

ASP.NET Core basis

Table of Contents

1.	Inleiding ASP.NET Core Framework	2
	ASP.NET Core 3.1:	2
	Wat is .NET Core?	2
	Waarom .NET Core?	2
	Waarvoor kunnen we .NET Core apps ontwikkelen?	3
	Welke types van apps kunnen we ontwikkelen met .Net Core?	3
	ASP.NET Core 3.1:	3
	Wat is ASP .NET Core?	3
2.	.NET Core en ASP.NET Core Installatie	4
	a. Installatie van Visual Studio 2017/2019:	4
	Download en installeer Visual Studio	4
	b. Download and install .NET Core SDK	6
	Er zijn 3 mogelijke opties die je kan downloaden en installeren:	6
	Verifieer de installatie	7
3.	Aanmaken van een ASP.NET Core Web Applicatie	8
	Creëren van een ASP.NET Core Web Application in Visual Studio 2019	8
	Project templates in ASP.NET Core Application	11
4.	ASP.NET Core Project File	14
5.	ASP.NET Core Main Methode	18
6.	ASP.NET Core InProcess Hosting	21
7.	Kestrel Web Server in ASP.NET Core	25
8.	ASP.NET Core launchSettings.json file	31
9.	ASP.NET Core Startup Class	35
10.	ASP.NET Core appsettings.json file	38
11.	ASP.NET Core Middleware Componenten	44
12.	De ASP.NET Core Request Processing Pipeline	52
13.	wwwroot folder in ASP.NET Core	54
14.	Statische Files Middleware in ASP.NET Core	56
15.	Configuratie van Default Page in ASP.NET Core	59
16.	Developer Exception Pagina Middleware in ASP.NET Core	63
17.	De .NET Core Command Line Interface	67
18.	Referenties	73

1. Inleiding ASP.NET Core Framework

Nota : De nieuwste versie is .NET 5 (nov 2020)

ASP.NET Core 3.1:

ASP.NET Core is een gratis, open-source web development framework (van Microsoft), geoptimaliseerd voor cloud, dat cross-platform is (runt op Windows, Linux, en macOS).

Wat is .NET Core?

- > Development Framework
- > Open-Source
- > Cross-Platform
- > Modulair
- > Cloud - geoptimaliseerd

Het .NET Core framework kan gebruikt worden om verschillende soorten apps te ontwikkelen, bv applicaties voor **console, desktop, web, mobile**, cloud, IoT, machine learning, Microservices, games, ...

.NET Core is een modulair, lichtgewicht, performant, cross-platform framework. Het bevat alle basis-functionaliteit nodig om een basis .NET Core app te schrijven. Extra functionaliteit is beschikbaar in de vorm van NuGet Packages, die je kan toevoegen aan je app wanneer je ze nodig hebt. Op deze manier wordt een .NET Core application zo performant mogelijk gehouden, gebruikt het zo weinig mogelijk geheugen en wordt eenvoudig te onderhouden.

Waarom .NET Core?

Er zijn enkele beperkingen aan het vorige .NET Framework. Bijvoorbeeld runnen .NET Framework apps enkel op het Windows OS Platform.

Je hebt ook verschillende .NET APIs nodig voor verschillende windows-devices, bv voor Windows Desktop, Windows Store, Windows Phone en Web Applicaties. Het .NET Framework is platform-afhankelijk.

Tegenwoordig hebben we nood aan apps die op verschillende platformen kunnen runnen.

Bv een web app op een webserver (bv Linux), een front-end app op windows desktop, mobile app op verschillende devices,... Er was nood aan één enkel framework dat deze apps kon runnen. Microsoft heeft hiervoor .NET Core ontworpen, met de volgende kenmerken: een open-source, cross-platform dat kan werken op verschillende devices.

.NET Core open source:

Open-source Framework:

.NET Core Runtime: <https://github.com/dotnet/runtime>

.NET Core SDK: <https://github.com/dotnet/sdk>

ASP.NET Core: <https://github.com/dotnet/aspnetcore>

Language Compiler Platform Roslyn: <https://github.com/dotnet/roslyn>

Cross-Platform: .NET Core runt op Windows, Linux en macOS besturingssystemen. Voor elk besturingssysteem is er een runtime die de code uitvoert en dezelfde output genereert.

Consistent over verschillende Architecturen: Uitgevoerde code vertoont hetzelfde gedrag voor x64, x86, and ARM.

Ondersteunt voor soorten applicaties: Console, Desktop, Web, Mobile, Cloud, IoT, ML, Microservices, Gaming, ...

Ondersteunt meerdere programmeertalen: C#, F#, Visual Basic. Ook verschillende IDE's, bv Visual Studio 2017/2019, Visual Studio Code, Sublime Text, Vim,

Modulaire Architectuur: .NET Core is modulaire dmv NuGet Packages. Zelfs .NET Core library is provided een NuGet Package.

CLI Tools: .NET Core voorziet CLI tools (Command Line Interface) voor development en continuous-integratie.

Flexibele Deployment: deployment user-wide of system-wide of met Docker Containers (Microservices).

Compatibiliteit: Compatibel met .NET Framework en Mono APIs dmv .NET Standard Specificatie.

Waarvoor kunnen we .NET Core apps ontwikkelen?

Voor verschillende types van besturingssystemen en machines, bv Windows, Mac en Linux

Welke types van apps kunnen we ontwikkelen met .Net Core?

