

A decorative graphic on the left side of the slide, consisting of a 5x5 grid of squares in various shades of gray and blue, creating a subtle gradient effect.

.NET: ASP.NET Core

© J. Heinzlreiter
Version 2.1

Überblick

- Ausgangssituation und Merkmale
- Architektur
- Projektstruktur und Konfiguration
- Middleware
- Implementierung von Web-Anwendungen
- Neuerungen in MVC
 - Tag-Helpers
 - Razor-Pages
- Unterstützung für Single-Page-Webanwendungen

ASP.NET Core – Ausgangssituation

- Bestehender Web-Stack ist veraltet
 - Teile stammen aus Version 1 des .NET-Frameworks (System.Web.dll).
 - Komponenten werden geladen, obwohl sie nicht benötigt werden.
- .NET-Upgrades auf Serverseite können problematisch sein.
 - Alle Web-Anwendungen sind von .NET-Version des Servers abhängig.
- Schlechte Laufzeiteffizienz
 - Limitierungen durch Architektur der Plattform
- Starke Abhängigkeit der Komponenten und unzureichende Erweiterungsmöglichkeiten.
 - ASP.NET war auf Deployment auf dem IIS ausgerichtet.
 - Erweiterungen wurden immer häufiger notwendig: Web API, SignalR, Self-Hosting etc.
- Ähnliche Funktionalität in ASP.NET MVC und Web-API wurde mehrfach implementiert: Routing, Controller, ...

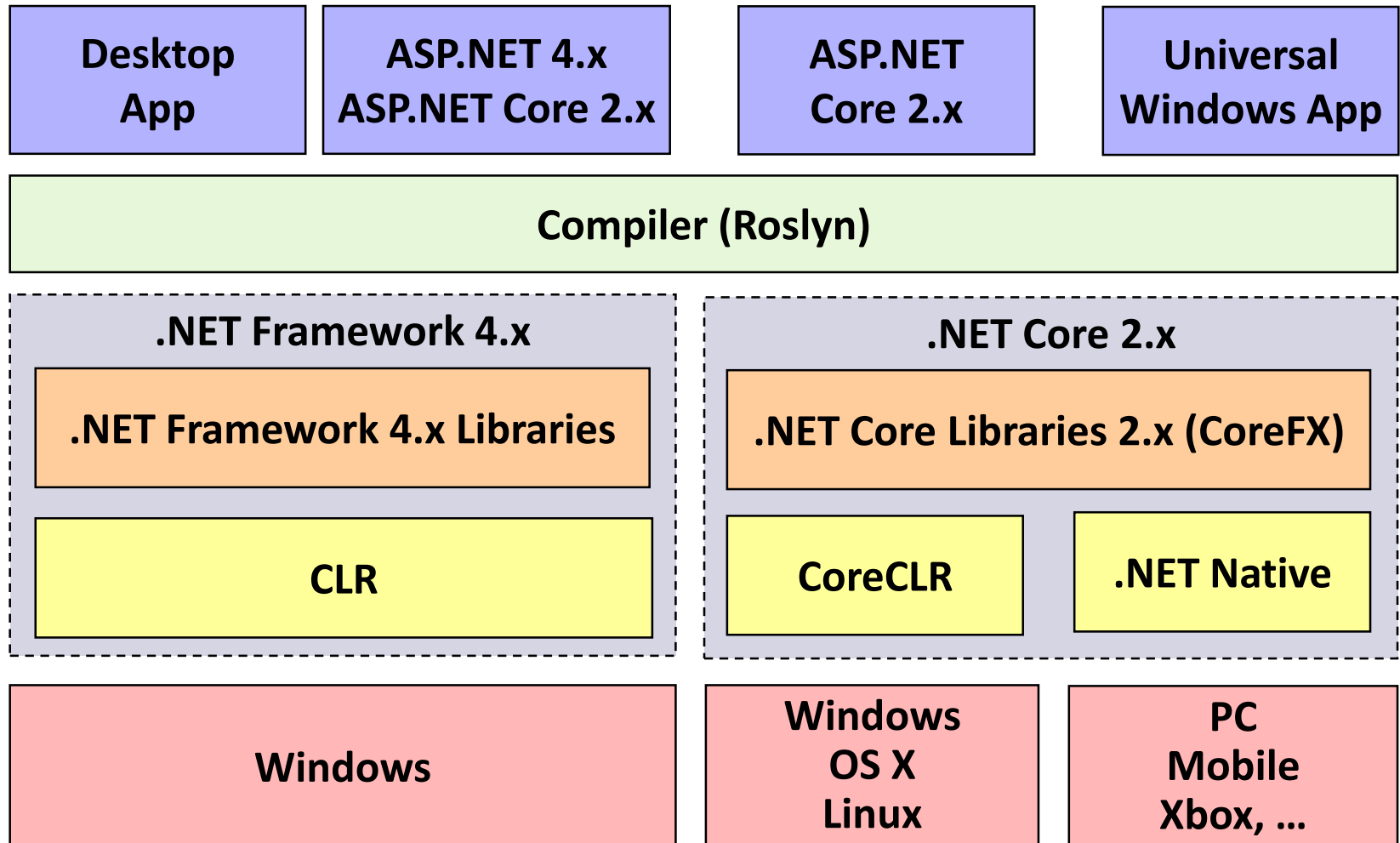
getrennte Komponenten - aber gemeinsame Funktionalitäten

