

REST WEBSERVICES MIT ASP.NET CORE

GITHUB



- Sourcen mit Beispielen zum Skript finden sie unter <https://github.com/florianwachs/FHRWebservices>

ASP.NET UND .NET CORE

- Offizielle Begrifflichkeiten
 - .NET Core
 - ASP.NET Core
 - dotnet-cli

ASP.NET UND .NET CORE VERSIONEN

- Aktuell (5.2020):
 - .net core 3.1
 - Asp.net core 3.1
 - LTS
- November 2020
 - .net (core) 5

WARUM ASP.NET CORE?

PROBLEME VON ASP.NET

- Aufstieg von Plattformen wie Node.js für serviceorientierte Architekturen
- Nachteile von ASP.NET gegenüber leichtgewichtigen Alternativen aufgrund des „All-Inclusive“-Ansatzes
- Aktualisierungen und neue Features meist nur halbjährlich
- Schwierigkeiten bei der Unterstützung von Containertechnologien wie Docker
- Höhere Lizenzkosten wegen Abhängigkeiten zu Windows
- Keine Cross-Plattform-Unterstützung
- Überschneidungen von APIs in ASP.NET WebAPI und MVC (Filters, Modelbinding Actions)

PROBLEME VON ASP.NET

	ASP.NET	
	MVC	Web API
Einsatzgebiet	Dynamische Generierung von HTML	REST-basierte Webservices
Basisklassen	Controller	ApiController
Konfiguration	Global.asax, web.config (XML)	OWIN Startup.cs
Routing	RoutingMap mit Parametern, Attribute-based	Attribute-based
Response-Objekte der Controller	ActionResult	IHttpResponse
ActionFilter	Unterschiedliche Implementierungen	
Dependency Injection System	Unterschiedliche Implementierungen	
....		

PROBLEME VON ASP.NET

- MVC und Web API basieren auf ähnlichen Konzepten unterscheiden sich aber in der Implementierung
- Stellenweise doppelte Implementierungen derselben Funktionalität notwendig
- MVC und Web API können im gleichen Webprojekt eingesetzt werden, müssen aber getrennt konfiguriert werden
- Die Behandlung von Requests in MVC und Web API läuft unterschiedlich
- Probleme für das ASP.NET Team:
 - Funktionalitäten müssen doppelt entwickelt werden
 - 2 Implementierungen müssen gewartet werden

WARUM ASP.NET CORE?

- 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 CORE EXTENSIBILITY

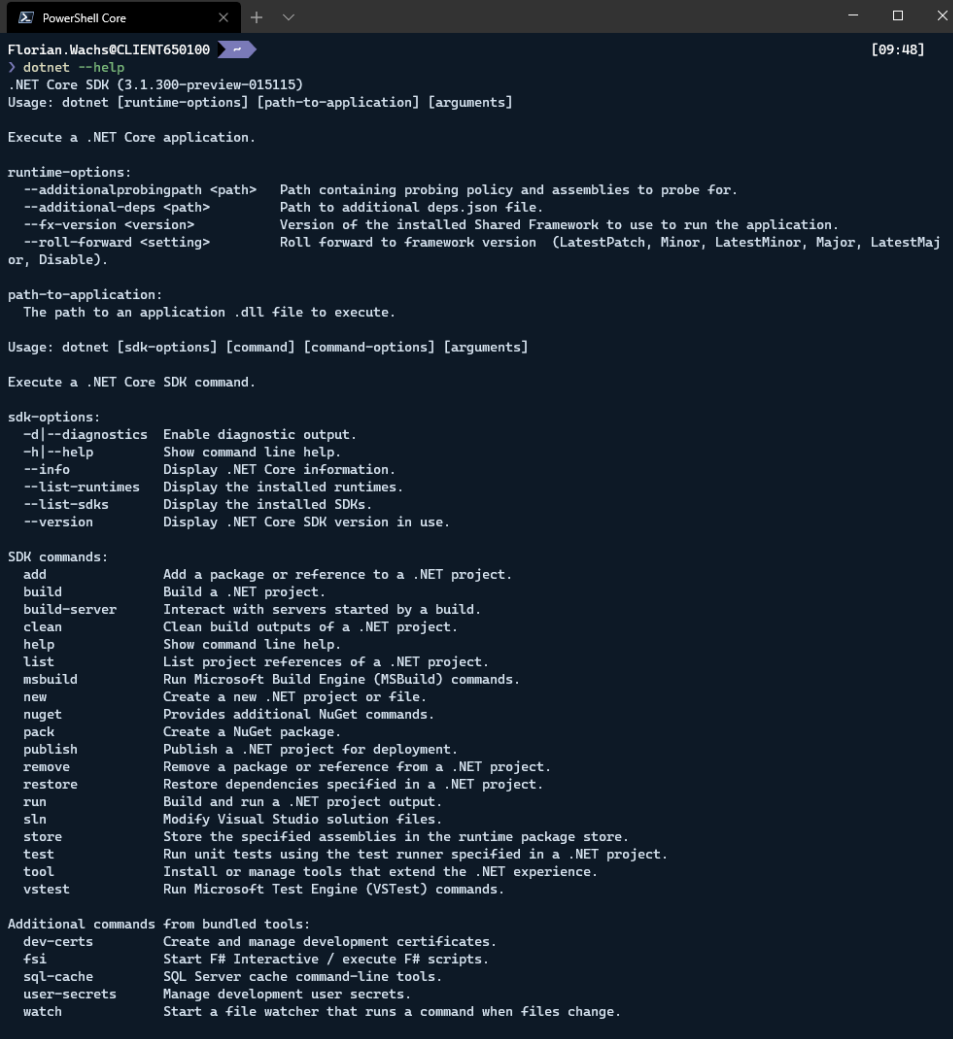
- Die Defaults verwenden (Implementiert über NuGet-Pakete)
- Von den bestehenden Klassen ableiten und Funktionalität anpassen
- Komplette Eigenimplementierung von Interfaces / abstrakten Klassen für maximale Anpassung