1. Web: ASP.NET Core MVC, Web API, Razor Pages, Microservices,
2. Mobile
3. Console
4. Desktop Applicaties (vanaf versie 3.0)
5. IoT (Internet of Things)
6. ML (Machine learning)
7. Gaming Apps
8. Cloud Apps

ASP.NET Core 3.1:

Applicatie Types:

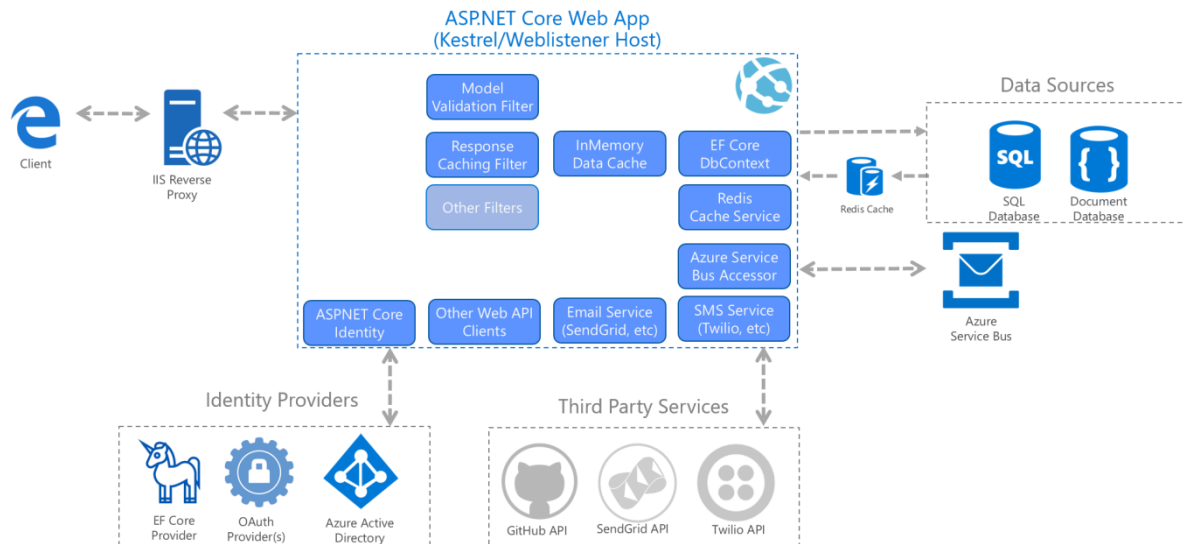
1. NET Core met Razor Pages
2. **NET Core met MVC (ASP.NET Core MVC)**
3. **NET Core with Web API (ASP.NET Core Web API)**
4. NET Core with Angular (Single Page Application) - SPA
5. NET Core with React JS (Single Page Application) - SPA
6. NET Core with React JS & Redux (Single Page Application) - SPA
7. ...

Wat is ASP .NET Core?

Volgens [MSDN](#), is ASP.NET Core een vrij recent cross-platform met hoge performantie, gebruikt weinig geheugen en resources, open-source Framework dat wordt gebruikt voor het ontwikkelen

van moderne, Internet-, cloud-gebaseerde Web Applicaties, IoT en Web APIs die ontwikkeld en gerund kunnen worden op Windows, Linux, of Mac besturingssystemen.

ASP.NET Core Architecture



Bron: <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>

2. .NET Core en ASP.NET Core Installatie

ASP.NET Core kan op 2 manieren worden geïnstalleerd:

- Door installatie van Visual Studio 2019**
- Door installatie van .NET Core SDK**

De .NET Core SDK installer bevat reeds de ASP.NET Core libraries.

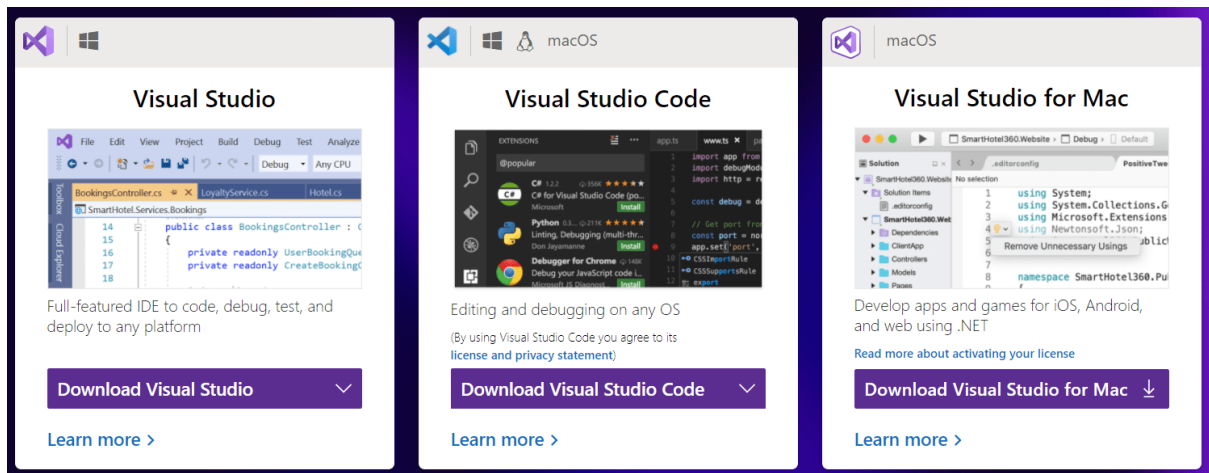
a. Installatie van Visual Studio 2017/2019:

.NET Core 2.1 en .NET Core 3.1 hebben een Long Term Support (LTS). Visual Studio 2019 ondersteunt beide versies.

