hosting und die gesamte Applikation konfiguriert werden kann.

Es werden viele sinnvolle Annahmen getroffen, z.B. das Konfigurationseinstellungen direkt aus den Umgebungsvariablen und einer **appsettings.json** Datei geladen werden.



# HOST

Wurden die gewünschten Konfigurationseinstellungen getroffen kann mit **Build()** ein Applikationsobjekt erstellt werden.

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

Mit **Run() oder RunAsync()** wird der Webservice gestartet.

```
app.Run();
```

# HOST

Für eine realistische Applikation sind einige Schritte essentiell.

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

```
// Konfiguration von  
// - Services über Dependency-Injection  
// - Logging  
// - Konfiguration des Hosts (IIS, Http.sys, Kestrel, ...)
```

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

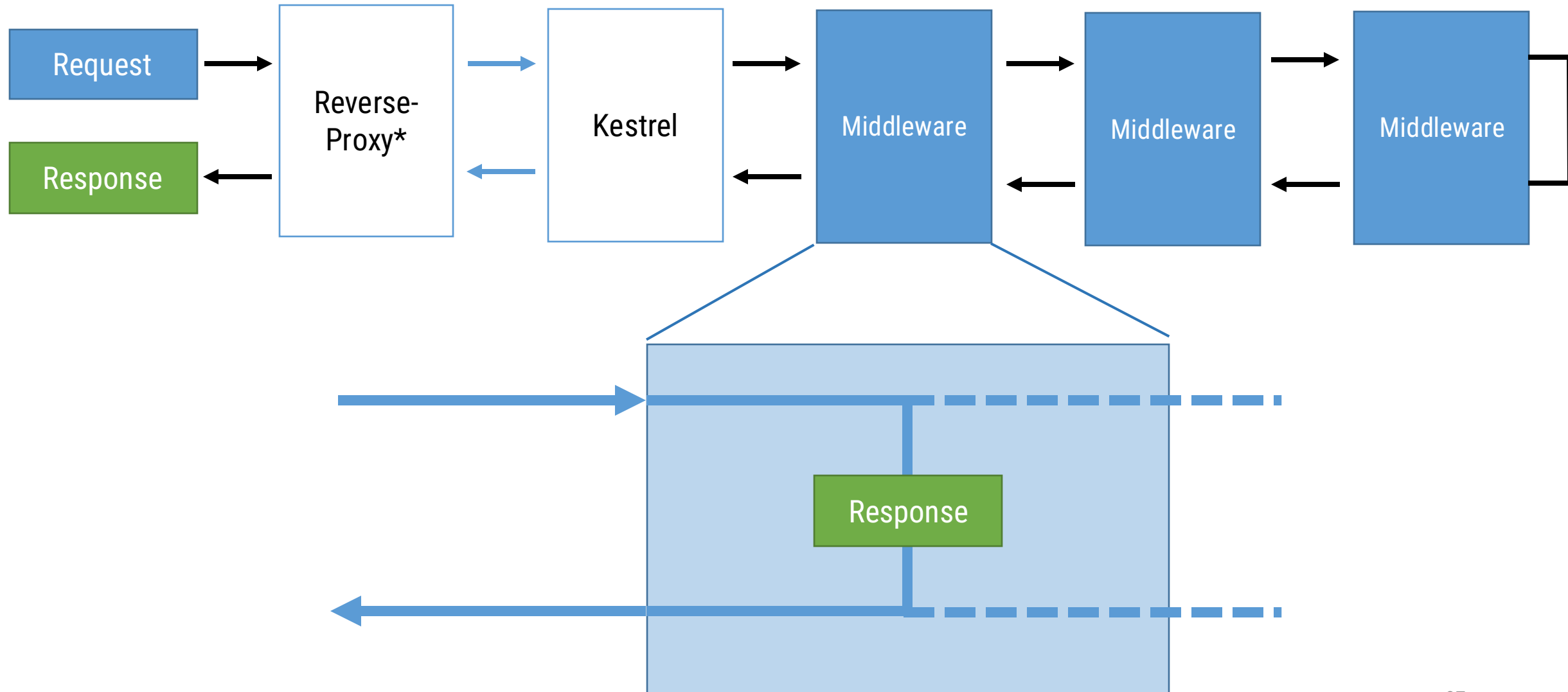
```
// Konfiguration der Middleware Pipeline  
// - (Autorisierung, Authentifizierung, gRPC,...)  
// - Konfiguration der Routes
```

```
app.Run();
```

**WIE KOMMEN WIR JETZT ZUR EIGENTLICHEN  
FUNKTIONALITÄT?**

# MIDDLEWARE

# WAS IST MIDDLEWARE?



# WAS IST MIDDLEWARE?

- Middleware stellt eine Pipeline dar, durch die ein Request weitergereicht wird
- Eine Middleware-Komponente kann
  - Die Pipeline unterbrechen und eine Response erzeugen (Content Generating Middleware / Short Circuit)
  - Den Request bearbeiten / erweitern und weiter durch die Pipeline leiten (Request Editing Middleware)
  - Die Response bearbeiten (Response Editing Middleware) und in der Pipeline weiterreichen
- Viele ASP.NET Features sind selbst als Middleware implementiert

# WAS IST MIDDLEWARE?

- Viele ASP.NET Features sind selbst als Middleware realisiert
  - Logging
  - Authentifizierung / Autorisierung
  - Routing
  - MVC
  - Fehlerbehandlung
  - Swagger / OpenId

# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?



# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

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

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

```
// Konfiguration der Middleware Pipeline  
// - (Autorisierung, Authentifizierung, gRPC,...)  
// - Konfiguration der Routes
```

```
app.Run();
```

# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

- `Run(async context=>...)`
  - Short-Circuit-Middleware, ruft nicht die nächste Middleware in der Pipeline auf
- `Use(async (context,next)=>...)`
  - Ermöglicht es Aktionen vor und nachdem ein Request durch die Middleware gelaufen ist auszuführen
- `Map(string pathSegment, IApplicationBuilder app)`
  - Erzeugt eine Verzweigung (branch) in der Pipeline Aufgrund des angegebenen Pfadsegmentes in der URL
  - Diese alternative Pipeline kann ebenfalls beliebig konfiguriert werden
- `MapWhen`
  - Wie Map aber mit Bedingung (Condition)

# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```

# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

// Wrapping Middleware

```
app.Use(async (context, next) =>
{
    Stopwatch watch = Stopwatch.StartNew();
    await next();
    watch.Stop();
    Console.WriteLine($"Processing Duration: {watch.ElapsedMilliseconds} ms");
});
```

// Branching Middleware

```
app.Map("/jokes", jokesPipelineBranch =>
{
    jokesPipelineBranch.Run(async context =>
    {
        await context.Response.WriteAsync("This is funny....");
    });
});
```