ASP.NET CORE INSTALLIEREN

- <https://dot.net>

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



```
Florian.Wachs@CLIENT650100
> dotnet --help
.NET Core SDK (3.1.300-preview-015115)
Usage: dotnet [runtime-options] [path-to-application] [arguments]

Execute a .NET Core application.

runtime-options:
--additionalprobingpath <path>  Path containing probing policy and assemblies to probe for.
--additional-deps <path>        Path to additional deps.json file.
--fx-version <version>          Version of the installed Shared Framework to use to run the application.
--roll-forward <setting>        Roll forward to framework version (LatestPatch, Minor, LatestMinor, Major, LatestMajor, or, Disable).

path-to-application:
The path to an application .dll file to execute.

Usage: dotnet [sdk-options] [command] [command-options] [arguments]

Execute a .NET Core SDK command.

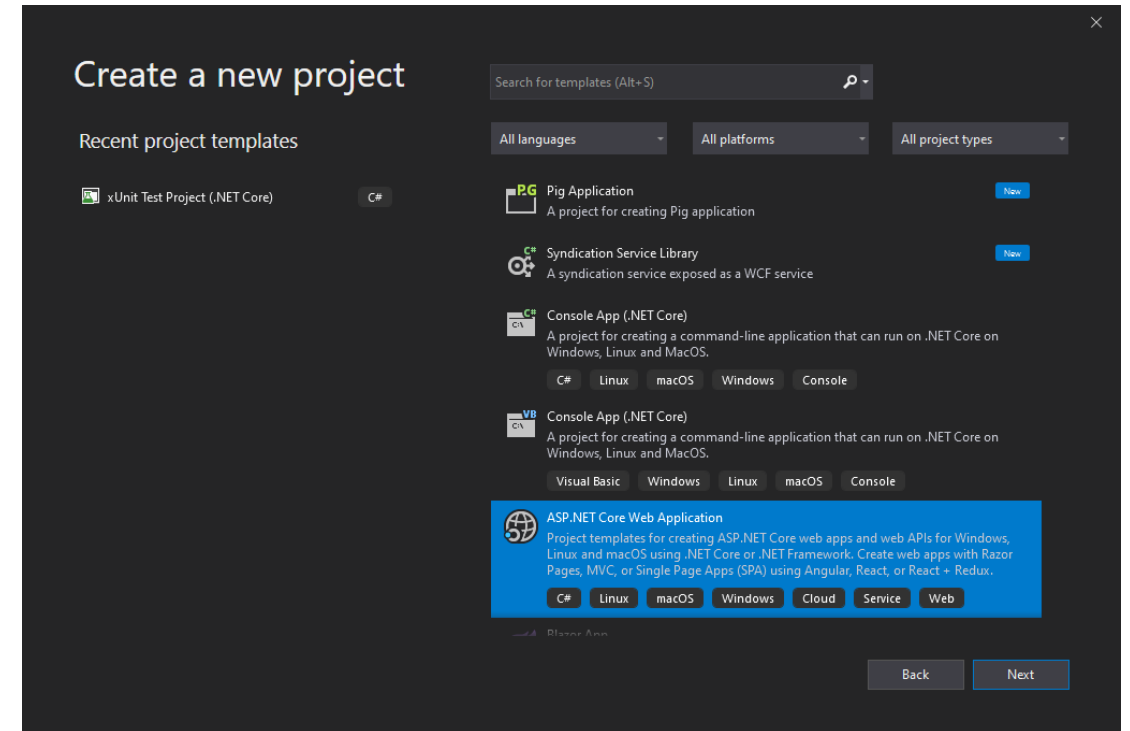
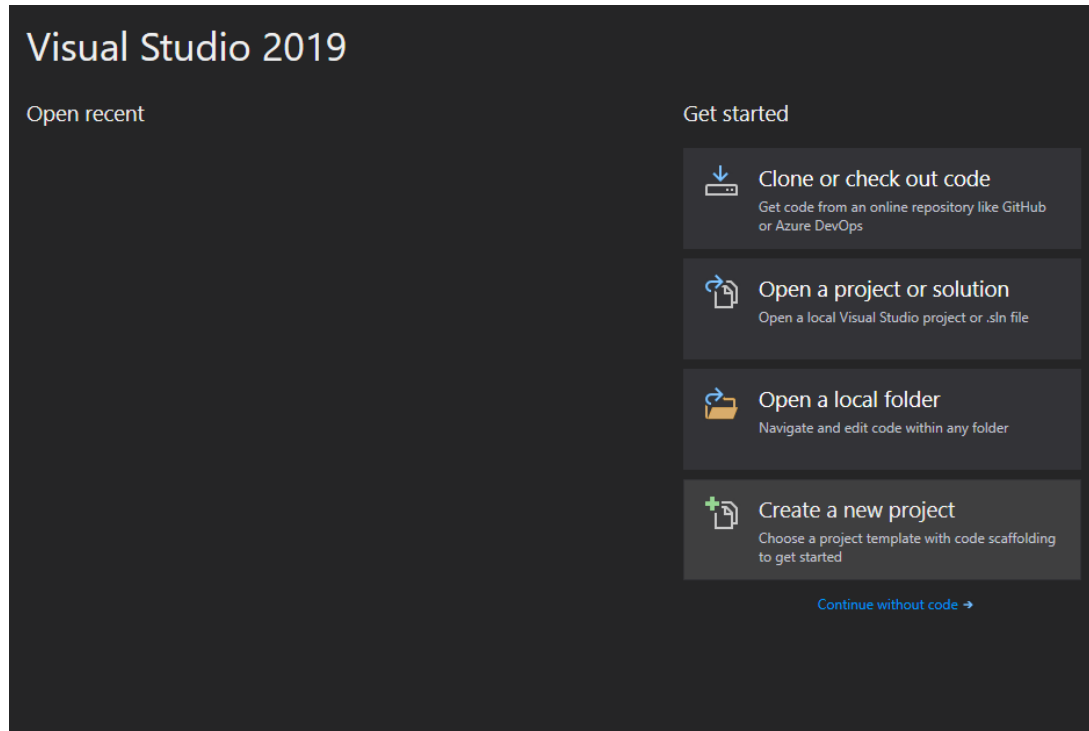
sdk-options:
-d|--diagnostics  Enable diagnostic output.
-h|--help         Show command line help.
--info           Display .NET Core information.
--list-runtimes  Display the installed runtimes.
--list-sdks      Display the installed SDKs.
--version        Display .NET Core SDK version in use.

SDK commands:
add              Add a package or reference to a .NET project.
build            Build a .NET project.
build-server    Interact with servers started by a build.
clean           Clean build outputs of a .NET project.
help            Show command line help.
list            List project references of a .NET project.
msbuild         Run Microsoft Build Engine (MSBuild) commands.
new             Create a new .NET project or file.
nuget           Provides additional NuGet commands.
pack            Create a NuGet package.
publish         Publish a .NET project for deployment.
remove          Remove a package or reference from a .NET project.
restore         Restore dependencies specified in a .NET project.
run             Build and run a .NET project output.
sln             Modify Visual Studio solution files.
store           Store the specified assemblies in the runtime package store.
test            Run unit tests using the test runner specified in a .NET project.
tool            Install or manage tools that extend the .NET experience.
vstest          Run Microsoft Test Engine (VSTest) commands.

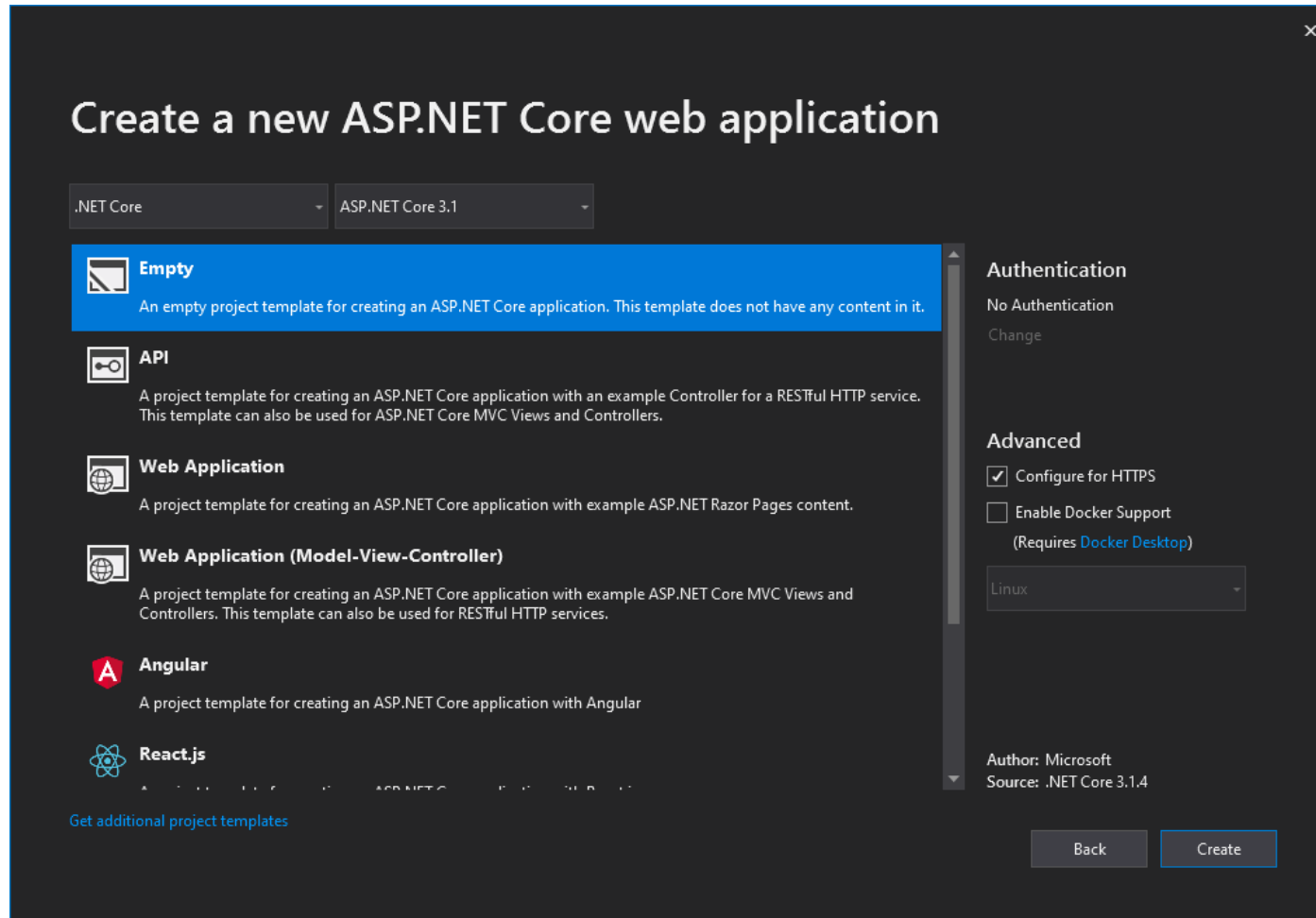
Additional commands from bundled tools:
dev-certs       Create and manage development certificates.
fsi             Start F# Interactive / execute F# scripts.
sql-cache       SQL Server cache command-line tools.
user-secrets    Manage development user secrets.
watch           Start a file watcher that runs a command when files change.
```

VISUAL STUDIO 2019

■ New Project



VISUAL STUDIO 2019



ASP.NET Core SDK

SDK

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

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

```
  <PropertyGroup>
```

```
    <TargetFramework>netcoreapp3.1</TargetFramework>
```

```
  </PropertyGroup>
```

```
</Project>
```

TargetFramework

Steuert implizit welche Metapakete eingebunden werden

ASP.NET Core SDK

- Spezielles SDK das auf das „Share-Framework“ verweist
- Updates über .NET SDK / Runtimeinstaller
- Zusätzliche Features wie Authentifizierung, GraphQL, GRPC, Entity Framework usw. müssen über NuGet-Pakete eingebunden 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/Student/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