Codeduplizierung

ASP.NET Core – Merkmale

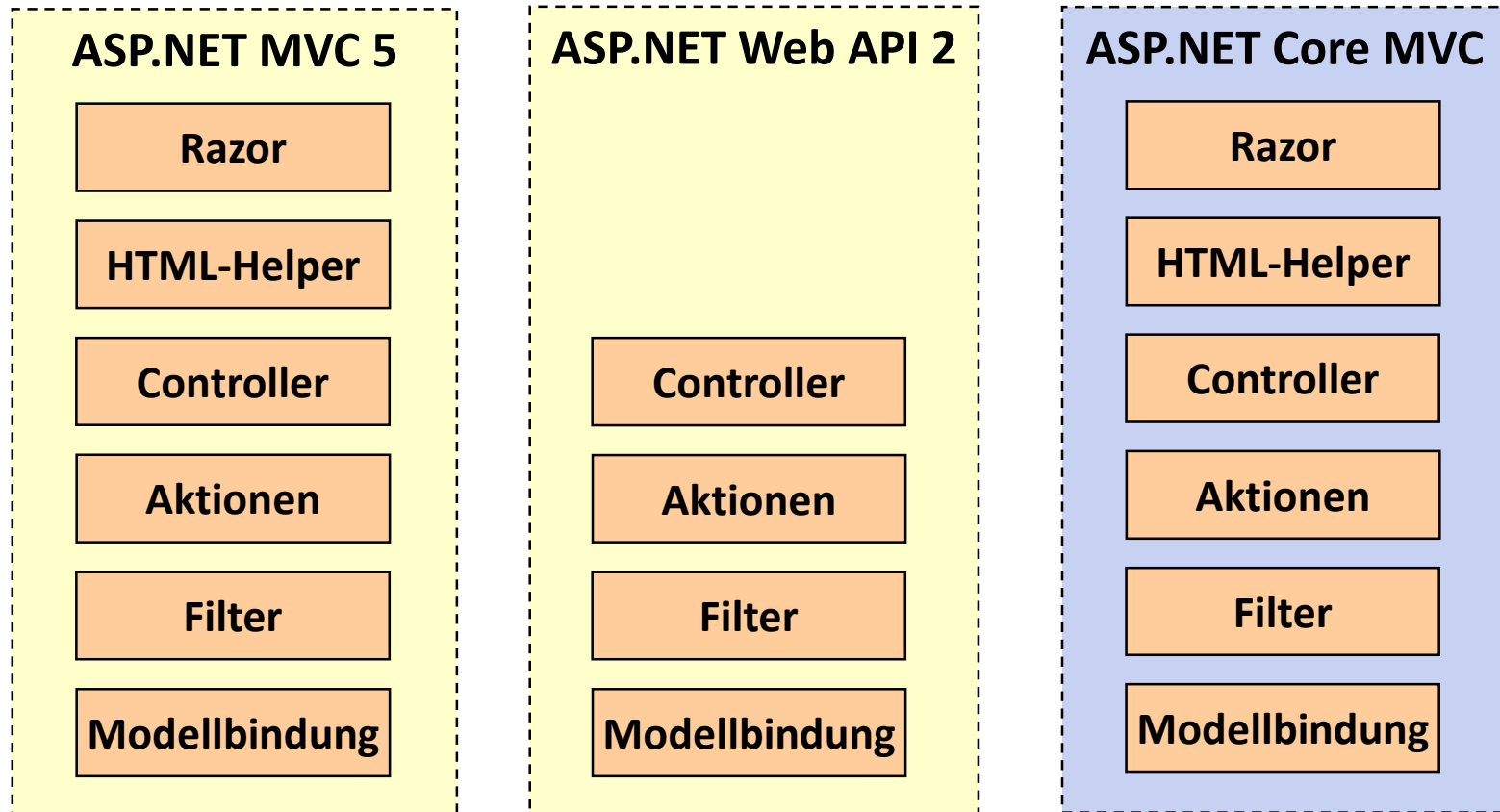
- Vollständige Neuimplementierung der Plattform in Form eines Open-Source-Projekts.
 - Basiert auf .NET Core.
 - Nur Konzepte werden übernommen.
- Plattformunabhängigkeit: Windows, OS X, Linux
- Modularer Aufbau
 - Alle Komponenten müssen explizit hinzugefügt werden.
 - Komponenten werden als Nuget-Pakete ausgeliefert.
- Berücksichtigung aktueller Trends in der Web-Entwicklung
 - Single-Page-Webanwendungen,
 - Werkzeuge für Web-Entwicklung.
- Unterstützung verschiedener Deploymentarten
- Web-Forms wird nicht unterstützt
 - Wird aber als Bestandteil des .NET-Frameworks fortgeführt.