// Short-Circuit Middleware

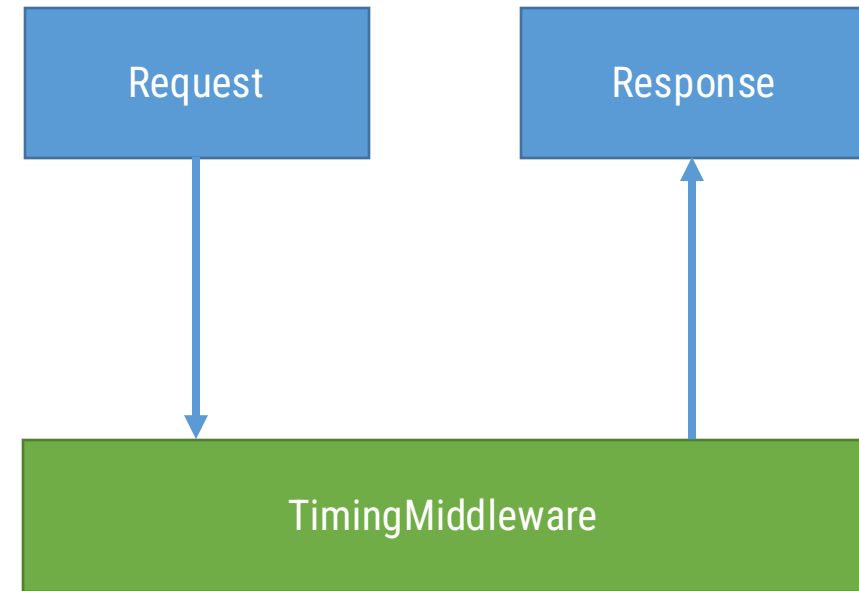
```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```

# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

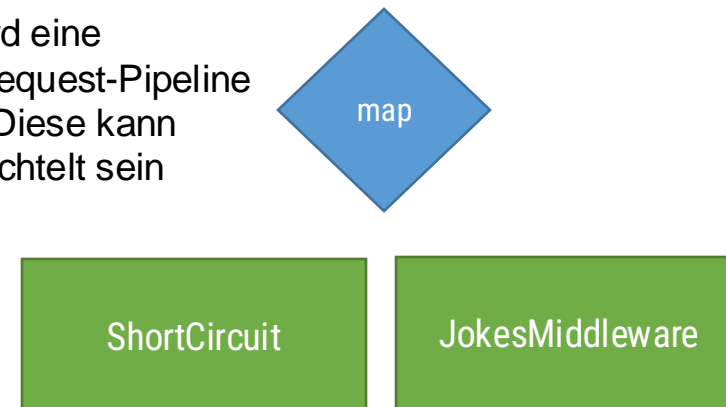
```
// Wrapping Middleware
app.Use(async (context, next) =>
{
    Stopwatch watch = Stopwatch.StartNew();
    await next();
    watch.Stop();
    Console.WriteLine($"Processing Duration: {watch.ElapsedMilliseconds} ms");
});

// Branching Middleware
app.Map("/jokes", jokesPipelineBranch =>
{
    jokesPipelineBranch.Run(async context =>
    {
        await context.Response.WriteAsync("This is funny...");
    });
});

// Short-Circuit Middleware
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```



Mit **Map()** wird eine alternative Request-Pipeline konfiguriert. Diese kann auch geschachtelt sein