HTTP Status Codes

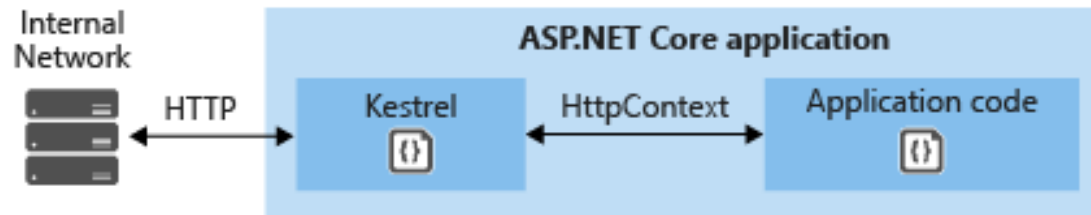
- 200: Alles OK
- 201: Resource created
 - Location Header enthält die URI zur neuen Ressource
- 400: Bad Request
- 401: Unauthorized
 - Sollte dem Aufrufer die Art der geforderten Authentifizierung mitteilen
- 403: Access denied
 - Authentifiziert aber nicht autorisiert, um auf die Ressource zuzugreifen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

HOSTING

HOSTING VON ASP.NET CORE

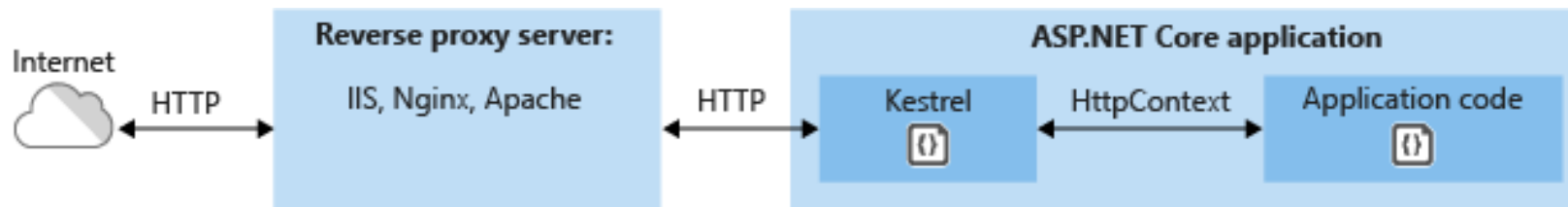
Kestrel

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



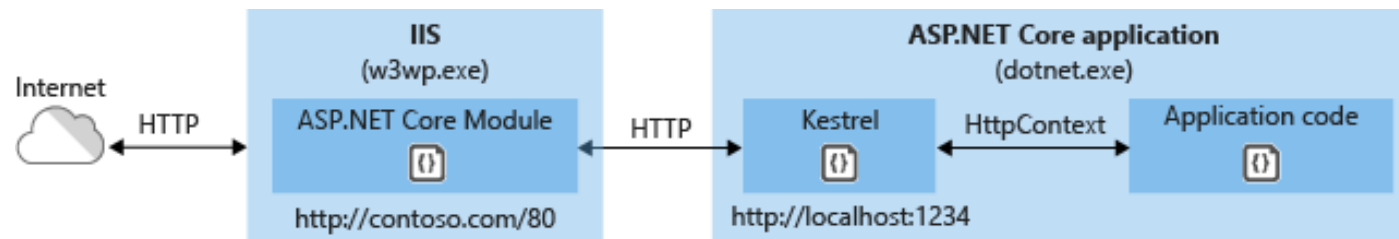
Kestrel ohne Reverse Proxy

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



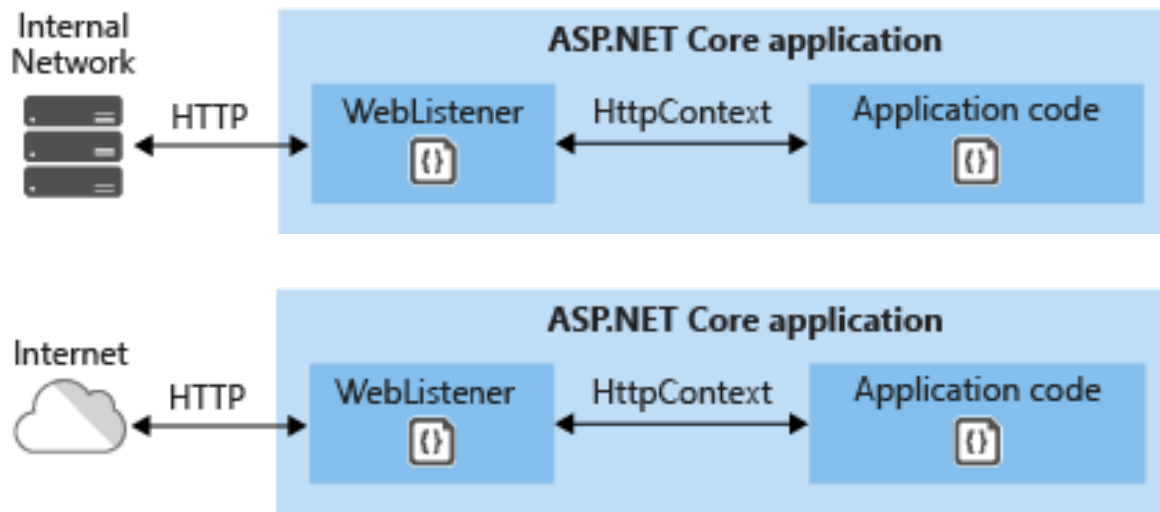
HOSTING VON ASP.NET CORE

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



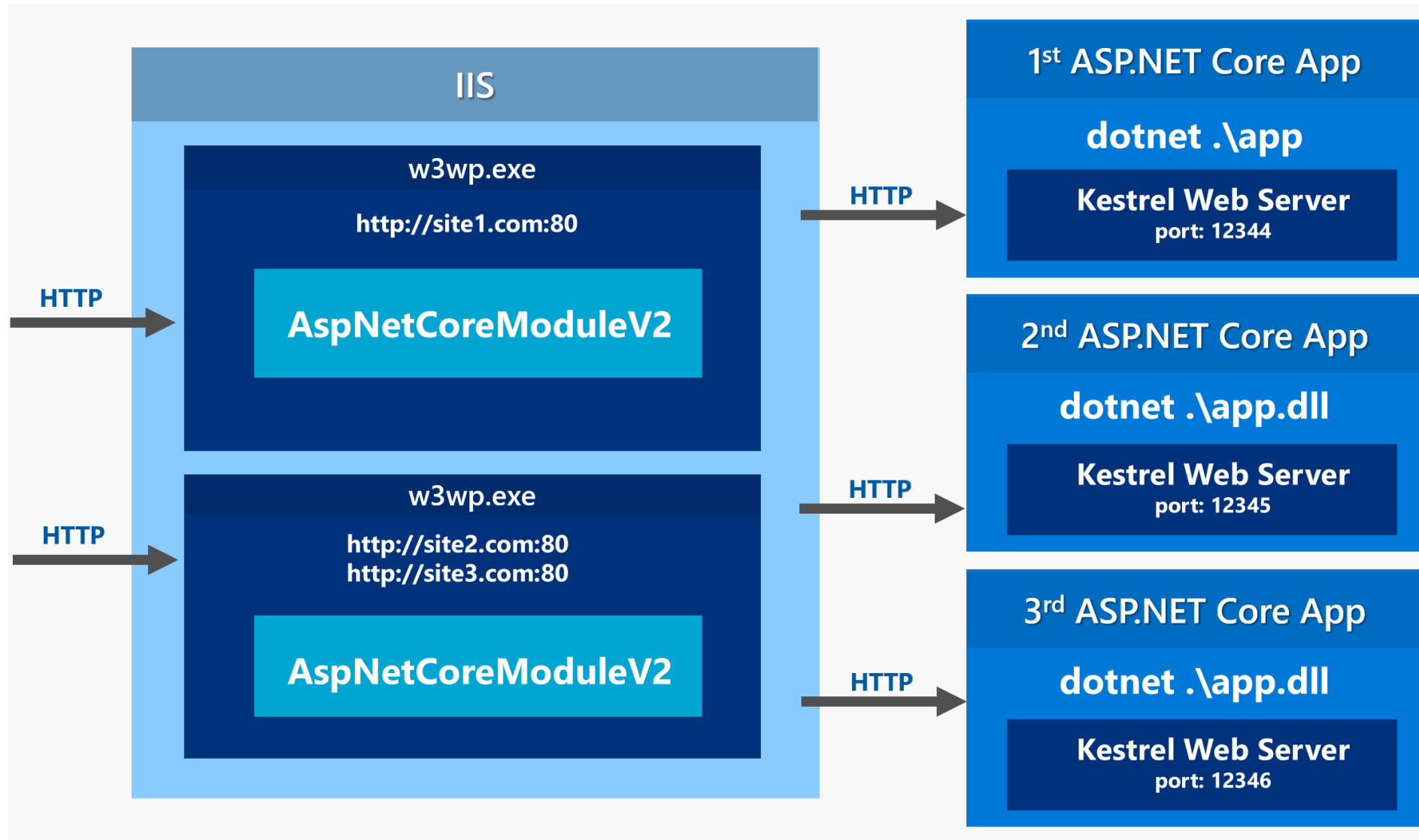
HOSTING VON ASP.NET CORE