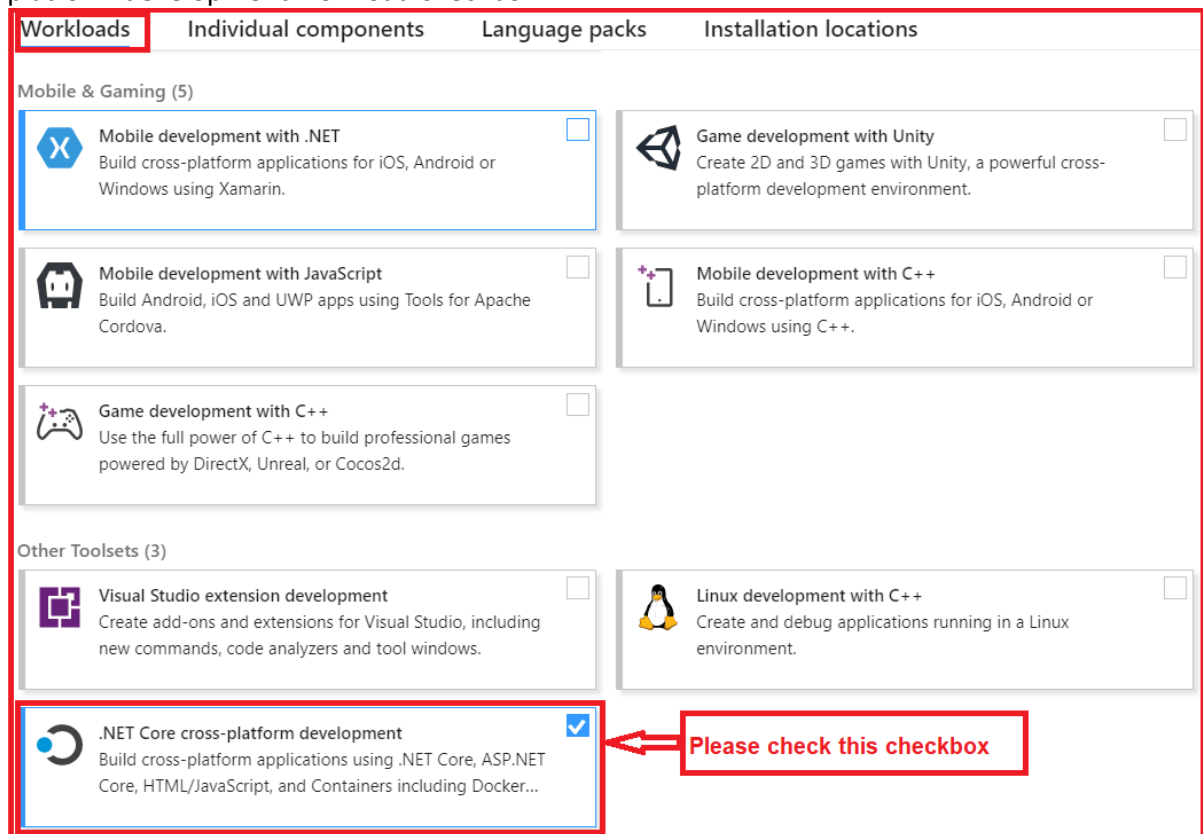
Visual Studio is een Integrated Development Environment (IDE) voor het ontwikkelen van o.a. .Core en ASP.NET Core applicaties.

Download en installeer Visual Studio

Download and install Visual Studio 2019 voor je besturingssysteem. Selecteer een edition. De Visual Studio community edition is gratis voor studenten, open-source contributors, en niet-professionelen. Momenteel is de meest recente versie **Visual Studio 2019** en je kan deze hier downloaden: <https://visualstudio.microsoft.com/>



Om .NET Core applicaties in Visual Studio 2019 te kunnen ontwikkelen heb je de .NET Core cross-platform development component nodig. Selecteer tijdens het installatie de .NET Core cross-platform development workload checkbox :



Via de command prompt (Windows OS) of Terminal (Linux OS) kan je de na de installatie de versie opvragen via de **“dotnet –version”** commando:

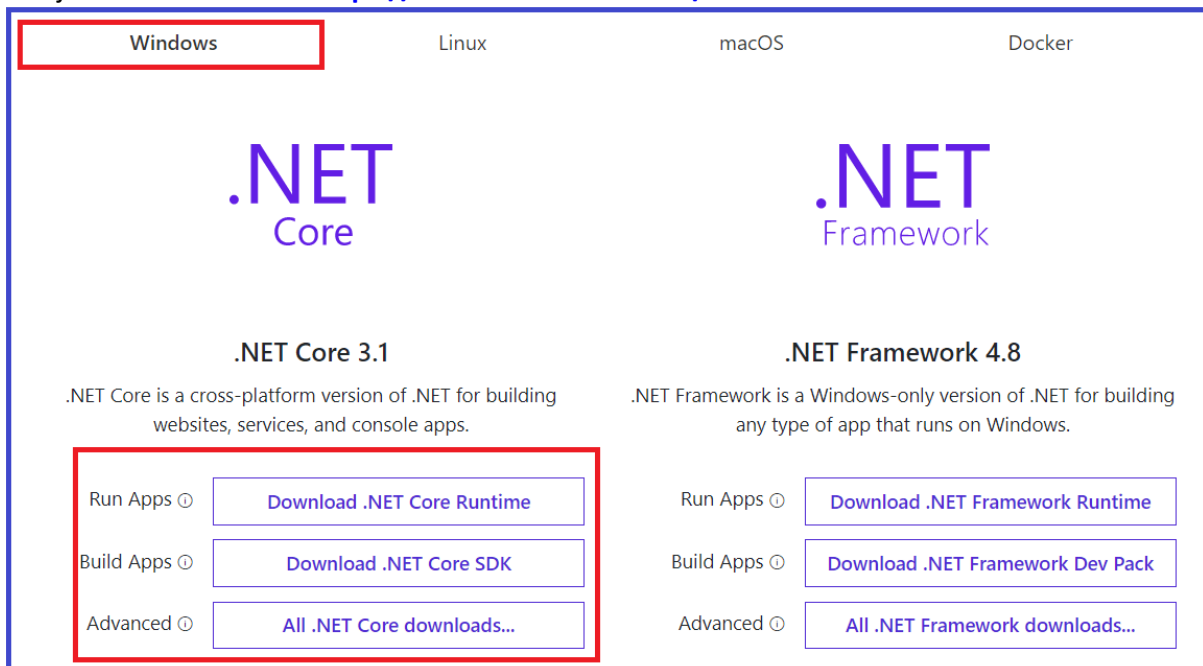
```
Command Prompt
Microsoft Windows [Version 10.0.19041.329]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\HP>dotnet --version
3.1.401
```