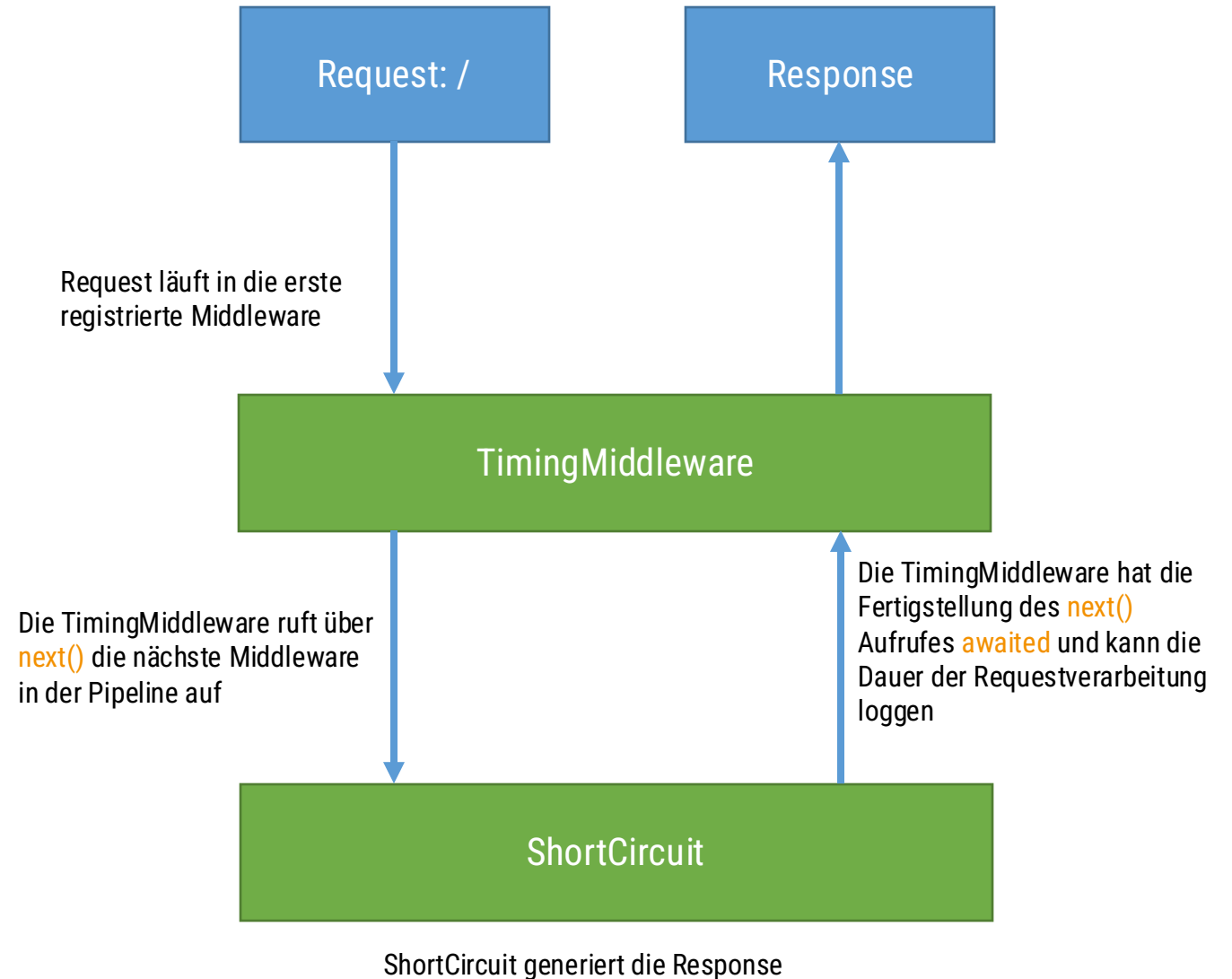


# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

```
// Wrapping Middleware
app.Use(async (context, next) =>
{
    Stopwatch watch = Stopwatch.StartNew();
    await next();
    watch.Stop();
    Console.WriteLine($"Processing Duration: {watch.ElapsedMilliseconds} ms");
});
```

```
// Branching Middleware
app.Map("/jokes", jokesPipelineBranch =>
{
    jokesPipelineBranch.Run(async context =>
    {
        await context.Response.WriteAsync("This is funny....");
    });
});
```

```
// Short-Circuit Middleware
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```

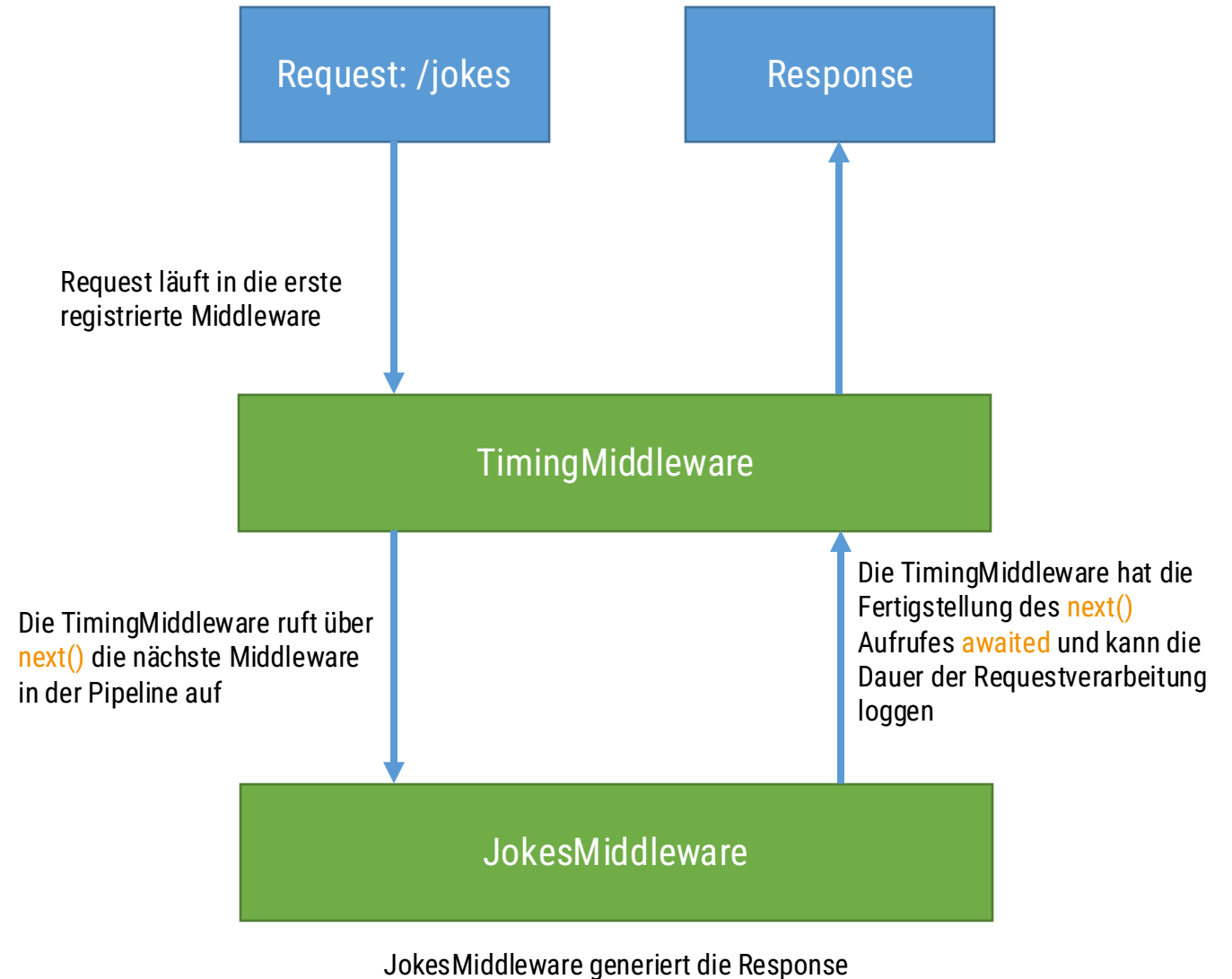


# WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

```
// Wrapping Middleware
app.Use(async (context, next) =>
{
    Stopwatch watch = Stopwatch.StartNew();
    await next();
    watch.Stop();
    Console.WriteLine($"Processing Duration: {watch.ElapsedMilliseconds} ms");
});
```

```
// Branching Middleware
app.Map("/jokes", jokesPipelineBranch =>
{
    jokesPipelineBranch.Run(async context =>
    {
        await context.Response.WriteAsync("This is funny....");
    });
});
```

```
// Short-Circuit Middleware
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```



# MIDDLEWARE ALS KLASSE

Komplexere Middleware sollte in einer eigenen Klasse ausgelagert werden

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

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

    // Die Invoke-Methode muss vorhanden sein und kann auch DI-Services anfordern,
    // indem man sie in die Parameterliste aufnimmt
    public async Task Invoke(HttpContext context)
    {
        Stopwatch watch = Stopwatch.StartNew();
        await _next(context);
        watch.Stop();
        Console.WriteLine($"Processing Duration: {watch.ElapsedMilliseconds} ms");
    }
}
```

Im Konstruktor können weitere Argumente angegeben werden, welche über das DI-System bezogen werden. Achtung: Services im Konstruktor sind Singletons, da Middleware-Komponenten nur einmalig pro Applikation erzeugt werden

DI-Services in der Invoke-Methode sind Per-Request-Dependencies



# MIDDLEWARE ALS KLASSE

```
public static class TimingMiddlewareExtensions
{
    public static IApplicationBuilder UseTimingMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<TimingMiddleware>();
    }
}
```

```
// ...
```

```
app.UseTimingMiddleware();
```

```
// ...
```

# KOMPLEXE MIDDLEWARE KANN ANDERE DIENSTE BENÖTIGEN

- Bisher haben wir “nur” Middleware kennengelernt, welche ihre Funktionalität eigenständig bereitstellen kann, ohne von anderen Services / Klassen abhängig zu sein
- Komplexere Middleware benötigt meist weitere Komponenten um funktionsfähig zu sein. Um eine hohe Flexibilität zu gewährleisten, können sich Middlewares des im ASP.NET integrierten Dependency Injection Systems bedienen

# KOMPLEXE MIDDLEWARE KANN ANDERE DIENSTE BENÖTIGEN

- Per Konvention bieten Middlewares meist zwei Extension-Methods
  - Use[Middleware]: Hängt die Middleware in die Request-Pipeline ein
  - Add[Middleware]: Registriert die benötigten Dienste im Dependency Injection (DI) System.

# HTTP Status Codes

- 200: Alles OK
- 201: Resource created
  - Location Header enthält die URI zur neuen Ressource
- 400: Bad Request
- 401: Unauthorized
  - Sollte dem Aufrufer die Art der geforderten Authentifizierung mitteilen
- 403: Access denied
  - Authentifiziert aber nicht autorisiert, um auf die Ressource zuzugreifen
- 404: Resource not found
- 500: Server error
- <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

# MINIMAL APIS

# MINIMAL APIS

- Eine API mit Run und Use aufzubauen ist performant (Microservices) aber mühevoll
- Das modulare ASP.NET bietet eine spezielle Middleware-Komponente, welche mittels string-Templates und Http-Methoden eine weit einfachere Möglichkeit bietet, API-Endpunkte aufzubauen, ohne jedoch das MVC-Framework (mehr dazu gleich) einbinden zu müssen.

# WIE KÖNNEN ROUTEN REGISTRIERT WERDEN?

# REST| Basics

- Representational State Transfer
- HTTP-Verben haben spezifische Bedeutung
  - GET: Read
  - POST: Create
  - PUT: Update (manchmal auch Neuanlage)
  - DELETE: Delete
  - PATCH: Teil-Update
- URI's repräsentieren eine Ressource („Nouns over Verbs“)
  - REST: myService.com/students/1/courses
  - RPC: myService.com/GetCoursesForStudent?studentId=1



# REST| Basics

- Zustandslose Client/Server Kommunikation
- Identifizierbare Ressourcen
- Unterschiedliche Ressourcen-Repräsentationen (Mime-Typen)
- Hypermedia (Verwendung von Links)
- Zustandsübergänge durch Links
- Entspricht den Kernprinzipien des WWW-Protokolls HTTP

# ROUTING

```
var builder = WebApplication.CreateBuilder(args);  
  
var app = builder.Build();  
  
app.MapGet(/* Route Pattern */, () => /* Route-Delegate */);  
  
app.Run();
```

## Route

Jede Route besteht aus einem oder mehreren HTTP-Verben (MapGet), einem Route Pattern und einem Route Delegate

- MapGet: Die Route reagiert ausschließlich auf GET-Requests
- Route Pattern: Url Pfad auf den der Route-Delegate reagieren soll
- Route-Delegate: Der Handler der den Request beantwortet

# MINIMAL APIS

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

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