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



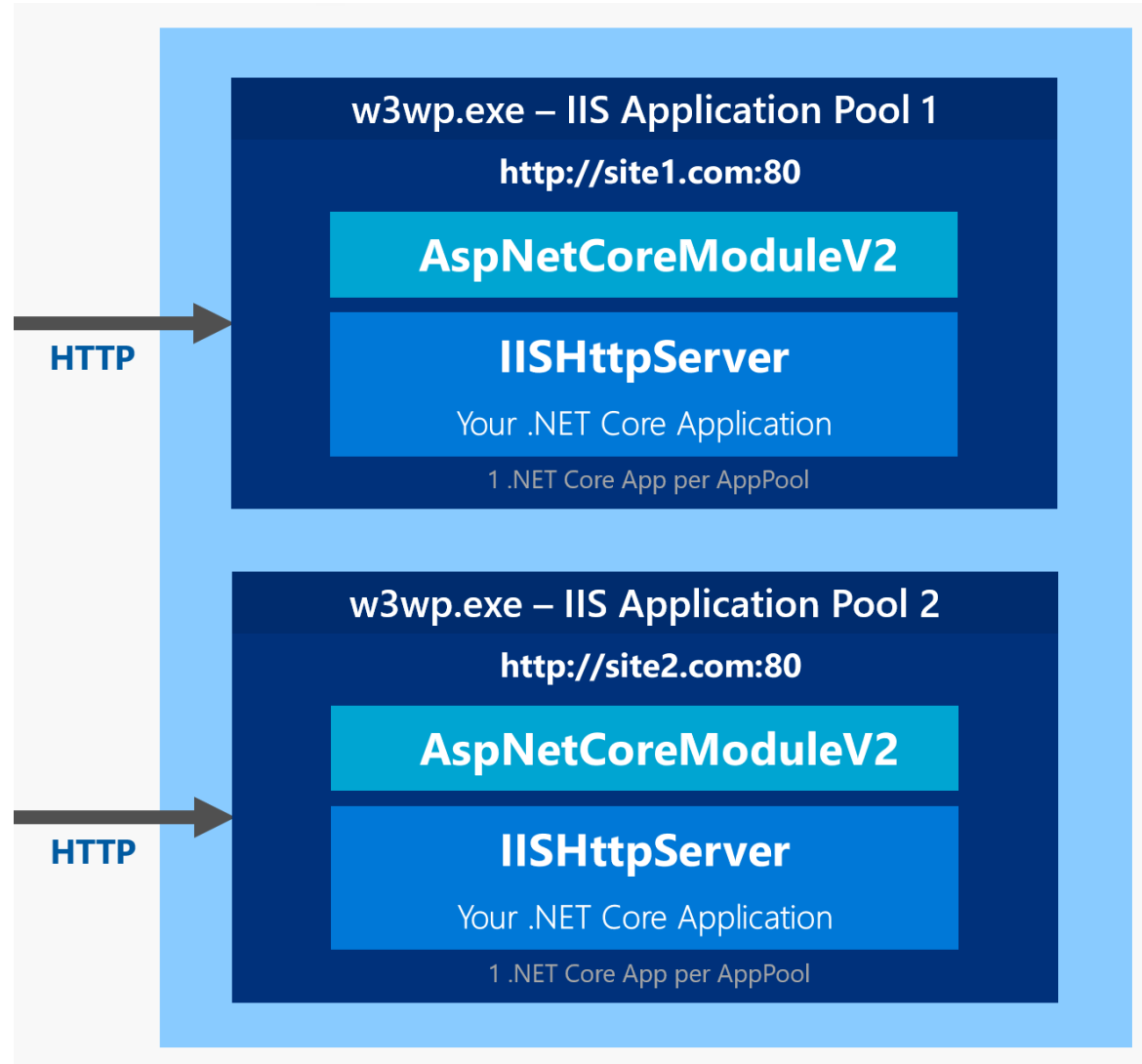
Ist eine gute Lösung wenn keine Features vom IIS benötigt werden

ASP.NET CORE 2.2+ 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 CORE 2.2+ 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 CORE 2.2+ 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

ASP.NET CORE 3.0+ KESTREL

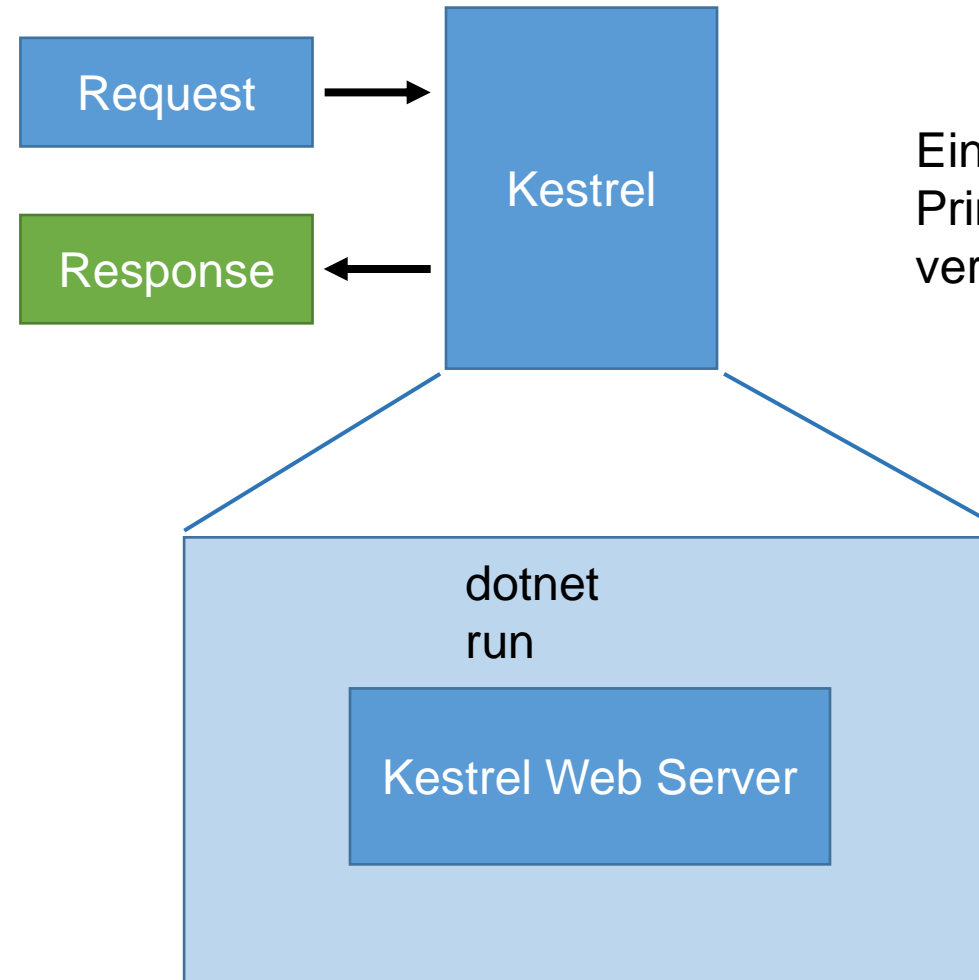
- Ab 3.0 ist Kestrel für die Verwendung ohne Reverse Proxy freigegeben.
- Aktuell (2020-05) entsteht ein auf .net core / Kestrel basierender Reverse Proxy (YARP)

WIE WIRD DAS ASP.NET CORE HOSTING KONFIGURIERT?

HOST

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

WEBHOSTBUILDER



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

KONFIGURATION DES HOSTINGS

Mittels **HostBuilder** wird das Hosting konfiguriert.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