Indien het .NET Core Framework goed geïnstalleerd is op je machine, zie je het versie nummer verschijnen.

b. Download and install .NET Core SDK

Indien .NET Core 3.x niet automatisch geïnstalleerd is in je VS 2019, dan kan je deze ook apart bijinstalleren. ASP.NET Core is een onderdeel van .NET Core SDK (Software Development Kit), dus wanneer je de SDK voor .NET Core i.e. 3.1 installeert, kan je deze gebruiken in je VS 2019. De SDK kan je hier downloaden: <https://dotnet.microsoft.com/download>



Hier gebruiken we **.NET Core 3.1** . De laatste versie is momenteel .NET 5

Er zijn 3 mogelijke opties die je kan downloaden en installeren:

.NET Core Runtime: De .NET Core Runtime is enkel vereist om .NET Core applicaties te runnen. De .NET Core Runtime bevat enkel de libraries om .NET Core applicaties uit te voeren.

.NET Core SDK: Indien je zowel .NET Core applicaties wil kunnen ontwikkelen als runnen, download dan de .NET Core SDK. De .NET Core SDK bevat eveneens de .NET Core Runtime.

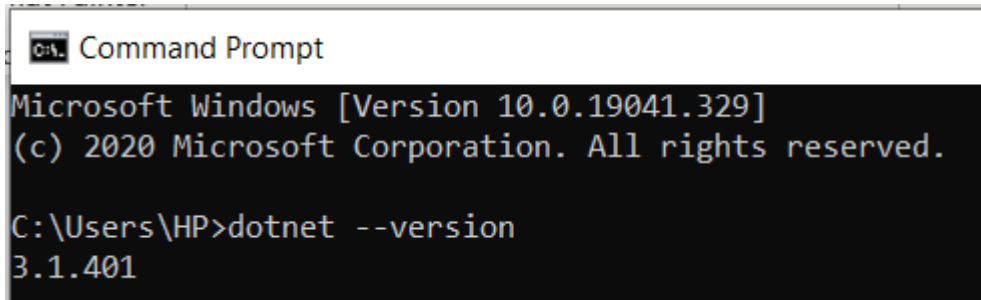
Alle .NET Core downloads: Indien je een oudere versie van .NET Core nodig hebt, kan je deze ook nog installeren:

Version	Status	Latest release	Latest release date	End of support
.NET 5.0	Preview	5.0.0-preview.7	2020-07-21	
.NET Core 3.1 (recommended)	LTS	3.1.7	2020-08-11	2022-12-03
.NET Core 3.0	End of life	3.0.3	2020-02-18	2020-03-03
.NET Core 2.2	End of life	2.2.8	2019-11-19	2019-12-23
.NET Core 2.1	LTS	2.1.21	2020-08-11	2021-08-21
.NET Core 2.0	End of life	2.0.9	2018-07-10	2018-10-01
.NET Core 1.1	End of life	1.1.13	2019-05-14	2019-06-27
.NET Core 1.0	End of life	1.0.16	2019-05-14	2019-06-27

Klik op **Download .NET Core SDK** button om .NET Core 3.1 SDK te downloaden

Verifieer de installatie

Verifieer na installatie .NET Core SDK via een Command Prompt. Typ het **dotnet --version** commando en dan enter.



```

C:\Users\HP>dotnet --version
3.1.401

```

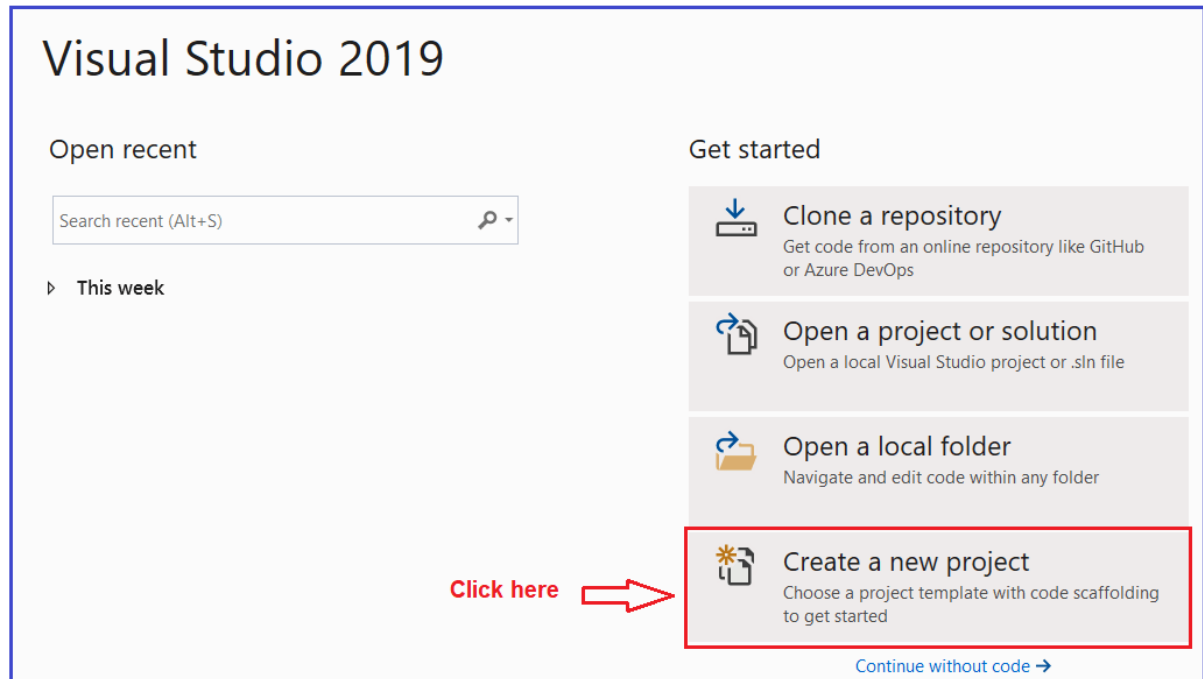
Indien het .NET Core Framework goed geïnstalleerd is op je machine, zie je het versie nummer verschijnen.