```
app.MapGet("/api/messages", () => "Hello World");  
app.MapPost("/api/messages", /* Add new message */);  
app.MapPut("/api/messages/{id}", /* Update message */);  
app.MapDelete("/api/messages/{id}", /* Delete Message */);
```

```
app.Run();
```

## Route-Parameters

ASP.NET unterstützt Parameter die der Route mitgegeben werden können, diese Parameter können aus der URL, dem Body oder der Query stammen. Man erkennt sie an den geschweiften Klammern. Eine Route kann mehr als einen Parameter definieren

# MINIMAL APIS

```
app.MapGet("/api/customers/{customerId}/orders/{orderId}",  
  (string customerId, string orderId) =>  
  {  
    // Kunde und Order aus der DB laden  
    // Result oder Fehler zurück liefern  
  });
```

## Route-Delegate

Der Route-Delegate erhält die Route-Parameter als Argumente. Im weiteren Verlauf werden wir sehen das der Delegate noch deutlich mehr Argumente aufnehmen kann, nicht ausschließliche Route-Parameter. Hier ist der Delegate als Lamda-Ausdruck definiert.

# MINIMAL APIS

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

var app = builder.Build();

app.MapGet("/api/customers/{customerId}/orders/{orderId}",
    HandleOrderDetailsRoute);

app.Run();

// Handler als "Local Function"
OrderDetails HandleOrderDetailsRoute(string customerId, string orderId)
{
    // Kunde und Order aus der DB laden
    // Result oder Fehler zurück liefern
    return new OrderDetails();
}
```

## Route-Delegate

Statt den Route-Delegate als Inline-Lambda-Ausdruck zu schreiben kann auch eine lokale, Instanz oder statische Methode verwendet werden.

# MINIMAL APIS

```
public static class CustomerRoutes
{
    public static void Register(WebApplication app)
    {
        app.MapGet("/api/customers/{customerId}/orders/{orderId}", HandleOrderDetailsRoute);
    }

    public static OrderDetails HandleOrderDetailsRoute(string customerId, string orderId)
    {
        // Kunde und Order aus der DB laden
        // Result oder Fehler zurück liefern
        return new OrderDetails();
    }
}
```

## CustomerRoutes.cs

Zur besseren Organisation können Routen in eigenen Dateien definiert und konfiguriert werden. Dies verkürzt die Program.cs deutlich und erhöht somit die Übersichtlichkeit signifikant.

# MINIMAL APIS

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

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

```
CustomerRoutes.Register(app);
```

```
app.Run();
```

## Program.cs

In der Program.cs kann nun die statische Klasse direkt registriert werden.

# MINIMAL APIS

- Die Routing-Middleware bietet noch weitere Möglichkeiten
  - Route-Constraints helfen die möglichen Route Parameter weiter einzuschränken
  - Das Routing ist sehr performant, da kaum Overhead durch das ASP.NET Framework vorhanden ist



# ROUTING| CONSTRAINTS

constraint	Example	Example Matches	Notes
int	{id:int}	123456789, -123456789	Matches any integer
bool	{active:bool}	true, FALSE	Matches true or false (case-insensitive)
datetime	{dob:datetime}	2016-12-31, 2016-12-31 7:32pm	Matches a valid DateTime value (in the invariant culture - see warning)
decimal	{price:decimal}	49.99, -1,000.01	Matches a valid decimal value (in the invariant culture - see warning)
double	{weight:double}	1.234, -1,001.01e8	Matches a valid double value (in the invariant culture - see warning)
float	{weight:float}	1.234, -1,001.01e8	Matches a valid float value (in the invariant culture - see warning)
guid	{id:guid}	CD2C1638-1638-72D5-1638-DEADBEEF1638, {CD2C1638-1638-72D5-1638-DEADBEEF1638}	Matches a valid Guid value
long	{ticks:long}	123456789, -123456789	Matches a valid long value
minlength(value)	{username:minlength(4)}	Rick	String must be at least 4 characters
maxlength(value)	{filename:maxlength(8)}	Richard	String must be no more than 8 characters
length(length)	{filename:length(12)}	somefile.txt	String must be exactly 12 characters long
length(min,max)	{filename:length(8,16)}	somefile.txt	String must be at least 8 and no more than 16 characters long
min(value)	{age:min(18)}	19	Integer value must be at least 18
max(value)	{age:max(120)}	91	Integer value must be no more than 120
range(min,max)	{age:range(18,120)}	91	Integer value must be at least 18 but no more than 120
alpha	{name:alpha}	Rick	String must consist of one or more alphabetical characters (a-z, case-insensitive)
regex(expression)	{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}\$)}	123-45-6789	String must match the regular expression (see tips about defining a regular expression)
required	{name:required}	Rick	Used to enforce that a non-parameter value is present during URL generation

# MINIMAL APIS

```
app.MapGet("/api/greetings/{id:int}", (int id) => $"Hello Nr. {id}");  
app.MapGet("/api/greetings/{name:alpha}", (string name) => $"Hello {name}");
```

## Route-Constraints

Durch die Constraints kann die gleiche Route für unterschiedliche Route-Parameter registriert werden.

# MODELBINDING

# Model Binding

```
POST http://localhost:64000/api/simple HTTP/1.1
Content-Type: application/json
Host: localhost:64000
Content-Length: 40
```

```
{
  "greeting" : "Jo ge leck da franzi"
}
```

```
public record GreetingDto(string Greeting);
```

```
app.MapPost("api/greetings", AddGreeting);
```

```
public static IActionResult AddGreeting(GreetingDto greeting)
{
    // ...
}
```

## 1. Request

POST-Request enthält als Body ein JSON-Objekt

## 2. Model Binding

Asp.NET weiß über den RequestDelegate, welche Parameter er erwartet. Daher ist bekannt, dass es einen Parameter `greeting` vom Typ `GreetingDto` geben soll. Eine neue Instanz von `GreetingDto` wird erzeugt und der Model Binder setzt die Werte in das Objekt aus den Daten aus dem Request

## 3. Nutzung

Das vom Model Binder erzeugte Objekt kann nun verwendet werden

# Model Binding

```
app.MapPut("api/greetings/{id}", UpdateGreeting);
```

```
public static IActionResult UpdateGreeting(string id, GreetingDto greeting)
{
    // ...
}
```

Der Model-Binder kann auch mit Route-Parametern genutzt werden. Somit erhalten wir sowohl eine id (aus der Request-Url), als auch unser gewünschtes GreetingDto-Objekt

# Model Binding

- Wandelt den Payload eines Requests in ein gefordertes POCO um (Plain Old C# Object)
- Verwendet automatisch JSON oder XML Deserialisierung (Content Negotiation) wenn es konfiguriert wurde (NuGet-Pakete)
- Eigene Serializer können registriert und bestehende konfiguriert werden
- Definition eigener Model Binder möglich
- Es kann nur ein komplexes Objekt als Parameter geben das aus dem Request kommt
- Mit der Attribute [FromBody] kann dem Model-Binder explizit mitgeteilt werden, das zur Deserialisierung der Body des Requests betrachtet werden soll (Bei größeren Objekten ist die Übertragung per URL nicht zu empfehlen).