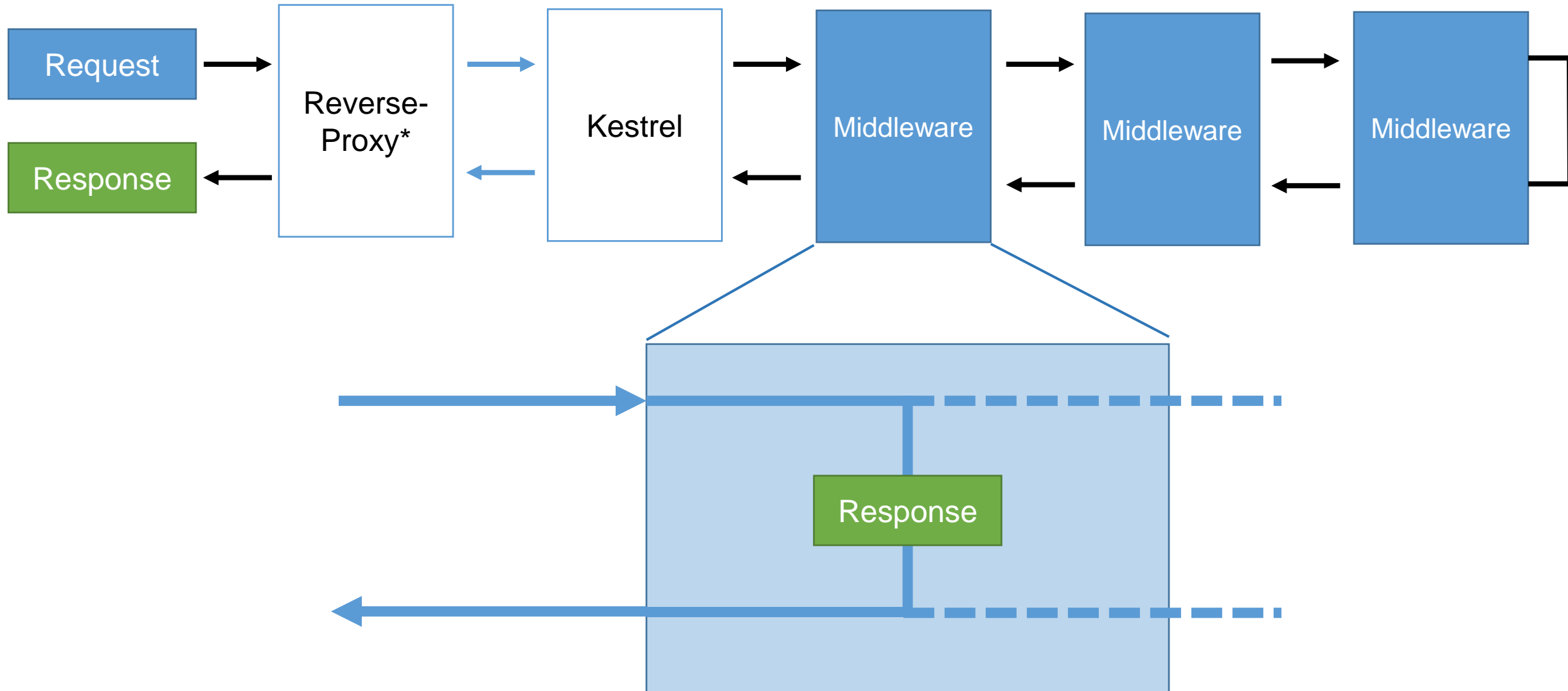
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

<https://docs.microsoft.com/de-de/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-3.1>

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 Core Features sind selbst als Middleware

WAS IST MIDDLEWARE?

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

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

Im Minimalfall muss die Configure-Methode angegeben werden. Die Parameter für Methoden und Konstruktor kommen aus dem Dependency Injection System

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Services die per Dependency Injection bereitgestellt werden sollen konfigurieren (Optional)
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env, ILoggerFactory loggerFactory)
    {
        // Http-Pipeline durch Middleware konfigurieren (Pflicht)
    }
}
```

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

Die Konfiguration macht selbst Gebrauch von Services die aus dem ASP.NET Core eigenen Dependency Injection (DI) System kommen

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Services die per Dependency Injection bereitgestellt werden sollen konfigurieren (Optional)
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env, ILoggerFactory loggerFactory)
    {
        // Http-Pipeline durch Middleware konfigurieren (Pflicht)
    }
}
```

Es wäre auch möglich nur den Parameter IApplicationBuilder der Configure-Methode zu verwenden, in der Regel werden aber auch die anderen eingesetzt

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

- In **ConfigureServices**
 - IServiceCollection
- In **Configure**
 - IApplicationBuilder
 - IWebHostEnvironment
 - ILoggerFactory
 - IHostApplicationLifetime

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

- **IApplicationBuilder**
 - Klasse für die Registrierung der zu verwendenden Middleware
- **IWebHostEnvironment**
 - Gibt Auskunft über die Umgebung in der die Applikation läuft
 - <https://docs.microsoft.com/de-de/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-3.1#ihostenvironment>
- **ILoggerFactory**
 - Dient der Konfiguration des ASP.NET Core eigenen Logging-Systems
 - Kann einfach durch andere Logging Libraries wie z.B. das hervorragende [Serilog](#) ersetzt werden

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

■ IHostApplicationLifetime

- Lifecycle Hooks der Application: ApplicationStarted, ApplicationStopping, ApplicationStopped
- Die Methode StopApplication() versucht die Applikation herunterzufahren
- <https://docs.microsoft.com/de-de/dotnet/api/microsoft.extensions.hosting.ihostapplicationlifetime?view=dotnet-plat-ext-3.1>

■ IServiceCollection

- Eine Kollektion an der für z.B. Middleware zusätzliche Komponenten für das DI-System registriert werden können
- In der Regel werden für jede komplexere Middleware zusätzliche Dienste benötigt damit die Middleware funktionieren kann

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?

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

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

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

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

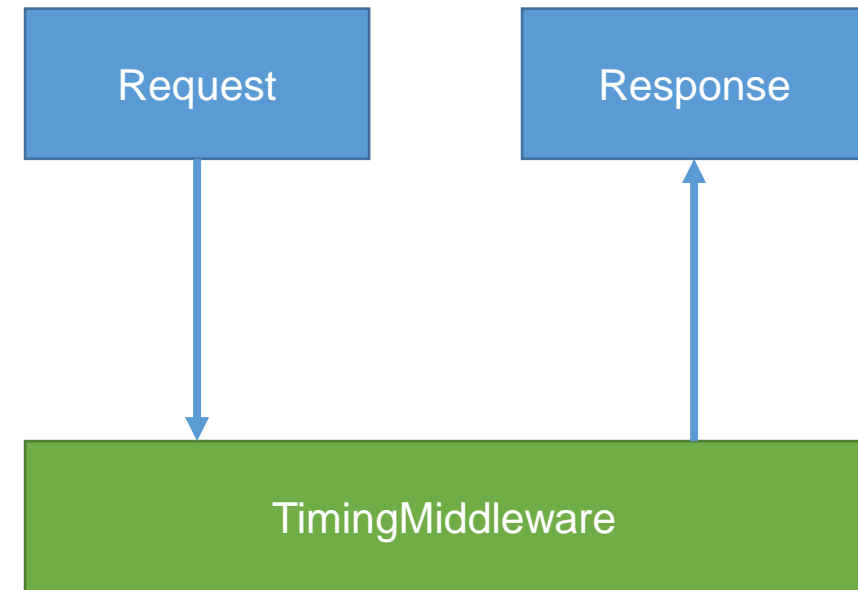
    // Short-Circuit Middleware
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