3. Aanmaken van een ASP.NET Core Web Applicatie

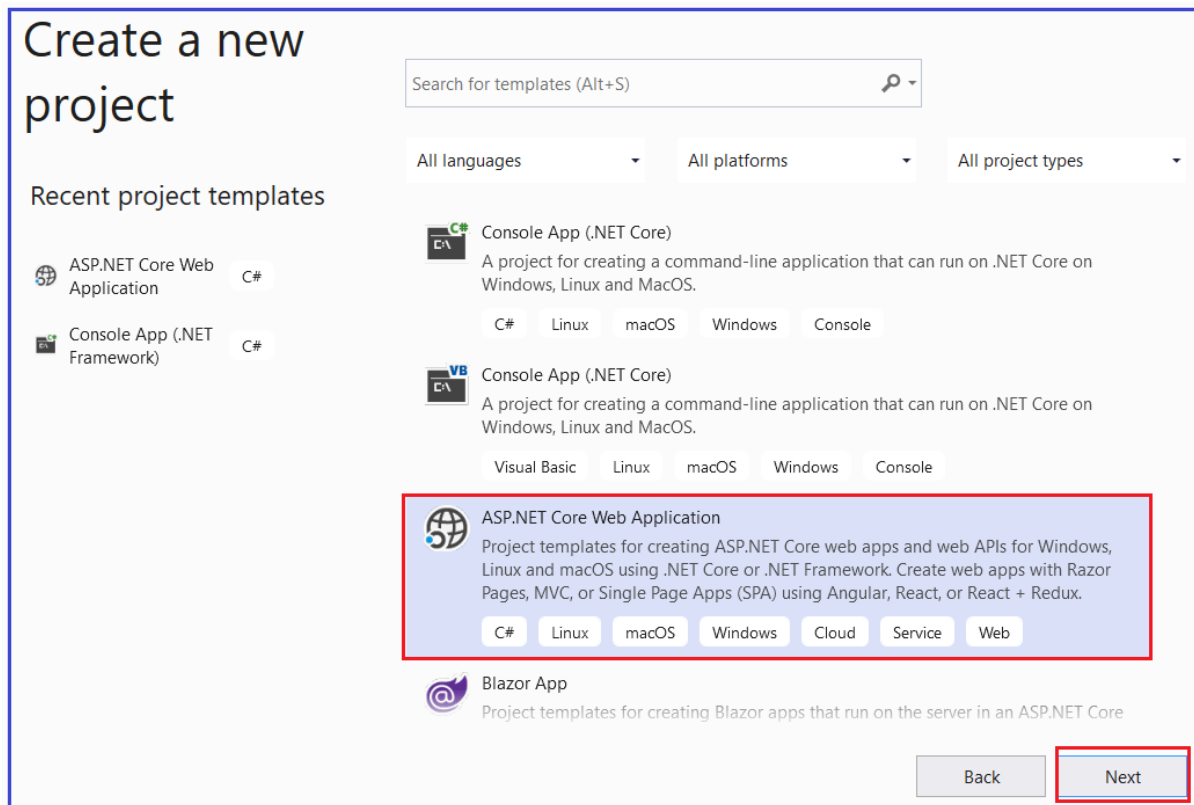
1. **Creëren van een ASP.NET Core Web Application met Visual Studio 2019.**
2. **Verschillende project templates van ASP .NET Core.**

Creëren van een ASP.NET Core Web Application in Visual Studio 2019

Open Visual Studio 2019, klik op “Create a new project “:



Het venster “Create a new project” wordt geopend. Selecteer **ASP.NET Core Web Application** template en click op de “Next” button:.



Create a new project

Search for templates (Alt+S)

All languages All platforms All project types

Recent project templates

- ASP.NET Core Web Application C#
- Console App (.NET Framework) C#

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C# Linux macOS Windows Console