# IRESULT

# IResult

- Statt direkt Objekte zurückzugeben, ist es empfehlenswerter das Ergebnis in ein IResult zu wrappen
- Wir müssen das Interface dabei nicht selbst implementieren
- Die Klasse Results liefert die am häufigsten benötigten Wrapper
- Die Methodennamen entsprechen meist dem gewünschten HTTP-Status
  - Results.Ok()
  - Results.NotFound()
  - Results.BadRequest()
  - Results.Created()
  - Results.Unauthorized()
- Aber auch Methoden für Streams
  - Results.File()
  - Results.Stream()



# IResult

```
app.MapPost("/evennumbers", ([FromBody] int number) =>
{
    return number % 2 == 0 ? Results.Ok(number) : Results.BadRequest();
});
```

```
app.MapPost("/evennumbers", ProcessEvenNumber);
```

```
app.Run();
```

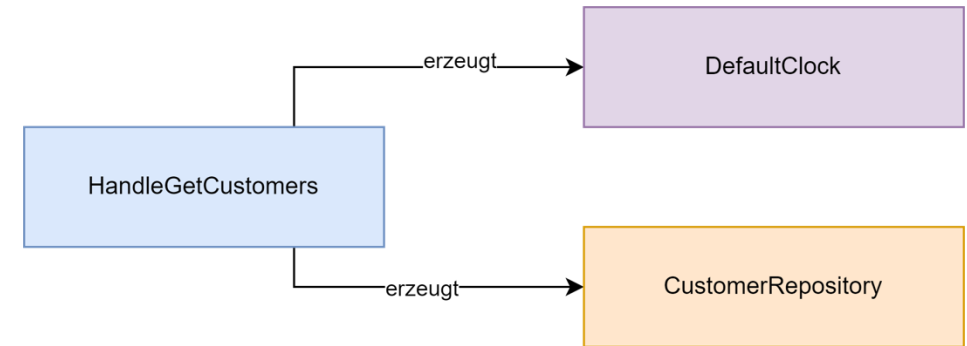
```
IResult ProcessEvenNumber([FromBody] int number)
{
    return (number % 2) switch
    {
        0 => Results.Ok(number),
        _ => Results.BadRequest($"is not a even number")
    };
}
```

## Results

Mit den Methoden an der Results-Klasse kann der gewünschte Ergebnis an den Aufrufer samt korrektem Status-Code zurückgegeben werden

# DEPENDENCY INJECTION

# Dependency Injection| Worum geht's?



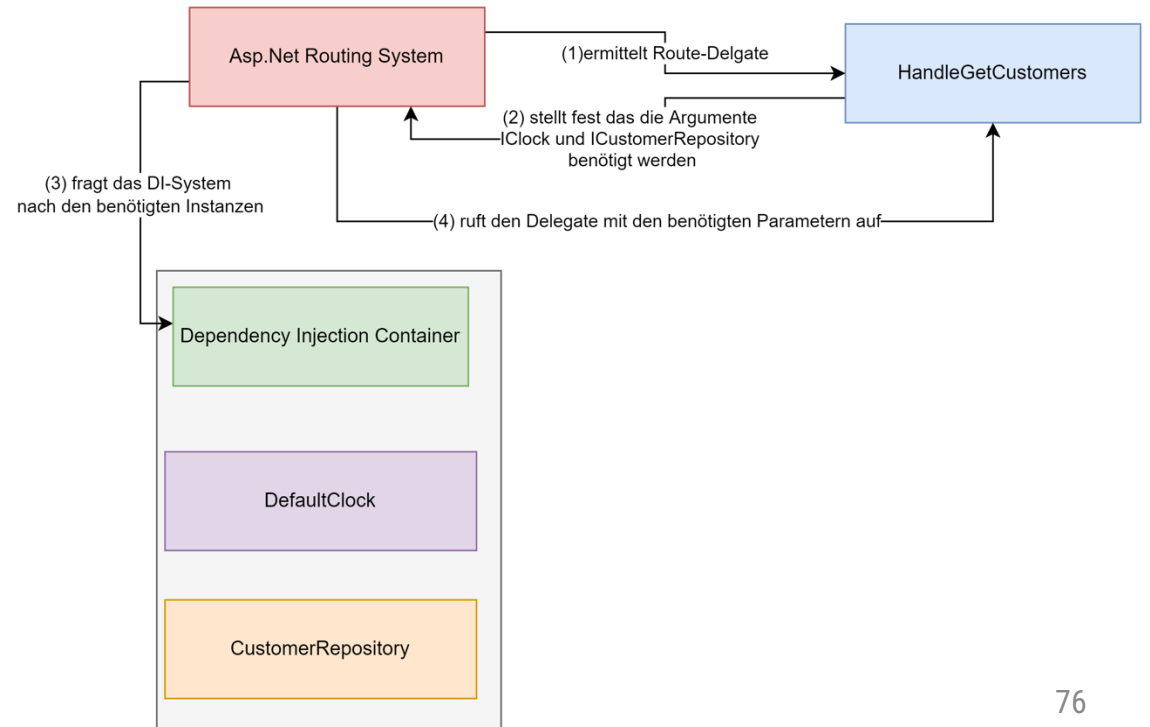
```
public static class CustomerEndpoints
{
    public static void Register(WebApplication app)
    {
        app.MapGet("/api/customers", HandleGetCustomers);
    }

    public static IReadOnlyCollection<Customer> HandleGetCustomers()
    {
        ICustomerRepository repository = new CustomerRepository();
        return repository.GetAll();
    }
}
```

# Dependency Injection| Worum geht's?

```
public static class CustomerEndpoints
{
    public static void Register(WebApplication app)
    {
        app.MapGet("/api/customers", HandleGetCustomers);
    }

    public static IReadOnlyCollection<Customer> HandleGetCustomers(IClock clock,
                                                                    ICustomerRepository repository)
    {
        var now = clock.Now;
        return repository.GetAll();
    }
}
```



# Scope und Lifetime von Dependencies

- Lifetime
  - Zeitspanne von der Erzeugung einer Dependency bis zu deren Ende (Dispose).
- Scope
  - Definiert wie Dependencies zwischen Komponenten verwendet werden können.
- Lifetime Scopes
  - **Singelton**: Nur eine Instanz pro Application
  - **Transient / Instance per Dependency**: Bei jeder Anfrage eine neue Instanz
  - **Scoped**: Für jeden HTTP-Request eine neue Instanz
- Das Konzept von Scopes / Lifetime ist wichtig um Memory-Leaks zu vermeiden.
- Für das Unit-of-Work-Pattern / Repository-Pattern eignet sich meist **Scoped**

# Dependency Injection| Implementierungen

- ASP.NET hat bereits eine einfache aber sehr performante Implementierung dabei
- StructureMap
  - Performant
  - Leider aktuell Probleme beim Integrieren da das Integrationspaket veraltet ist
- Autofac
  - Gute Dokumentation
  - Funktionierende Integrationspakete
  - Modularisierung
  - Performant
- Performance Vergleich: <https://github.com/stebet/DependencyInjectorBenchmarks>

# Dependency Injection

- Andere DI-Frameworks haben unterschiedliche Integrationsstrategien, meist gibt es aber NuGet-Pakete für die erleichterte Einbindung.
- Dependency Injection fügt der Anwendung ein nicht unerhebliches Maß an Komplexität hinzu. Im Gegenzug überwiegen die Vorteile durch die lose Koppelung von Abhängigkeiten und deren Konfigurierbarkeit.