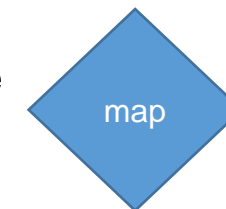
```
public void Configure(IApplicationBuilder app)
{
    // TimingMiddleware
    app.Use(async (context, next) =>
    {
        Stopwatch watch = Stopwatch.StartNew();
        await next();
        watch.Stop();
        Console.WriteLine($"Processing Duration: {watch.ElapsedMilliseconds} ms");
    });

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

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



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

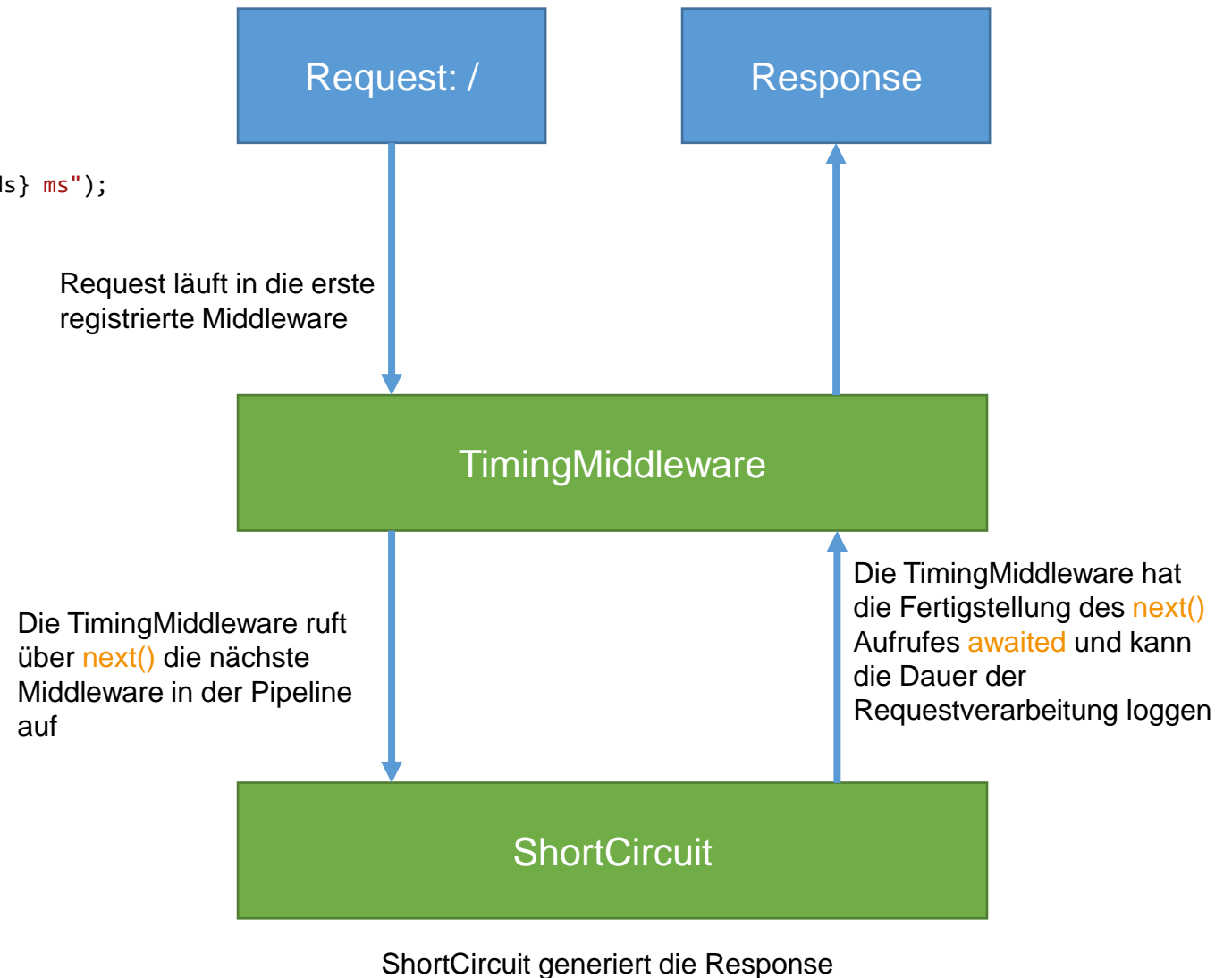


WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

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

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

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

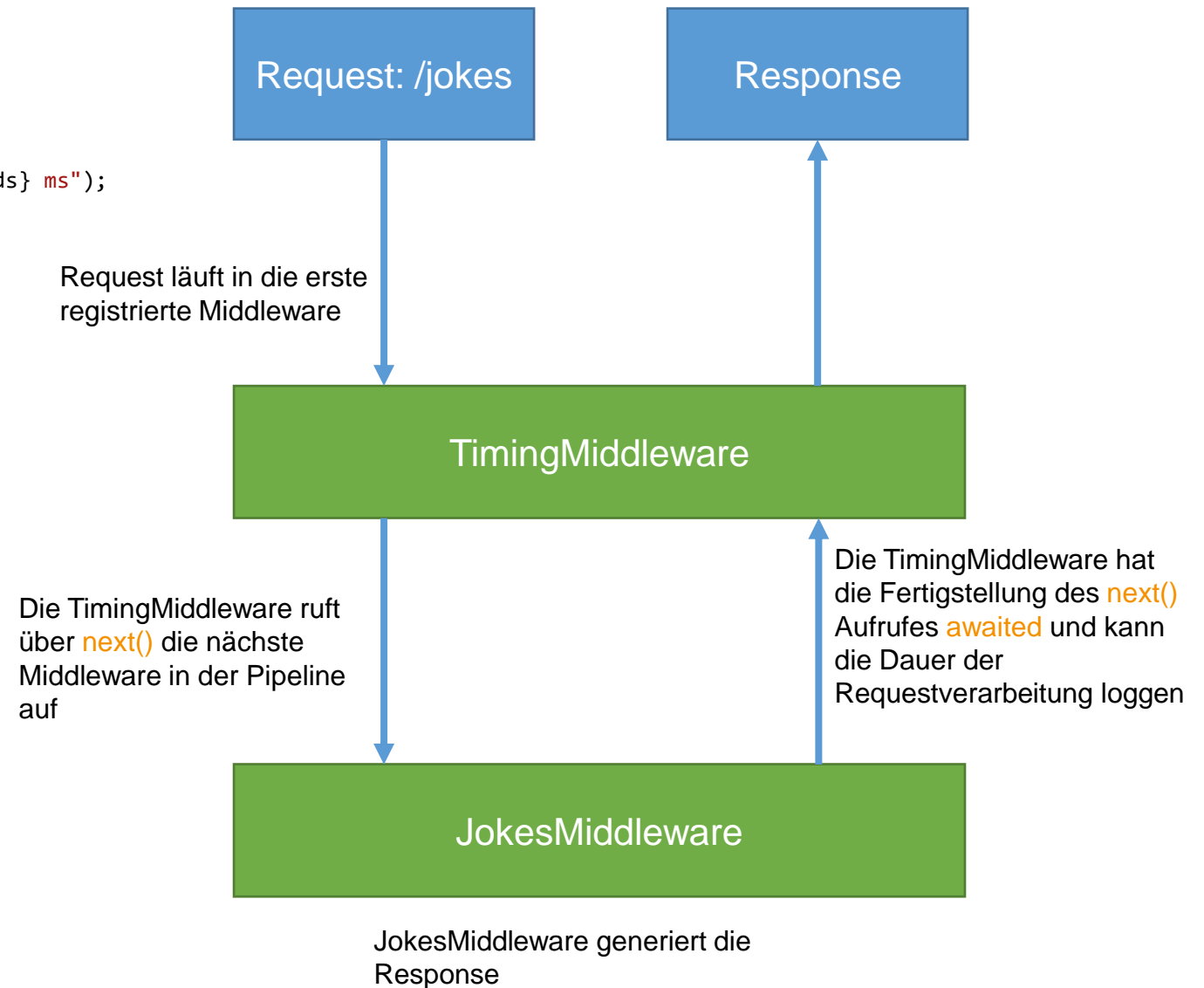


WIE WIRD DIE MIDDLEWARE KONFIGURIERT?

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

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

    // ShortCircuitMiddleware
    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>();
    }
}
```

```
public void Configure(IApplicationBuilder app)
{
    // ...
    app.UseTimingMiddleware();
    // ...
}
```

MIDDLEWARE ROUTING

MIDDLEWARE ROUTING

- Eine API mit Run und Use aufzubauen ist performant (Microservices) aber mühevoll
- Das modulare ASP.NET CORE 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.

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 Core 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.

WO WERDEN DIESE DIENSTE REGISTRIERT?

MIDDLEWARE MIT SERVICES REGISTRIEREN

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRouter(routes.Build());
    }
}
```


MIDDLEWARE MIT SERVICES REGISTRIEREN

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Dies fügt die von der Routing-Middleware benötigten
        // Dienste zum Dependency Injection Container hinzu.
        services.AddRouting();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseRouter(routes.Build());
    }
}
```

WIE KÖNNEN ROUTEN REGISTRIERT WERDEN?

ROUTING MIDDLEWARE

```
public void Configure(IApplicationBuilder app)
{
    // Der Route-Builder folgt dem "Builder-Pattern"
    // und dient der vereinfachten Definition von Routen
    var routes = new RouteBuilder(app);

    // HINWEIS: Wie bei der Middleware mit Use, Map, Run ist die Reihenfolge der Routen
    // wichtig. Die erste Definition die zutrifft wird verwendet.

}
```

ROUTING MIDDLEWARE

```
public void Configure(IApplicationBuilder app)
{
    // ...
    // string-Template das der eingehende Request matchen muss
    // um in den Routehandler zu gelangen
    routes.MapGet("api/weather/chicago", async context =>
    {
        // Response erstellen
    });

    // Routing unterstützt sogenannte Constraints, diese können den Request auf ein
    // bestimmtest Format einschränken, oder wie hier einen "Catch-All" definieren
    routes.MapGet("{*path}", context =>
    {
        // Response erstellen
    });
}
```

ROUTING MIDDLEWARE

```
public void Configure(IApplicationBuilder app)
{
    // die Routen werden der Routing-Middleware bei der Registrierung übergeben.
    app.UseRouter(routes.Build());
}
```

ROUTING MIDDLEWARE

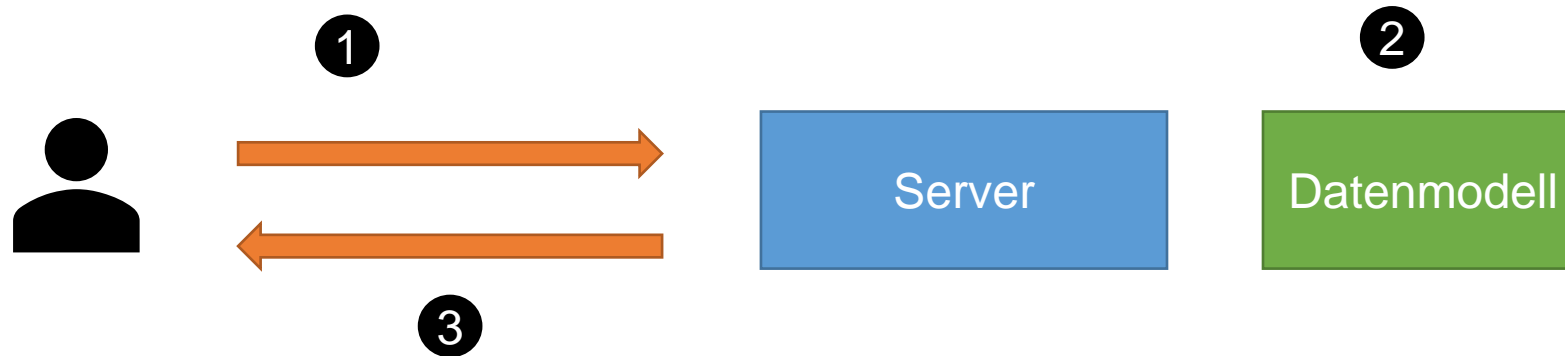
- Die Routing-Middleware bietet noch weitere Möglichkeiten
 - <https://docs.microsoft.com/de-de/aspnet/core/fundamentals/routing?view=aspnetcore-2.2#use-routing-middleware>
- Das Routing ist sehr performant, da kaum overhead durch das Asp.Net Core Framework vorhanden ist