Die .NET-Plattform



ASP.NET vs. ASP.NET Core

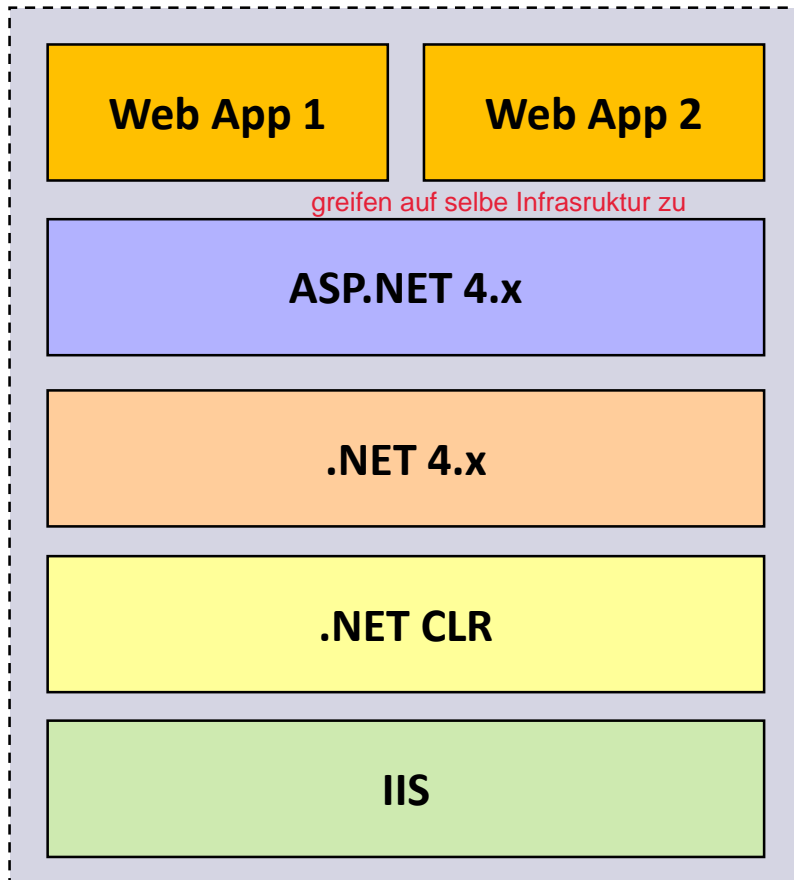
- ASP.NET Core MVC ist *ein* Framework für die Entwicklung von Web-Anwendungen *und* REST-Services.



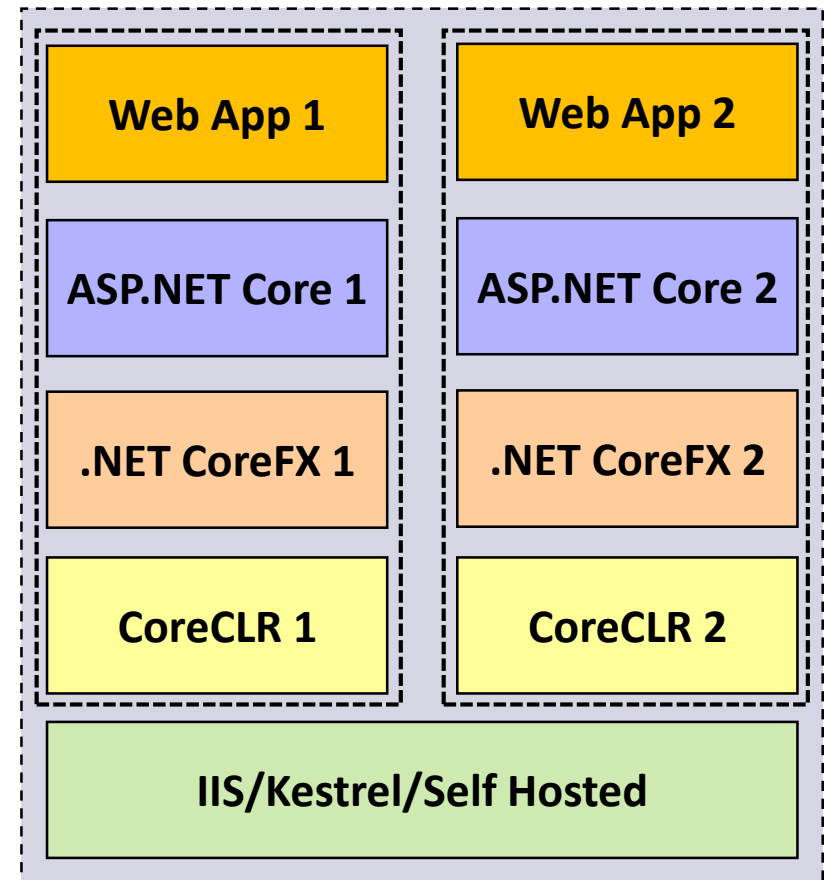
Deployment von Web-Anwendungen

self contained deployment

.NET 4.6: Framework-Komponenten sind im GAC installiert.