# Dependency Injection

```
var builder = WebApplication.CreateBuilder(args);  
  
// Gleiche Instanz für die Lebensdauer der Applikation  
builder.Services.AddSingleton<IClock, DefaultClock>();  
  
// Gleiche Instanz von Beginn bis eine eines Requests  
builder.Services.AddScoped<ICustomerRepository, CustomerRepository>();  
  
// Neue Instanz bei jeder Anfrage  
builder.Services.AddTransient<NewObjectWithEveryRequest>();  
  
var app = builder.Build();  
  
// Middleware / HTTP-Pipeline konfigurieren  
  
app.Run();
```

## Hinweis

Es kann auch ein konkreter Typ (z.B. Singeltons) in das DI-System registriert werden, es muss nicht immer <Interface, ConcreteType> sein



# CONFIGURATION

# CONFIGURATION

ConfigurationBuilder stellt eine Fluent-API für die Konfiguration bereit (kann auch in nicht ASP.NET Projekten verwendet werden)

```
var builder = new ConfigurationBuilder()  
    .SetBasePath(env.ContentRootPath)  
    .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)  
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)  
    .AddEnvironmentVariables()  
    .AddUserSecrets();  
Configuration = builder.Build();
```

Build() erzeugt das Configuration-Objekt. Gleiche Settings aus verschiedenen Providern werden überschrieben. Daher ist die Reihenfolge der Aufrufe wichtig. Der Aufruf AddEnvironmentVariables() überschreibt alle vorangegangenen Werte.

Weitere Optionen sind über zusätzliche NuGet-Pakete verfügbar, z.B. Microsoft.Extensions.Configuration.Xml oder Microsoft.Extensions.Configuration.Ini. Ebenfalls können eigene Provider implementiert werden

# CONFIGURATION

Ab ASP.NET 2.2+ führt CreateDefaultBuilder automatisch zum Laden der Konfiguration aus typischen Locations (appsettings.json, Environment Variables). Das gleiche gilt für WebApplication.CreateBuilder() in .NET 6

# CONFIGURATION

appsettings.json

```
{
  "Features": {
    "UseInMemoryBookRepository": true
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

Handelt es sich um ein Development-Environment werden die Werte für den LogLevel überschrieben, bzw. ersetzt

appsettings.Development.json

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

# CONFIGURATION PROVIDER

Provider	Provides configuration from...
<a href="#">Azure Key Vault Configuration Provider</a> ( <i>Security topics</i> )	Azure Key Vault
<a href="#">Command-line Configuration Provider</a>	Command-line parameters
<a href="#">Custom configuration provider</a>	Custom source
<a href="#">Environment Variables Configuration Provider</a>	Environment variables
<a href="#">File Configuration Provider</a>	Files (INI, JSON, XML)
<a href="#">Key-per-file Configuration Provider</a>	Directory files
<a href="#">Memory Configuration Provider</a>	In-memory collections
<a href="#">User secrets (Secret Manager)</a> ( <i>Security topics</i> )	File in the user profile directory

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-2.2#providers>

# CONFIGURATION

```
public class Features
{
    public bool HasLightspeed { get; private set; }
    public bool UseBetaApi { get; private set; }
    public string? EmailOverride { get; private set; }
}
```

# CONFIGURATION

Zugriff auf die Konfiguration kann  
untypisiert oder typisiert sein

```
string useInMemoryBookRepositorySetting = Configuration["Features:UseInMemoryBookRepository"];  
bool useInMemoryBookRepository = Configuration.GetValue<bool>("Features:UseInMemoryBookRepository");
```

```
var features = builder.Configuration.GetSection("Features")  
    .Get<Features>(opt => opt.BindNonPublicProperties = true);
```

Es muss nicht die gesamte Konfiguration  
typisiert sein, es können auch nur einzelne  
SubSections typisiert werden

```
public class Features  
{  
    public bool UseInMemoryBookRepository { get; set; }  
}
```

# CONFIGURATION

Die Konfiguration oder Teile davon können in das DI-System eingespeist werden.

```
builder.Services.Configure<Features>(builder.Configuration.GetSection("Features"),  
    opt => opt.BindNonPublicProperties = true);
```



# DEPENDENCY INJECTION DER OPTIONS

**IOptions<T>** und **IOptionsSnapshot<T>** sind Möglichkeiten eine aktuelle Repräsentation der Optionen im Controller zu verwenden. IOptionsSnapshot besitzt dabei die Möglichkeit auf Änderungen an der Konfiguration zu reagieren, wenn der Provider es unterstützt

```
app.MapGet("/features", (IOptions<Features> options) =>
{
    return Results.Ok(options.Value);
});
```

# VALIDIERUNG MIT FLUENT VALIDATION

# Fluent Validation

## \* FLUENT VALIDATION

A popular .NET library for building strongly-typed validation rules.

nuget v8.4.0 downloads 14M Azure Pipelines succeeded

Download Now

View On Github

```
public class CustomerValidator : AbstractValidator<Customer> {  
    public CustomerValidator() {  
        RuleFor(x => x.Surname).NotEmpty();  
        RuleFor(x => x.Forename).NotEmpty().WithMessage("Please specify a first name");  
        RuleFor(x => x.Discount).NotEqual(0).When(x => x.HasDiscount);  
        RuleFor(x => x.Address).Length(20, 250);  
        RuleFor(x => x.Postcode).Must(BeAValidPostcode).WithMessage("Please specify a valid postcode");  
    }  
  
    private bool BeAValidPostcode(string postcode) {  
        // custom postcode validating logic goes here  
    }  
}
```



Getting Started



Built-in Validators



Custom Validators

<https://fluentvalidation.net/>

# Fluent Validation

- OpenSource NuGet-Package
- Definition von Validatoren mittels Builder-Pattern
- Unterstützung von asynchronen Validatoren
- Validatoren können aus anderen Validatoren kombiniert werden
  - z.B. PersonValidator = NameValidator + AddressValidator
- Bindet sich in ASP.NET MVC ein
  - Mittels FluentValidation.AspNetCore
  - **ACHTUNG: Nur MVC , nicht Minimal APIs**
- Gute Unit-Test-Unterstützung
- Wiederwendbare Property Validatoren

# Fluent Validation

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public bool WantsToParty { get; set; }
    public int Age { get; set; }
}
```

# Fluent Validation

```
public record Student(string FirstName,  
    string LastName,  
    bool WantsToParty,  
    int Age);
```

# Fluent Validation

```
public class StudentValidator : AbstractValidator<Student>
{
    private const int MinimumAgeToParty = 18;

    public StudentValidator()
    {
        RuleFor(s => s.FirstName).NotEmpty();
        RuleFor(s => s.LastName).NotEmpty();
        When(s => s.WantsToParty, () =>
        {
            RuleFor(s => s.Age)
                .GreaterThan(MinimumAgeToParty)
                .WithMessage("Not old enough to party");
        });
    }
}
```

# Fluent Validation

```
Student student = null;  
var validator = new StudentValidator();  
  
var validationResult = validator.Validate(student); // Oder ValidateAsync
```

## ValidationResult

Das Ergebnis der Validierung enthält eine IsValid Property und eine Auflistung aller aufgetretenen Validierungsfehler in der Property Errors.