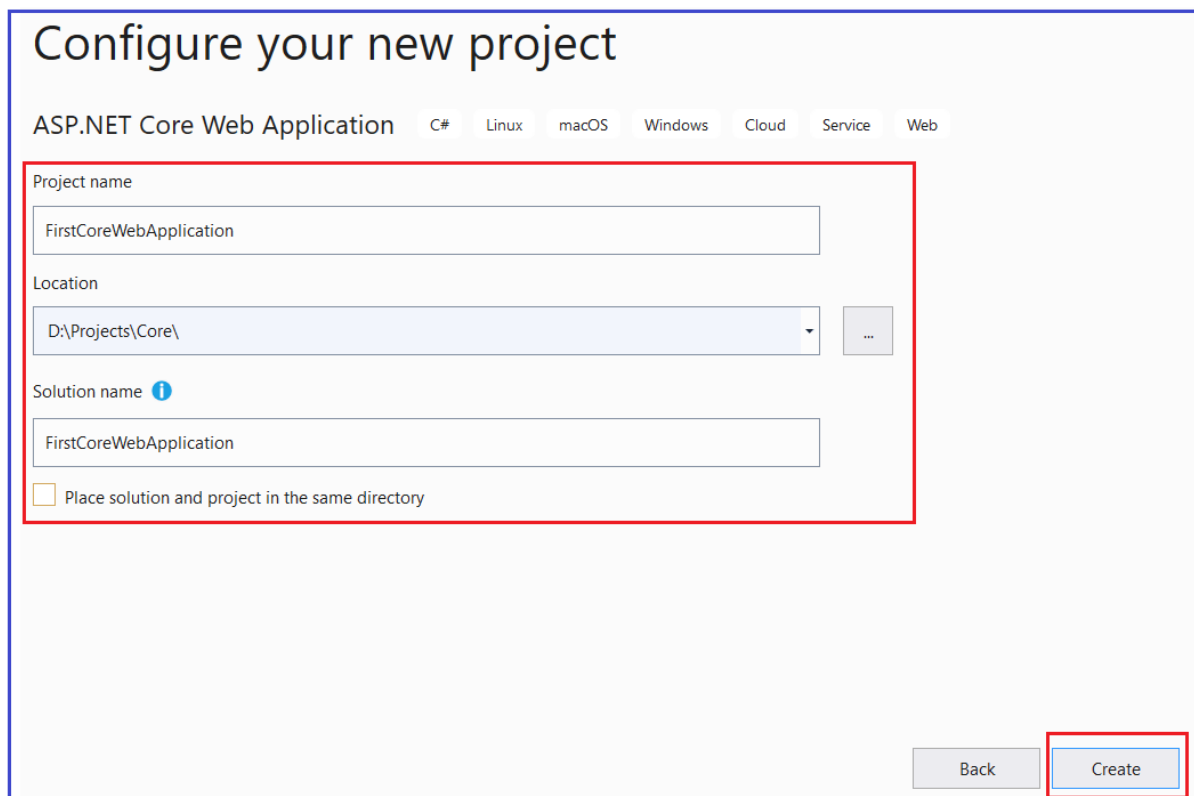
Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Visual Basic Linux macOS Windows Console

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
C# Linux macOS Windows Cloud Service Web

Blazor App
Project templates for creating Blazor apps that run on the server in an ASP.NET Core

Back Next

Het venster **Configure Your New Project** wordt geopend. Geef de naam van het project, locatie en naam van solution van je ASP.NET Core Web applicatie.
Bv “**FirstCoreWebApplication**” en klik op de Create button:



Configure your new project

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

Project name
FirstCoreWebApplication

Location
D:\Projects\Core\

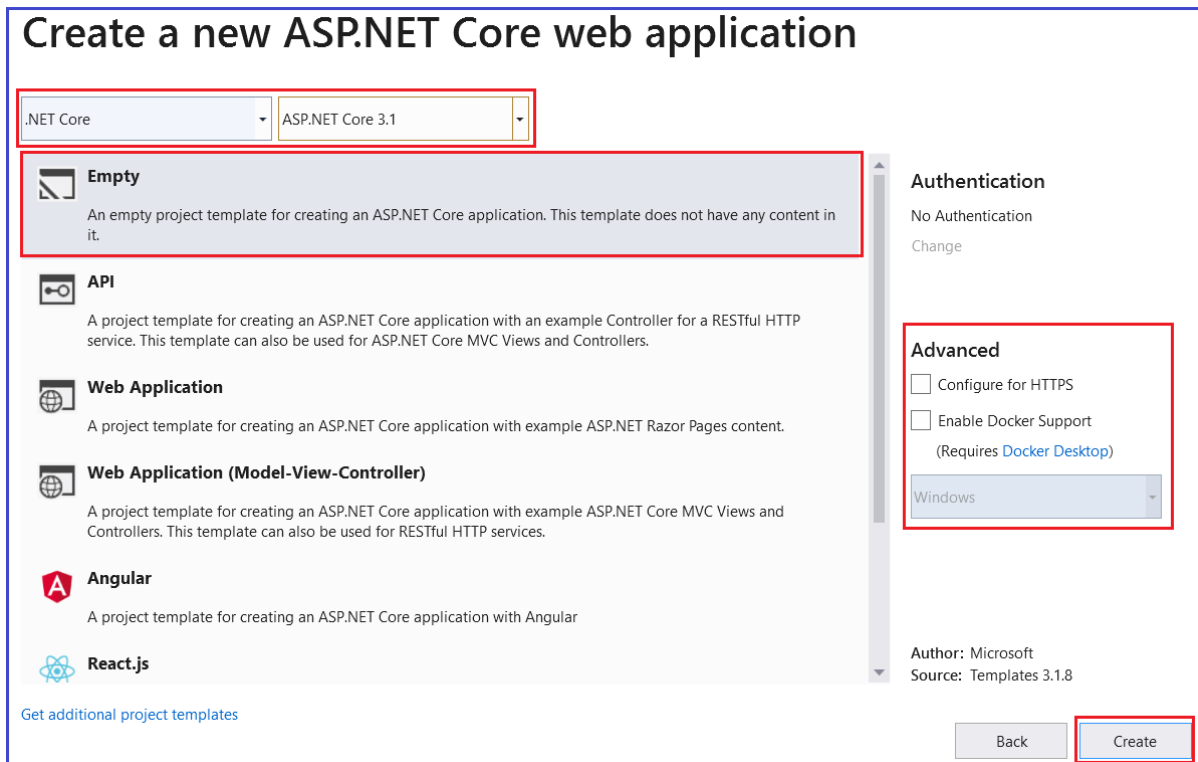
Solution name ⓘ
FirstCoreWebApplication

☐ Place solution and project in the same directory

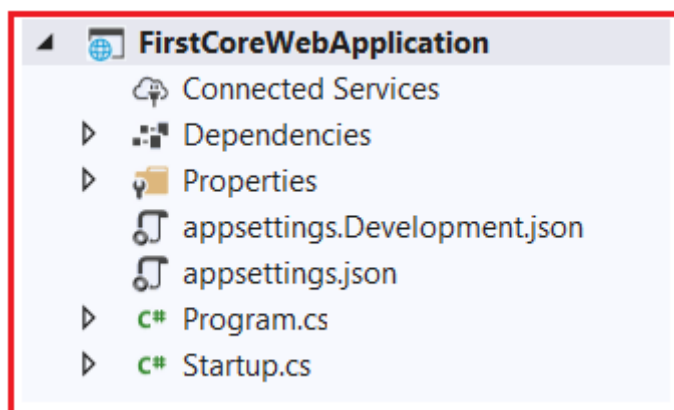
Back Create

Het venster **Create a new ASP.NET Core Web Application** wordt getoond:
Selecteer één van de ASP.NET Core Web application templates
(Empty, API, Web Application, Web Application (MVC), Angular, ...).