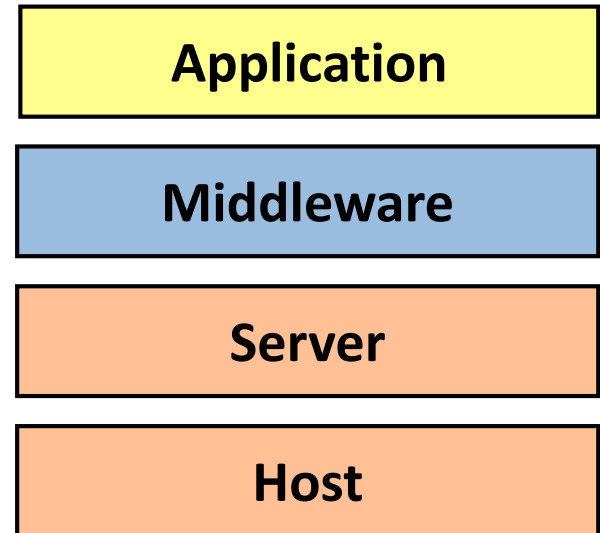
.NET Core: Framework-Komponenten werden mit Web-Anwendung deployt.



Architektur

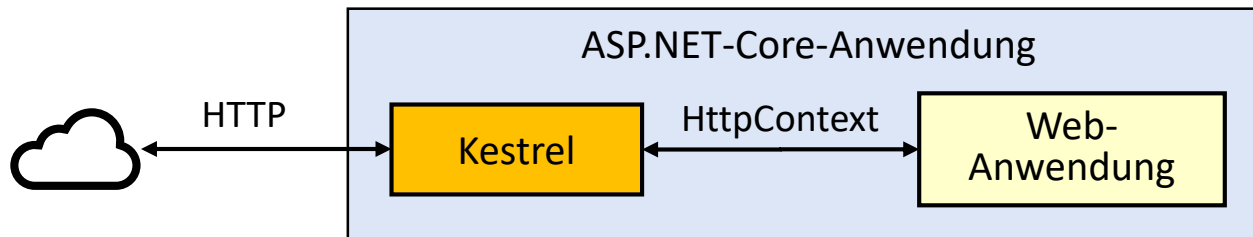
- **Host:** Orchestrierung der Request-Pipeline.
 - Kann vom Web-Server übernommen werden.
 - Self-Hosting z. B. in Konsolenanwendung
- **Server:** Horchen auf Requests und Weiterleiten an Pipeline.
 - IIS und IIS Express
 - Http.sys (WebListener)
 - Kestrel: plattformunabhängiger Server
- **Middleware:** Komponente der Pipeline, die Requests verarbeitet
 - Erhält Request/Response-Kontext
 - Gibt Kontrolle an nachgelagerte Middleware-Komponente weiter.
 - Wird in *Startup*-Klasse konfiguriert.
- **Application:** Web-, SignalR-Anwendung, REST-Service, ...

gegenstück zu tomcat

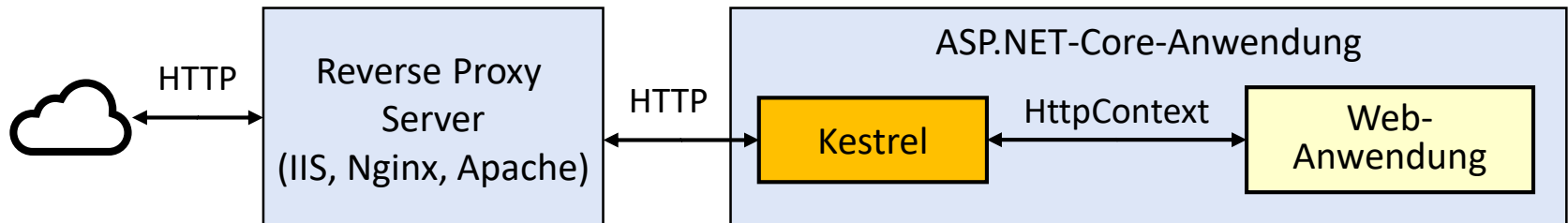


Einbindung von Web-Servern

- Kestrel: Plattformunabhängiger Standard-Server in .NET Core



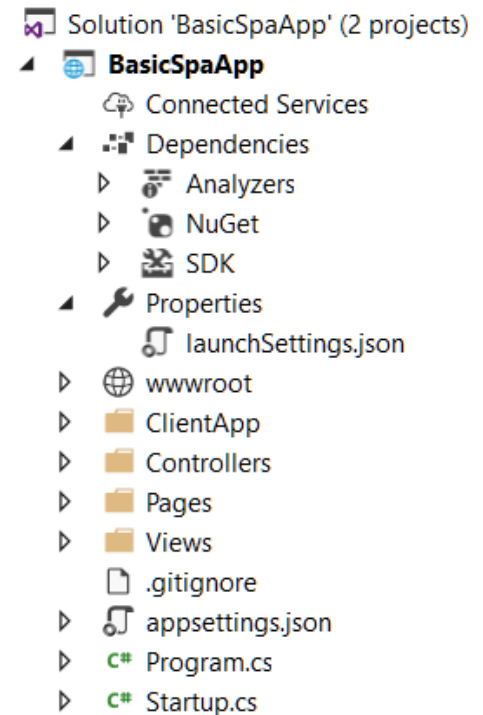
- Kestrel mit einem Reverse Proxy weil der request von außen kommt - reverse



- Port-Sharing: Mehrere Webanwendungen teilen sich eine IP-Adresse und einen Port.
- Bessere Integration in bestehende Infrastruktur.
- Nur Reverse Proxy benötigt SSL-Zertifikat.
- Zentrale Authentifizierung.