# ASP.NET MVC

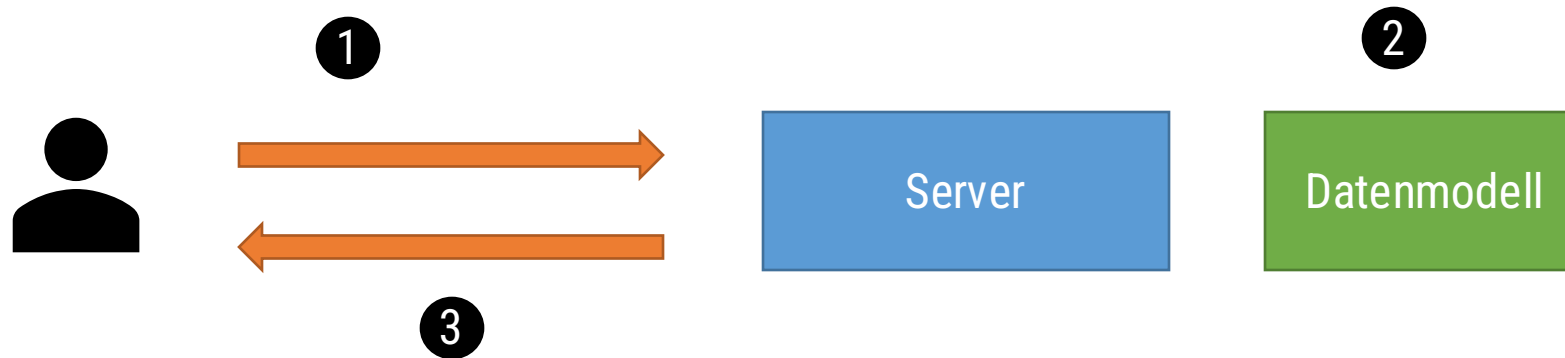
# DAS MVC PATTERN

- Weitere Möglichkeit zum Erstellen von REST-APIs mit ASP.NET
- Ursprüngliche Version vor „Minimal APIs“ und „Endpoint Routing“
- Weniger funktional- mehr objektorientierter Ansatz
- Kann mit Minimal APIs kombiniert werden (Mix-and-Match)

# DAS MVC PATTERN

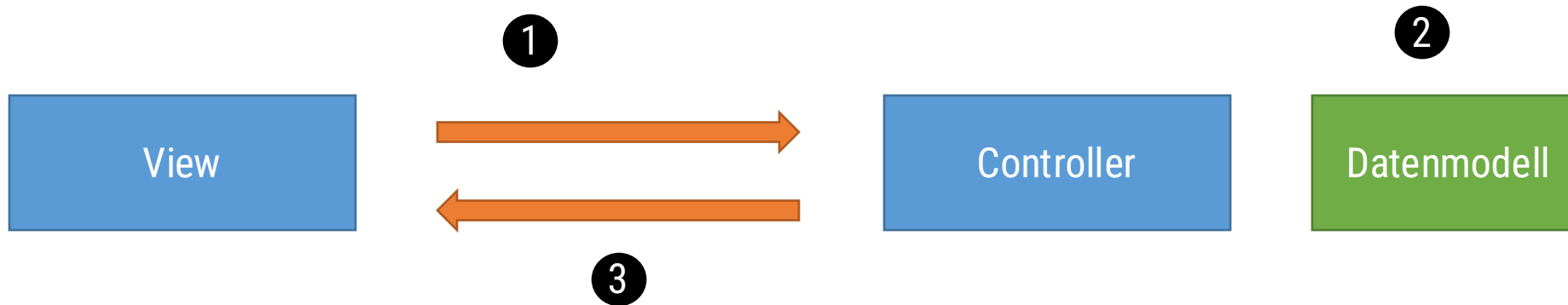
- **Model**
  - Repräsentation von Daten mit denen gearbeitet wird
  - Business Logik
  - Domain Model
  - View Model
- **View**
  - Rendert Teile der Daten als UI
- **Controller**
  - Verarbeitet eingehende Requests
  - Führt Operationen am Datenmodell durch
  - Selektiert den / die Views die erzeugt werden sollen

# DAS MVC PATTERN



- 1** Benutzer löst eine Aktion aus
- 2** Datenmodell wird aktualisiert
- 3** View für den Benutzer erzeugen und ausliefern

# DAS MVC PATTERN



- ① Ein Request geht ein
- ② Datenmodell wird aktualisiert
- ③ Eine Response wird erzeugt und zurückgegeben

# ASP.NET MVC

- ASP.NET MVC ist auch „nur“ Middleware

# KONFIGURATION ASP.NET MVC

Middleware benötigt in der Regel auch einige Services, um die Funktionalität bereitstellen zu können und so auch MVC. Die ConfigureServices-Methode ist die Stelle, um diese am Dependency-Injection-System zu registrieren. Viele Middleware-Komponenten liefern spezielle Extension-Methoden um die Registrierung durchzuführen (**Add[Middleware Component Services]()**)

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

```
// Notwendige Services für MVC Controllers registrieren
```

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

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

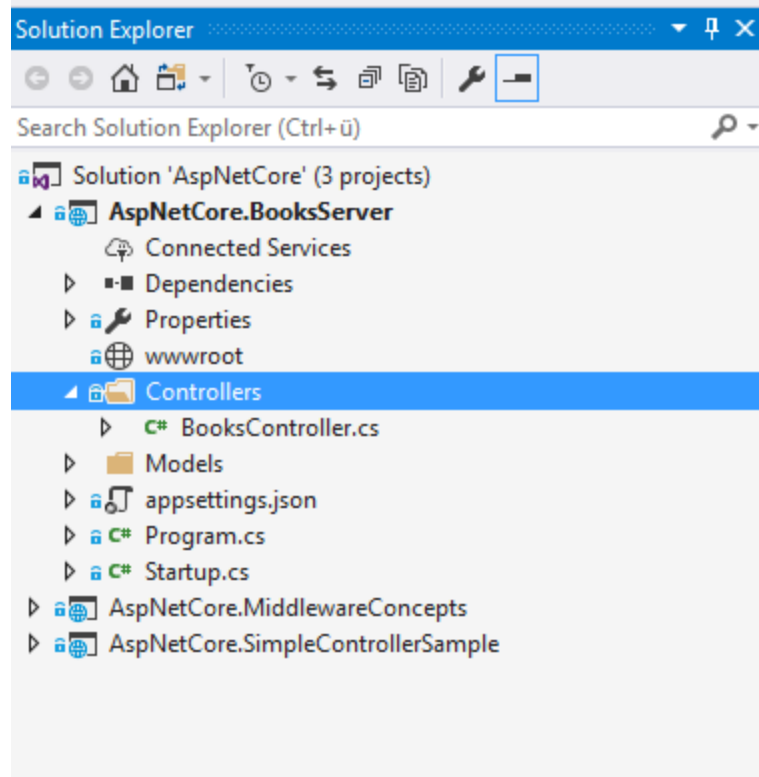
```
// Middleware / HTTP-Pipeline konfigurieren
```

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

```
app.Run();
```

Anschließend muss noch die MVC-Middleware in die Request-Pipeline eingehängt werden. Die meisten Middleware-Komponenten stellen dafür Extension-Methoden bereit (**Use...()** oder **Map...()**)

# ASP.NET MVC| CONTROLLERS



Per Konvention liegen die Controller-Klassen im Controller-Ordner des Projekts. Häufig besser ist aber die Strukturierung nach Feature. Gerade bei größeren API-Projekten ist eine derartige Strukturierung sinnvoll.



# ASP.NET MVC| CONTROLLERS

Attribute-based Routing ist meist die effektivste Möglichkeit eine URL auf einen Controller und eine Action zu mappen\*

```
[Route("api/[controller]")]
public class SimpleController : ControllerBase
{
    [HttpGet]
    public string GetGreeting()
    {
        return "Hello World";
    }
}
```

In der einfachsten Form wird der Controller durch eine einfache C#-Klasse repräsentiert. Durch das Ableiten von der Controller-Klasse erhält man eine Vielzahl nützlicher Methoden für die Erzeugung von REST-APIs.

Public Methoden innerhalb des Controllers bezeichnet man Controller-Actions. Über Attribute wird gesteuert, auf welche URL (-Patterns) und HTTP-Verben reagiert wird.