ASP.NET CORE MVC

DAS MVC PATTERN

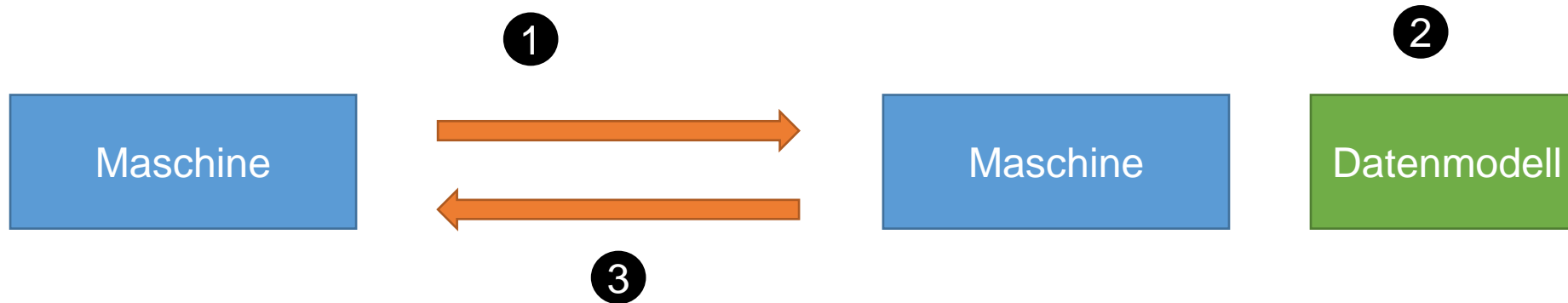
- Model
 - Repräsentation von Daten mit denen gearbeitet wird
 - Business Logik
 - Domain Model
 - View Model
- View
 - Rendert Teile der Daten als UI
- Controller
 - Verarbeitet eingehende Requests
 - Führt Operationen am Datenmodell durch
 - Selektiert den / die Views die erzeugt werden sollen

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 CORE MVC

- Routing mittels Middleware selbst aufzubauen ist anstrengend und fehlerträchtig
- Keine klare Trennung zwischen der URL und dem auszuführenden Code
- Dafür aber eine der schnellsten Möglichkeiten Request zu bedienen
- ASP.NET Core MVC ist auch „nur“ Middleware

KONFIGURATION ASP.NET CORE MVC

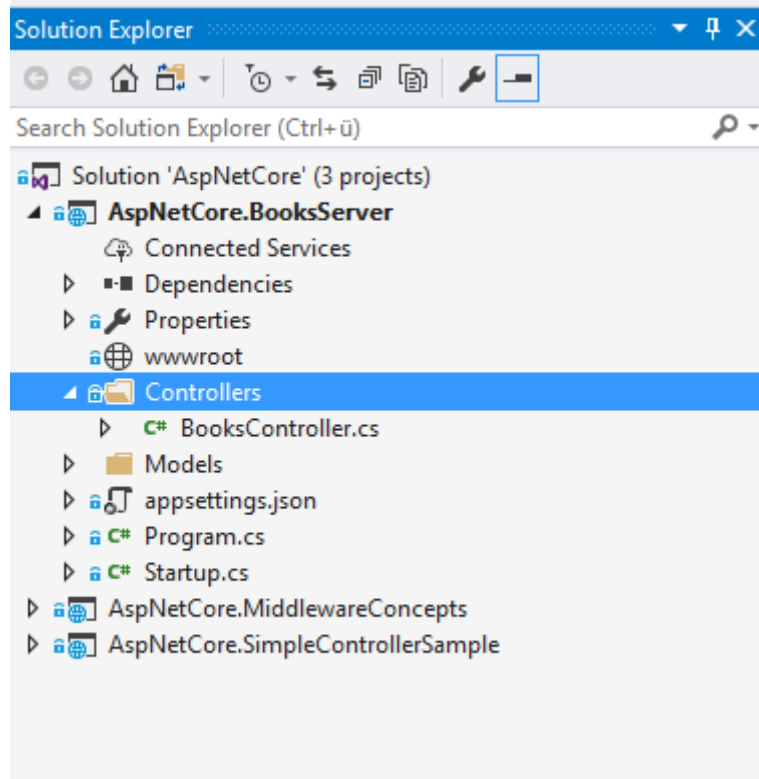
Middleware benötigt in der Regel auch einige Services, um die Funktionalität bereitstellen zu können und so auch MVC. Die ConfigureServices-Methode ist die Stelle, um diese am Dependency-Injection-System zu registrieren. Viele Middleware-Komponenten liefern spezielle Extension-Methoden um die Registrierung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

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    app.UseMvc();
```

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

ASP.NET CORE 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 CORE 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)
```

ATTRIBUTE 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

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 Core 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

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 class GreetingDto
{
    public string Greeting { get; set; }
}
```

```
public IActionResult PostGreeting(GreetingDto greeting)
{
}
```

1. Request

POST-Request enthält als Body ein JSON-Objekt

2. Model Binding

Die Web API hat als aufzurufende Controller Action die Methode **PostGreeting** ermittelt. Web API kennt den Parameter **greeting** und dessen Typ **GreetingDto**. Eine neue Instanz von **GreetingDto** wird erzeugt und der Model Binder setzt die Werte in das Objekt

3. Nutzung

Das vom Model Binder erzeugte Objekt kann nun verwendet werden

Model Binding

- Wandelt den Payload eines Requests in ein von der Controller Action gefordertes POOCO um
- 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
- Kein DataContract nötig wie bei WCF
- Es kann nur ein komplexes Objekt als Parameter einer Controller-Action geben
- 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).

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}.");
    }
}
```

ALTERNATIVE VALIDIERUNG MIT FLUENT VALIDATION

Grenzen von DataAnnotations

- Eigene Validatoren müssen durch neue Klassen umgesetzt werden
- Keine „Conditional“-Validierung möglich
- „Verunreinigen“ das Datenmodell
 - Was ist wenn es fachliche Fälle gibt, in welchen andere Validierungsregeln greifen sollen
- Keine Abhängigkeiten zwischen Validatoren möglich
 - „Validiere diese Eigenschaft x wenn Eigenschaft y den Wert z hat“
- Keine asynchronen Validatoren
 - Z.B. Aufruf eines Webservices ob der Wert gültig ist
- DataAttributes sind statisch, keine Konfiguration über z.B. Settings möglich

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 Core ein
 - Mittels FluentValidation.AspNetCore
- Gute Unit-Test-Unterstützung
- Wiederwendbare Property Validatoren

Fluent Validation

```
[Route("api/[controller]")]
public class StudentsController : ControllerBase
{
    [HttpPost]
    public IActionResult PostStudent([FromBody] Student student)
    {
        if (ModelState.IsValid)
        {
            // SAVE
            return Ok(student);
        }

        return BadRequest(ModelState);
    }
}
```

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 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");
        });
    }
}
```

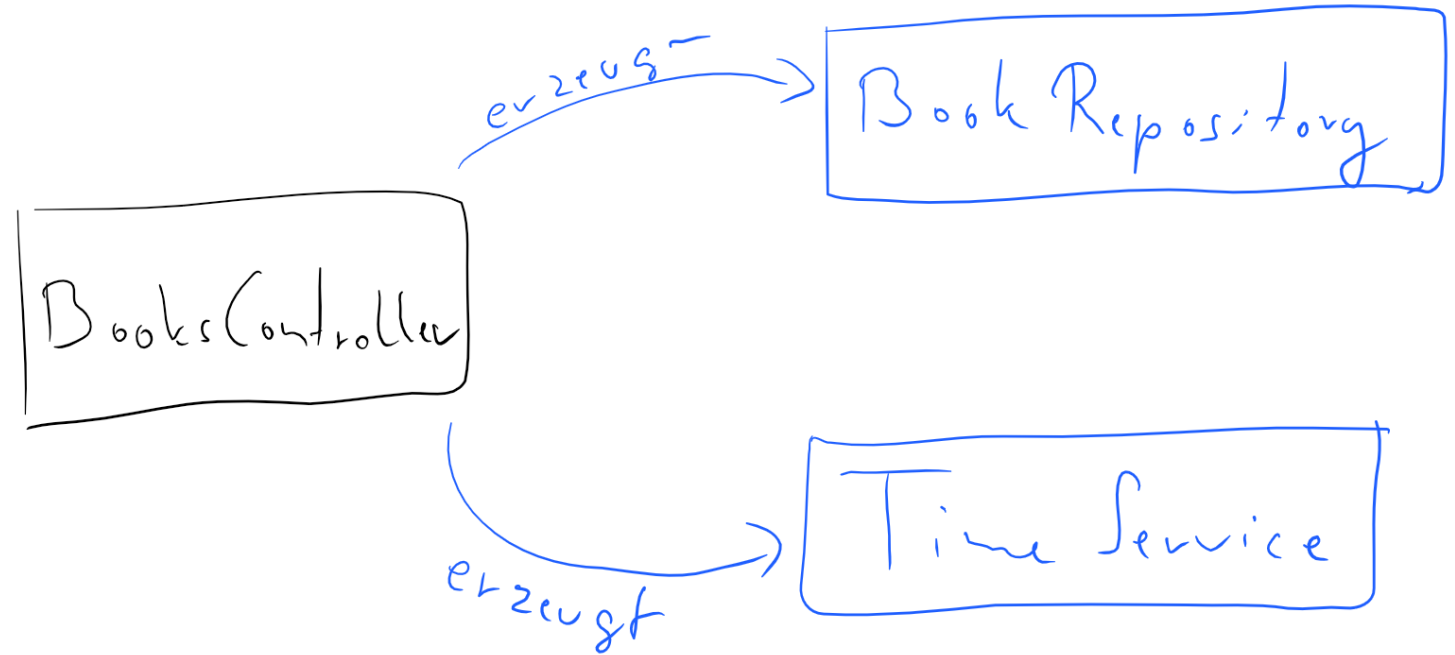
Fluent Validation

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
        .AddFluentValidation();

    services.AddTransient<IValidator<Student>, StudentValidator>();
}
```