Implementierung einer Web-Anwendung

- Erstellung eines Projekts
 - .NET CLI (Command Line Interface)
 - Projektvorlage in Visual Studio
- Projektstruktur
 - **.csproj*: Enthält Projekteinstellungen und Definition der serverseitigen Abhängigkeiten (Frameworks, verwendete Komponenten) → XML-Format
 - Abhängigkeitsverwaltung
 - .NET-Komponenten: NuGet (*.csproj)
 - *wwwroot*: Wurzelverzeichnis der Anwendung aus Sicht des Web-Servers
 - *ClientApp*: Single-Page-Webanwendung
 - *Controllers/Pages/Views*: Serverseitige Web-Anwendung
 - *Startup.cs*: Konfiguration der Anwendungskomponenten und der Middleware
 - *Programm.cs*: Konfiguration des Hosts.



Projektkonfiguration: Aufbau der Projektdatei (.csproj)

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFrameworks>netcoreapp2.0;net462</TargetFrameworks>
  </PropertyGroup>
  <PropertyGroup>
    <ItemGroup>
      <PackageReference Include="Microsoft.AspNetCore.App" />
      <PackageReference Include="Microsoft.AspNetCore.Razor.Design"
        Version="2.1.2" />
    </ItemGroup>
    <ItemGroup>
      <DotNetCliToolReference
        Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.3" />
    </ItemGroup>
  </PropertyGroup>
</Project>
```

- *TargetFrameworks*: „Multi-Targeting“ → Gleichzeitige Unterstützung mehrere Frameworks. Derzeit kein IDE-Support.
- *PackageReference*: Referenzen auf NuGet-Pakete
- *DotNetCliToolReference*: Erweiterung der CLI um zusätzliche Entwicklertools (z. B. dotnet ef)

Konfiguration des Hosts

- Der *Host* ist für Konfiguration und das Starten der Web-Anwendung bzw. des Servers verantwortlich.
- Der *Server* verarbeitet die HTTP-Anfragen.

```
public static void Main(string[] args) {  
    var host =  
        new WebHostBuilder()  
            .UseKestrel()  
            .UseContentRoot(  
                Directory.GetCurrentDirectory())  
            .UseIISIntegration()  
            .UseStartup<Startup>()  
            .Build();  
    host.Run();  
}
```

} WebHost.
CreateDefaultBuilder(args)

- UseKestrel: Kestrel wird als Web-Server verwendet.
- UseIISIntegration: IIS wird als *Reverse Proxy* verwendet.
- UseStartup: Definition der Klasse, die Methoden zur Konfiguration der Web-Anwendung enthält.

Konfiguration der Komponenten

- Konfiguration erfolgt in Klasse *Startup*.
 - Der Name dieser Klasse wird im Host festgelegt.
- Methode *ConfigureServices*: Konfiguration der Services, die mit Abhängigkeitsinjektion an Klassen der Web-Anwendung übergeben werden können.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MyDbContext>(options =>
        options.UseSqlServer(connectionString));
    services.AddScoped<ILogic, BusinessLogicImpl>();
}
```

- Methode *Configure*: Konfiguration der Middlewarekomponenten

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env) {
    app.UseFacebookAuthentication(...);
    app.UseCors(...);
    if (env.IsDevelopment()) app.UseDeveloperExceptionPage();
}
```

Konfiguration der Services

- „Services“ in ASP.NET Core:
 - Austauschbare Komponenten: Interface und Implementierung
 - Infrastruktur- und anwendungsbezogene Services
- Implementierung eines Service:

```
public interface IService {  
    void myServiceMethod();  
}
```

```
public class MyService : IService { ... }
```

- Hinzufügen zum Service-Container:

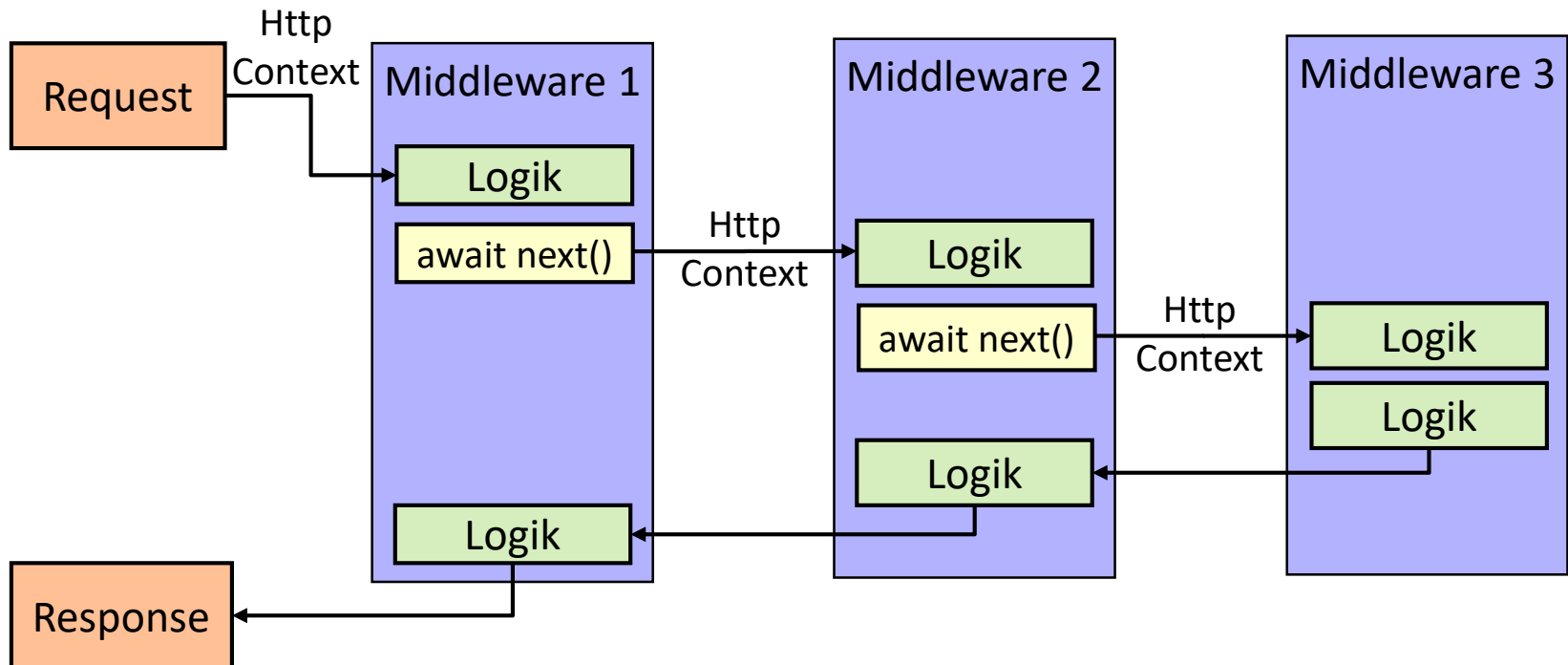
```
public void ConfigureServices(IServiceCollection services)  
    services.AddScoped<IService, MyServiceImpl>();  
}
```

- Injektion der Abhängigkeiten im Container:

```
public class MyController {  
    private IService myService;  
    public MyController(IService myService) { this.myService = myService; }  
}
```

Middleware – Konzept

- Middleware-Komponenten verarbeiten Requests und können Resultate zur Response hinzufügen.
- Middleware-Komponenten werden in der Konfiguration zu einer Kette zusammengefügt.



Implementierung von Middleware-Komponenten

```
public delegate Task RequestDelegate(HttpContext context);  
public class MyMiddleware {  
    private readonly RequestDelegate next;  
    public MyMiddleware(RequestDelegate next) { this.next = next; }  
    public async Task Invoke(HttpContext context) {  
        // process context.Request  
        await next(context);  
        // process context.Response  
    }  
}
```

```
public static class MyMiddlewareExtensions {  
    public static void UseMyMiddleware(this IApplicationBuilder builder) {  
        builder.UseMiddleware<MyMiddleware>();  
    }  
}
```

```
public class Startup {  
    public void Configure(IApplicationBuilder app) {  
        app.UseMyMiddleware();  
    }  
}
```


Konfiguration einer Web-Anwendungen

- Einfachste Web-Anwendung: Registrierung eines Callbacks, das für jedes Request aufgerufen wird.

```
public class Startup {  
    public void Configure(IApplicationBuilder app) {  
        app.Run(async context => {  
            await context.Response.WriteAsync("Hello ASP.NET Core");  
        });  
    }  
}
```

- Verarbeitung statischer Dateien
 - Dateien müssen sich in *wwwroot* befinden

```
public class Startup {  
    public void Configure(IApplicationBuilder app) {  
        app.UseDefaultFiles();  
        app.UseStaticFiles();  
    }  
}
```

Implementierung der ASP.NET-MVC-Anwendung

- Konfiguration