We gebruiken voor de demo de **Empty** template
Selecteer de juiste versie van **.NET Core (3.1)**.
Uncheck de checkbox Enable Docker support:

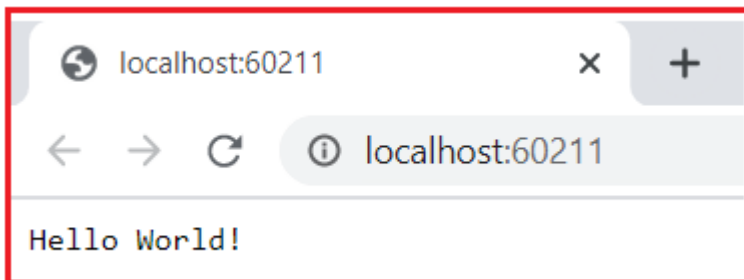


Klik op de Create button
Het ASP.NET web project wordt nu aangemaakt
Het project bevat de volgende folders en files in Visual Studio 2019:



Run de ASP.NET Core Web Applicatie:

Klik op IIS Express of druk op F5 (met Debug) of Ctrl + F5 (zonder Debug). Een browser wordt geopend en toont het volgende:

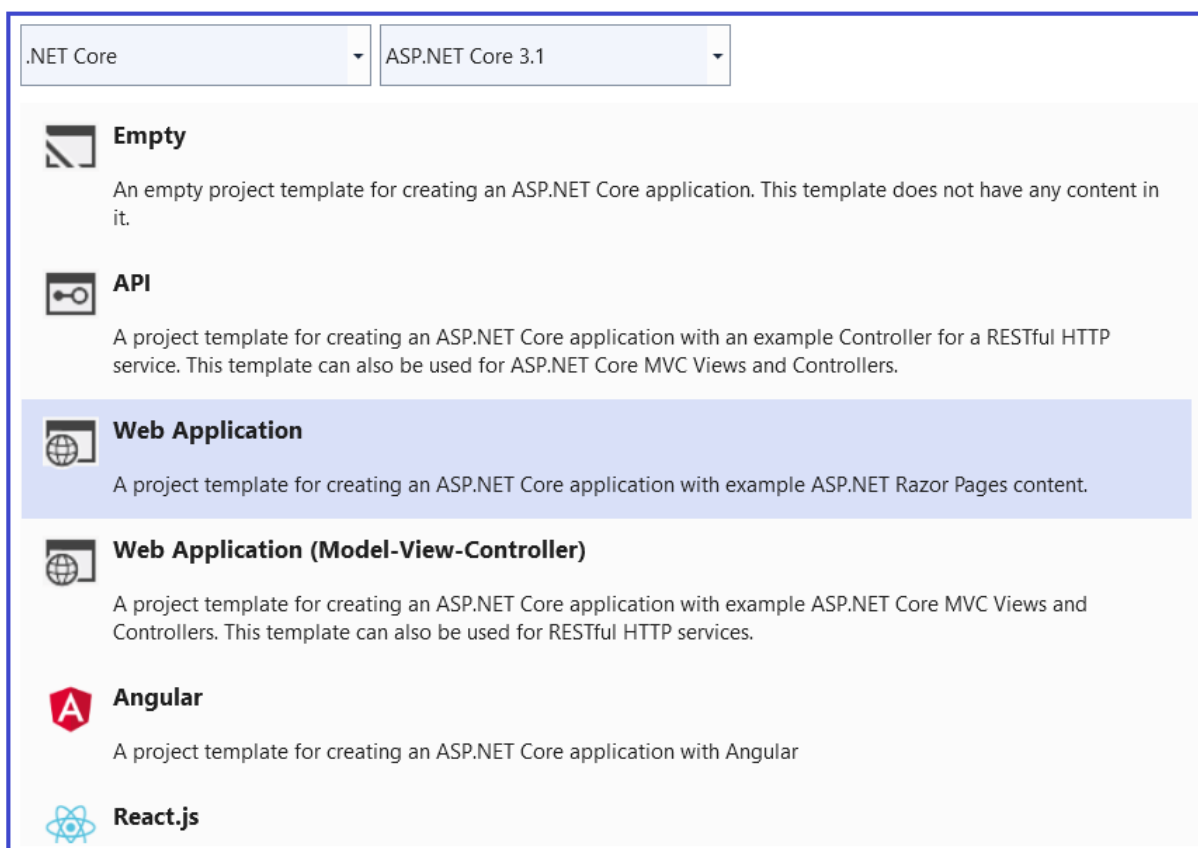


De output “**Hello World!**” komt van de **Configure** methode van de **Startup** class in het bestand **Startup.cs**.

Open **Startup.cs** en wijzig de “**Hello World!**” string en herstart de app.