DEPENDENCY INJECTION

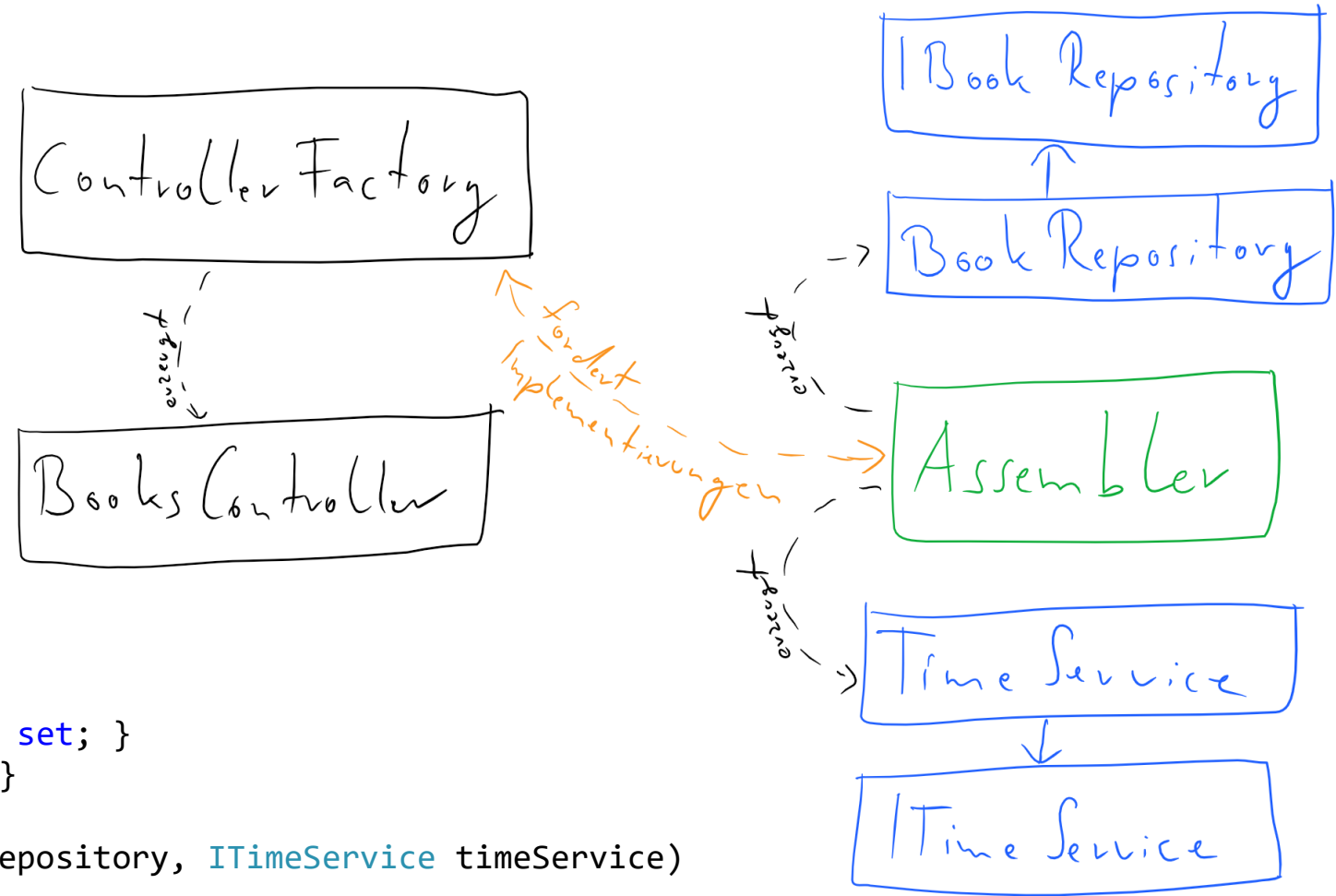
Dependency Injection| Worum geht's?



```
public class BooksController : ControllerBase
{
    public IBookRepository BookRepository { get; set; }
    public ITimeService TimeService { get; set; }

    public BooksController()
    {
        BookRepository = new InMemoryBookRepository();
        TimeService = new DefaultTimeService();
    }
}
```

Dependency Injection| Worum geht's?



```
public class BooksController : ControllerBase
{
    public IBookRepository BookRepository { get; set; }
    public ITimeService TimeService { get; set; }

    public BooksController(IBookRepository bookRepository, ITimeService timeService)
    {
        BookRepository = bookRepository;
        TimeService = timeService;
    }
}
```

Service Locator vs. Dependency Injection

```
public class BooksController : ControllerBase
{
    public IBookRepository BookRepository { get; set; }
    public ITimeService TimeService { get; set; }

    public BooksController()
    {
        BookRepository = ServiceLocator.Resolve<IBookRepository>();
        TimeService = ServiceLocator.Resolve<ITimeService>();
    }
}
```

```
public class BooksController : ControllerBase
{
    public IBookRepository BookRepository { get; set; }
    public ITimeService TimeService { get; set; }

    public BooksController(IBookRepository bookRepository, ITimeService timeService)
    {
        BookRepository = bookRepository;
        TimeService = timeService;
    }
}
```

Service Locator

Global zugängliches Objekt das nach Service Implementierungen gefragt werden kann.

BooksController fragt explizit nach Implementierungen der benötigten Services.

DI über Constructor Injection

Die Abhängigkeiten sind klar als Constructor Parameter kenntlich.

Die benötigten Services „tauchen“ in der Klasse auf (Inversion of Control)

Service Locator vs. Dependency Injection

- Service Locator hat den Ruf ein „Anti-Pattern“ zu sein. Richtig implementiert kann es aber sehr nützlich sein.
- Dependency Injection lässt macht die Abhängigkeiten (Dependencies) bei der Verwendung von Constructor Injection klar ersichtlich. Das erleichtert das Testen.

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 Core 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

```
public void ConfigureServices(IServiceCollection services)
{
    // Services für MVC registrieren
    services.AddMvc();

    // Pro Request wird ein neuer Bookservice erzeugt
    services.AddScoped<IBookRepository, InMemoryBookRepository>();

    // Ein Singleton bleibt für die Applikationslaufzeit das selbe Objekt
    services.AddSingleton<ITimeService, DefaultTimeService>();

    // Transient bedeutet bei Anfrage an das DI-System erhält man ein neues Objekt
    services.AddTransient<NewObjectWithEveryRequest>();
}
```

Es kann auch ein konkreter Typ (z.B. Konstanten) 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 Core 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();  
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 Core 2.2+ führt `CreateDefaultBuilder` automatisch zum Laden der Konfiguration aus typischen Locations (appsettings.json, Environment Variables)

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...
Azure Key Vault Configuration Provider (<i>Security topics</i>)	Azure Key Vault
Command-line Configuration Provider	Command-line parameters
Custom configuration provider	Custom source
Environment Variables Configuration Provider	Environment variables
File Configuration Provider	Files (INI, JSON, XML)
Key-per-file Configuration Provider	Directory files
Memory Configuration Provider	In-memory collections
User secrets (Secret Manager) (<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

Zugriff auf die Konfiguration kann
untypisiert oder typisiert sein

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

```
var featureConfiguration = Configuration.GetSection("Features").Get<FeatureOptions>();
```

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

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

CONFIGURATION

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

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<FeatureOptions>(Configuration.GetSection("Features"));
}
```

DEPENDENCY INJECTION DER OPTIONS

IOptionsSnapshot<T> ist eine Möglichkeit eine aktuelle Repräsentation der Optionen im Controller zu verwenden.

```
public class ValuesController : ControllerBase
{
    private readonly IOptionsSnapshot<FeatureOptions> _featureOptions;

    public ValuesController(IOptionsSnapshot<FeatureOptions> featureOptions)
    {
        _featureOptions = featureOptions;
    }
}
```