```
public class Startup {  
    public void ConfigureServices(IServiceCollection services) {  
        services.AddMvc();  
    }  
    public void Configure(IApplicationBuilder app) {  
        app.UseMvc(routes => {  
            routes.MapRoute(  
                name: "default",  
                template: "{controller=Home}/{action=Index}/{id?}");  
            });  
        }  
    }  
}
```

- `services.AddMvc()` fügt alle für MVC notwendigen Komponenten in den Service-Container.
- Implementierung der Controller
- Implementierung der Views

Werkzeuge zur plattformunabh. Softwareentwicklung

- `dotnet` ist ein plattformunabhängiges Kommandozeilenwerkzeug zur Entwicklung von .NET-Core-Anwendungen.
- `dotnet new`: Erzeugung eines neuen Projekts.
 - Beispiel: `dotnet new mvc --lang C#`
 - Auch *Yeoman* ist unterstützt: `yo aspnet`
- `dotnet restore`: Herunterladen der NuGet-Pakete.
 - Wird das Kommando auf der Ebene der Solution ausgeführt, werden die Pakete aller Projekte heruntergeladen.
- `dotnet restore`: Bauen des Projekts und seiner Abhängigkeiten.
 - Beispiel: `dotnet build --framework net46 --runtime win10-x64`
- `dotnet run`: Ausführen des Programms (Web: Starten des Hosts).
 - Beispiel: `dotnet run --framework net46`
- `dotnet publish`: Verpacken der Anwendung inklusive der Abhängigkeiten und Kopieren in ein Zielverzeichnis.
 - Beispiel: `dotnet publish --framework netcoreapp2.1 --output ~/target`

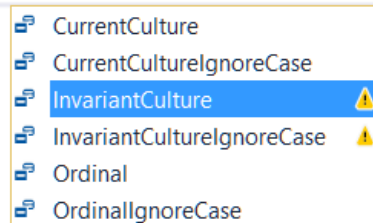
Unterstützung mehrerer Frameworks

- Eine .NetCore-Anwendungen können parallel für mehrere Frameworks entwickelt werden.

```
<TargetFrameworks>  
  netcoreapp2.1;net472  
</TargetFrameworks>
```

- Visual Studio überprüft die Kompatibilität mit allen Frameworks.

```
if (env.Equals("Development",  
              StringComparison.
```

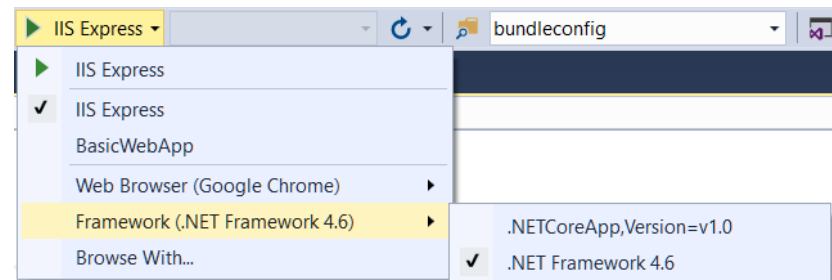


StringComparison.InvariantCulture = 2
Compare strings using culture-sensitive sort rules and the invariant culture.

BasicWebApp..NET Framework 4.6 - Available
BasicWebApp..NETCoreApp,Version=v1.0 - Not Available

You can use the navigation bar to switch context.

- .NET Core \subset .NET-Framework
→ für gewisse Funktionen muss Ersatz gesucht werden.
- Zu verwendendes Framework kann beim Starten der Anwendung angegeben werden:
 - `dotnet run --framework net472`



Tag-Helper

- Problem bei HTML-Hilfsmethoden:

```
@Html.EditorFor(model => model.Name,  
    new { htmlAttributes = new { @class = "form-control" } })
```

- C#/Razor-Code wird mit Markup-Code vermischt →
 - schwer zu lesen
 - Code-Vervollständigung nicht verfügbar

- HTML-Helper

- Verwendung:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

```
<input asp-for="Name" class="form-control" />
```

- Typsicherheit bleibt erhalten
- Es können auch benutzerdefinierte Tag-Helper implementiert werden.

```
<input asp-for="" class="form-control" />
```

- Equals
- GetHashCode
- GetType
- Name**
- ToString

```
string Person.Name { get; set; }
```

Tag-Helper: Beispiele

- Sichere Generierung von URLs, Zugriff auf das Modell

```
<a asp-action="Index" asp-controller="Home">Back to Home page</a>
<form asp-action="Add" asp-controller="Person">
  <label asp-for="Name"></label>
  <input type="text" asp-for="Name" />
  <span asp-validation-for="Name"></span>
</form>
```

- Bedingte Generierung von HTML-Code

```
<environment names="Development">
  <link rel="stylesheet" href="~/css/bootstrap.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/.../bootstrap.min.css"/>
</environment>
```

- Generierter Code ist von Umgebungsvariable **ASPNET_ENV** abhängig.

Razor Pages (1)

- Umsetzung von HTML-zentrierten Web-Seiten wird vereinfacht.
- Programmiermodell:
 - Seite besteht aus
 - Razor-View (*MyPage.cshtml*) und
 - aus einer Page-Klasse (*MyPage.cshtml.cs*)
 - Page-Klasse implementiert Callback-Methoden, die mit HTTP-Verbs assoziiert sind: `OnGet[Asnyc]`, `OnPost[Asnyc]`, ...
- Beispiel (View):