# ATTRIBUTE ROUTING

# ATTRIBUTE ROUTING

- Routing bezeichnet das Mapping einer URL auf einen Controller und eine seiner Actions.
- Routing ist case-insensitive
- Mehrfachangabe von Routing-Attributen erlaubt
- MVC stellt folgende Attribute für das Routing bereit
  - Http[Verb] z.B. HttpGet, HttpPost, HttpPut, HttpDelete
- Die Routen können durch sog. Constraints weiter eingeschränkt werden
  - Achtung dieses Feature nicht zur Validierung von Daten verwenden!

# ATTRIBUTE ROUTING

```
// GET api/books  
[HttpGet]  
public IEnumerable<Book> GetBooks()
```

```
// GET api/books/1  
[HttpGet("{id}")]  
public IActionResult GetBookById(int id)
```

In dem URL-Pattern der Route können auch Parameter eingegeben werden. Der Name des Parameters in der Route muss mit dem der Controller-Action übereinstimmen

```
// POST api/books  
[HttpPost]  
public IActionResult CreateBook(Book book)
```

```
// PUT api/books/1  
[HttpPut("{id}")]  
public IActionResult UpdateBook(int id, Book book)
```

# ATTRIBUTE ROUTING

```
// GET api/books/1  
[HttpGet("{id:int}")]
```

```
public IActionResult GetBookById(int id)
```

Constraints ermöglichen weitere Einschränkungen der URL welche zu einer Aktivierung des Controllers führt

```
// ~ überschreibt das RoutePrefix
```

```
// GET api/authors/skeet/books
```

```
[HttpGet("~/api/authors/{author:alpha}/books")]
```

```
public IEnumerable<Book> GetBookByAuthorName(string author)
```

Mit dem Attribute-Routing lassen sich sehr komplexe Routen definieren und auch das Child-Route-Pattern umsetzen.

```
[HttpGet("~/api/authors/{author:alpha}/books/{year:int:min(1950):max(2050)}")]
```

```
// GET api/authors/skeet/books/2015
```

```
public IEnumerable<Book> GetBookByAuthorNameInYear(string author, int year)
```

# HTTP PATCH

# PATCH VS PUT

- PUT aktualisiert ein komplettes Objekt
- PATCH kann auch nur Teile eines Objektes aktualisieren
- Für das teilweise Aktualisieren mittels JSON gibt es einen Standard <https://tools.ietf.org/html/rfc6902>
- Für ASP.NET wird die Unterstützung mittels dem generischen Typen **JsonPatchDocument<T>** bereitgestellt

# PATCH VS PUT

```
[HttpPatch("{id}")]
public IActionResult JsonPatch(string id, [FromBody] JsonPatchDocument<Book> doc)
{
    if (doc == null)
        return BadRequest();

    // Buch zur Bearbeitung laden
    Book book = GetBookById(id);

    // Patch auf Objekt anwenden
    doc.ApplyTo(book, ModelState);

    // Prüfen ob das Model nach dem Patch noch gültig ist
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    // Objekt speichern
    UpdateBook(book);

    // 200 OK samt aktualisiertem Objekt zurückgeben
    return Ok(book);
}
```



# PATCH VS PUT

Unterstützt werden als Operatoren add, remove, replace, move, copy und test

Ein JSON-PUT besteht aus einem Array von Updates

```
[
  {
    "op": "add",
    "path": "/title",
    "value": ".NET Memory Management"
  }
]
```

Der path kann auch im Objektgraph navigieren, z.B.  
/authors/address/0/street

# PATCH VS PUT

- PATCH ist zwar komplizierter in der Anwendung, gerade bei größeren (JSON) Objekten und leistungsschwächeren Geräten (IoT) kann sich aber ein erheblicher Performancevorteil ergeben

# MODELSTATE

# Model State

```
public class GreetingDto
{
    [Required]
    public string Greeting { get; set; }
}
```

## Validatoren

Es können die Annotationen aus den DataAnnotations-Namespace verwendet oder eigene erfunden werden

```
public IActionResult PostGreeting(GreetingDto greeting)
{
    if (greeting == null || !ModelState.IsValid)
    {
        return BadRequest(GetErrorMessage() ?? "No greeting provided");

        // oder
        //return BadRequest(ModelState);
    }

    // ...
}
```

## ModelState

Der Validierungszustand des Models kann innerhalb einer Controller-Action mit ModelState abgerufen werden. Achtung, das Modell kann auch NULL sein

```
private string GetErrorMessage()
{
    return string.Join(";", ModelState.Values.SelectMany(v => v.Errors).Select(e => e.ErrorMessage));
}
```

## Errors

Die Exceptions und Fehlermeldungen werden pro Eigenschaft des DTO's erfasst und können über den ModelState abgerufen werden

# Model State

- Während des Model Bindings werden an dem POCO definierte Validatoren ausgeführt
- Die Eigenschaft **ModelState** steht innerhalb einer Controller Action zur Verfügung und gibt Auskunft über den Validierungsstatus
- Über **ModelState** kann iteriert werden und auf die Error-Messages zugegriffen werden
- Mit ModelState.**GetFieldValidationState**(key) kann ein Feld direkt auf Korrektheit geprüft werden

# Model State Validation

- [Required]
- [CreditCard]
- [Compare]
- [EmailAddress]
- [Phone]
- [Range]
- [RegularExpression]
- [StringLength]
- [Url]
- [Remote]

<https://docs.microsoft.com/de-de/aspnet/core/mvc/models/validation?view=aspnetcore-2.2#built-in-attributes>

# Model State Validation

```
public class OldEnoughAttribute : ValidationAttribute
{
    private const int MinimumAge = 18;

    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        // Über den ValidationContext kann auf das validierende Objekt zugegriffen werden.

        if(value is int age && age >= MinimumAge)
        {
            return ValidationResult.Success;
        }

        return new ValidationResult($"Sorry, must be older than {MinimumAge}.");
    }
}
```

# Model State

- Die Validierung mittels Data-Annotations hat viele Unzulänglichkeiten
  - Abhängige Validierungen nicht möglich
  - Das „umgebende“ Framework muss die Annotations auswerten
  - Keine Wiederverwendung von Validierungsblöcken
  - Schwerere Testbarkeit
- Die FluentValidations-Library lässt sich vollständig in Asp.Net MVC integrieren und übernimmt die Aufgaben der Data-Annotations
- [ASP.NET Core – FluentValidation documentation](#)



# MINIMAL APIS VS CONTROLLERS (MVC)

# MINIMAL APIS VS CONTROLLERS (MVC)

## Minimal APIs

- **PRO:**
  - Perfekt für den Einstieg in Webservices
  - Wenig Boiler-Plate
  - Meiste Funktionalität von Controllern auch hier vorhanden
  - Performanter als MVC da eine Abstraktionsebene tiefer
- **CON:**
  - Swagger / OpenId support noch unvollständig z.B. keine XML Dokumentation
  - Noch keine "Best-Practices" da zu neu

## Controllers (MVC)

- **PRO:**
  - Strukturierung
  - (Partial) PATCH Support
  - Battle Tested, Best Practices
  - Voller Swagger / Open Id Support
  - Basis-Funktionalität durch Ableitungen (Base Controller)
  - Einfache Definition mehrerer Routen für die gleiche Action
- **CON:**
  - Boilerplate und viele Konzepte gleichzeitig zu erlernen (Controller, Routes, Actions)

# MINIMAL APIS VS CONTROLLERS (MVC) BEST PRACTICES

- Logik in Controller Actions und Minimal API Delegates möglichst gering halten
- Business Logik in Domain Driven Design (DDD) oder mittels Command Query Responsibility Segregation mittels MediatR
- Keine Exceptions werfen sondern eigene Ergebnistypen verwenden (Responses), anhand dieser Typen den korrekten HTTP-Status ermitteln