```
@page "{id:int}"
@model EditPersonModel

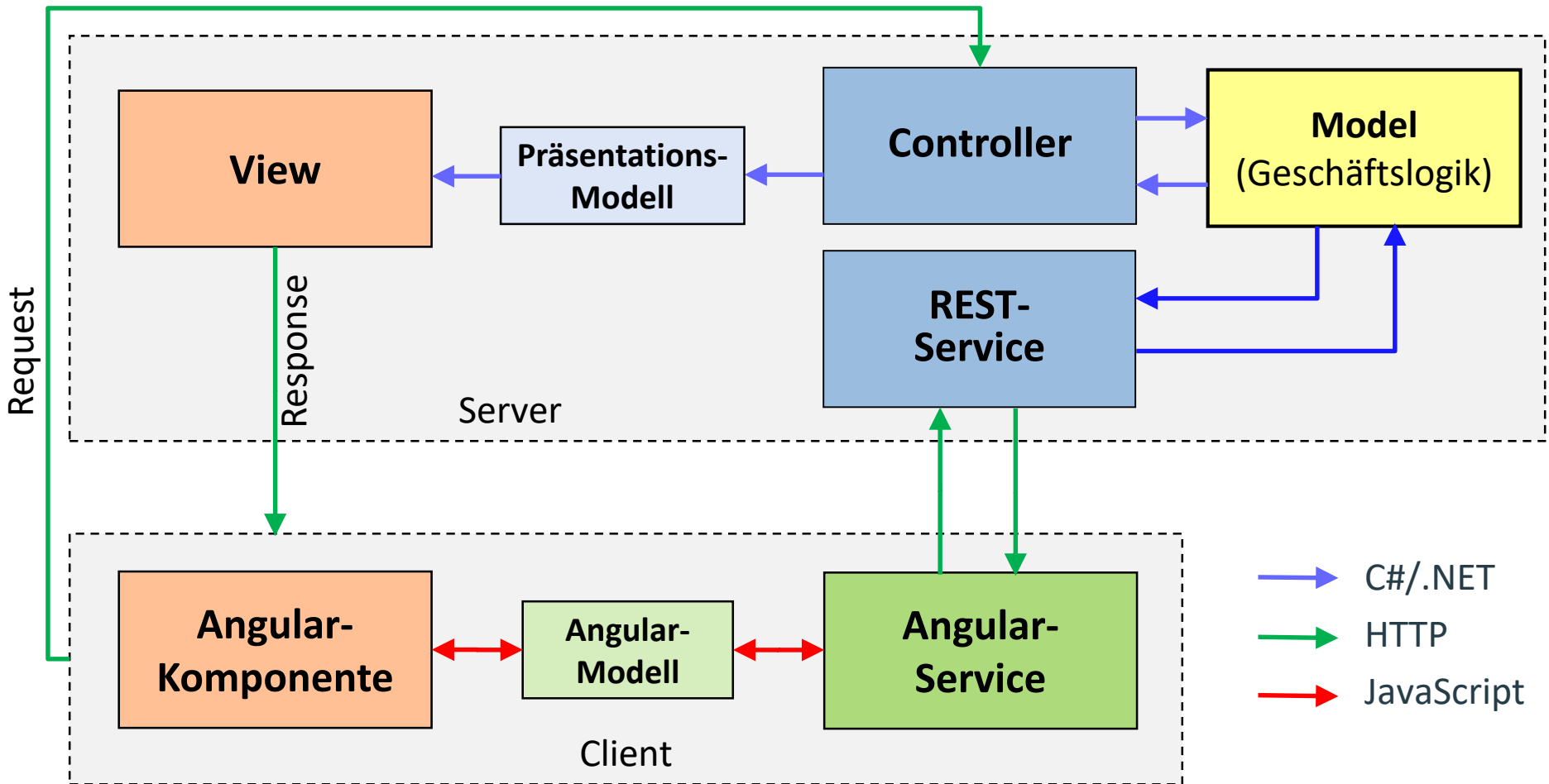
...
<div>
    <input asp-for="Person.Name" />
</div>
...
```

Razor Pages (2)

- Beispiel (Code-Behind):

```
public class EditPersonModel :  
    Microsoft.AspNetCore.Mvc.RazorPages.PageModel {  
    [BindProperty]  
    public PersonData Person { get; set; }  
    public async Task<IActionResult> OnGetAsync(int id) {  
        Person = await Logic.FindPersonAsync(id)  
        return Page()  
    }  
    public async Task<IActionResult> OnPostAsync() {  
        if (!ModelState.IsValid)  
            return Page();  
        await Logic.UpdatePersonAsync(Person);  
        return RedirectToPage("/Person");  
    }  
}
```


Architektur von „Single-Page-Webanwendungen“



Entwicklung von Single-Page-Webanwendungen

- SPA und ASP.NET-Core-Anwendung (Backend) kann gemeinsam entwickelt werden.
- Eine Middleware-Komponente unterstützt die Entwicklung von SPAs.

```
public void Configure(IApplicationBuilder app) {  
    app.UseSpaStaticFiles();  
    app.UseSpa(spa => {  
        spa.Options.SourcePath = "ClientApp";  
        if (env.IsDevelopment())  
            spa.UseAngularCliServer(npmScript: "start");  
    });  
}
```

- „Development“-Modus (Start in Visual Studio)
 - → **ng serve** (npm start)
- Deployment(dotnet publish)
 - → **ng build --prod**