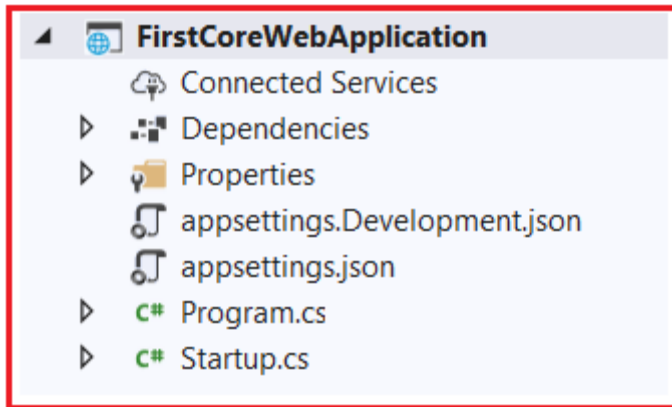
Project templates in ASP.NET Core Application

De andere project templates gaan we nu eens kort overlopen:



Empty Project Template:

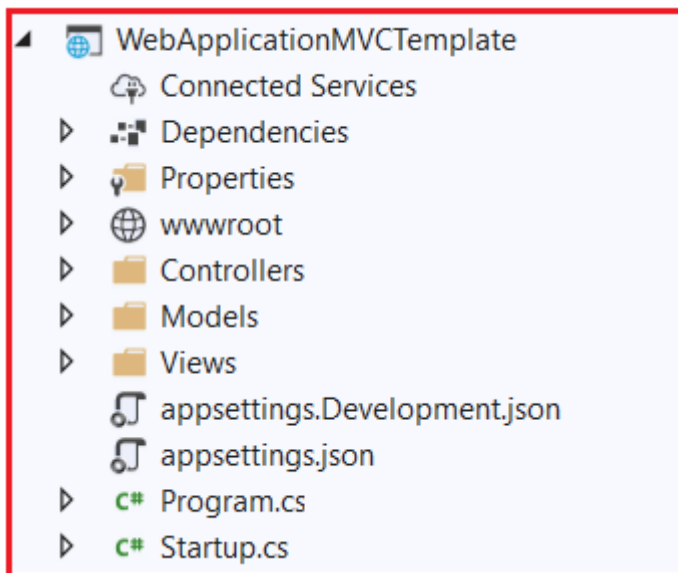
Bevat eigenlijk bijna niets van basisfunctionaliteit. Deze moet je zelf toevoegen wanneer je ze nodig hebt.



Web Application (Model-View-Controller) Template:

De Web Application (Model-View-Controller) template bevat alles dat nodig is om ASP.NET Core MVC Web Applicatie te maken. De Web Application (Model-View-Controller) template bevat reeds Models, Views, en Controllers folders. Het bevat evenens web-specifieke zaken, zoals JavaScript, CSS files, Layout files, ... die nodig of vereist zijn om een web applicatie te maken.

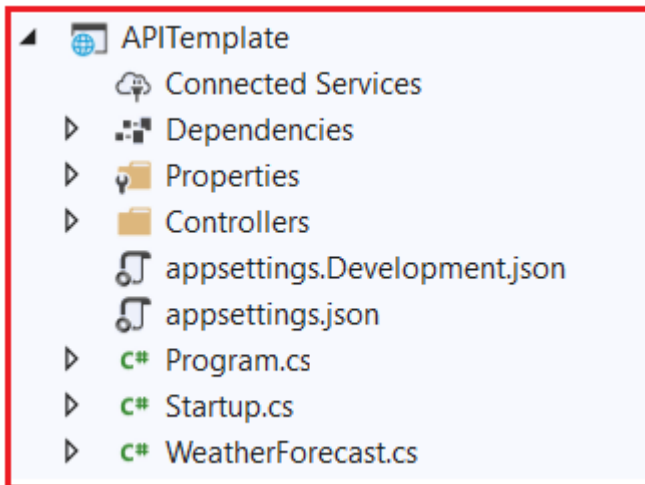
Een ASP.NET MVC Web Applicatie bevat de volgende folder-structuur :



API Template:

De API template bevat alles dat vereist en noodzakelijk is om een ASP.NET Core RESTful HTTP web service aan te maken.

.Deze heeft de volgende folder-structuur :



Deze bevat enkel de Controllers folder. The website-specifieke bestanden, zoals CSS files, JavaScript files, view files, layout files, .. zijn hier niet aangemaakt.

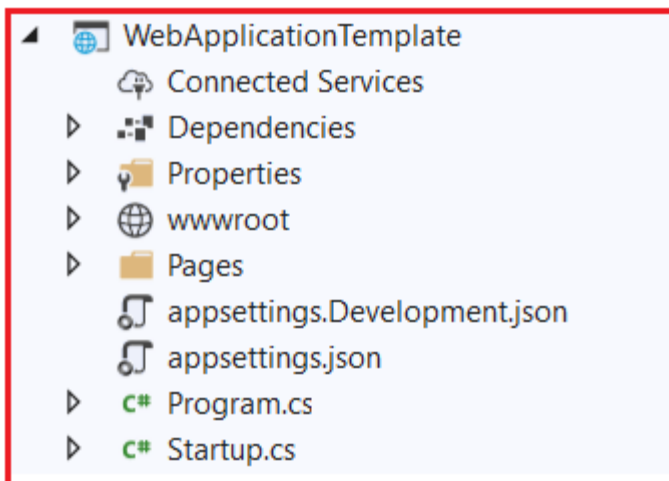
Dit is omdat een Web API geen userinterface heeft.

Deze API template heeft ook geen Views folder, aangezien views niet nodig zijn voor een API.

Web Applicatie Template

De Web Application Template gebruikt de **Razor Pages framework** om web pagina's te bouwen. Deze applicatie heeft niet de complexiteit nodig van een ASP.NET MVC web applicatie.

De folder-structuur is als volgt:



Angular, React.js, React.js, and Redux:

Je kan eveneens asp.net core web applicaties in combinatie met Angular, React of React en Redux maken. Deze zijn SPA (Single page applications). We gaan in deze module er niet dieper op in.

4. ASP.NET Core Project File

Wanneer je een project aanmaakt in VS 2019, wordt er een project file aangemaakt. Deze heeft de extensie “.csproj

Er zijn wel een aantal verschillen tussen een console, wpf project file en een web project file

De Project file in ASP.NET Core Application:

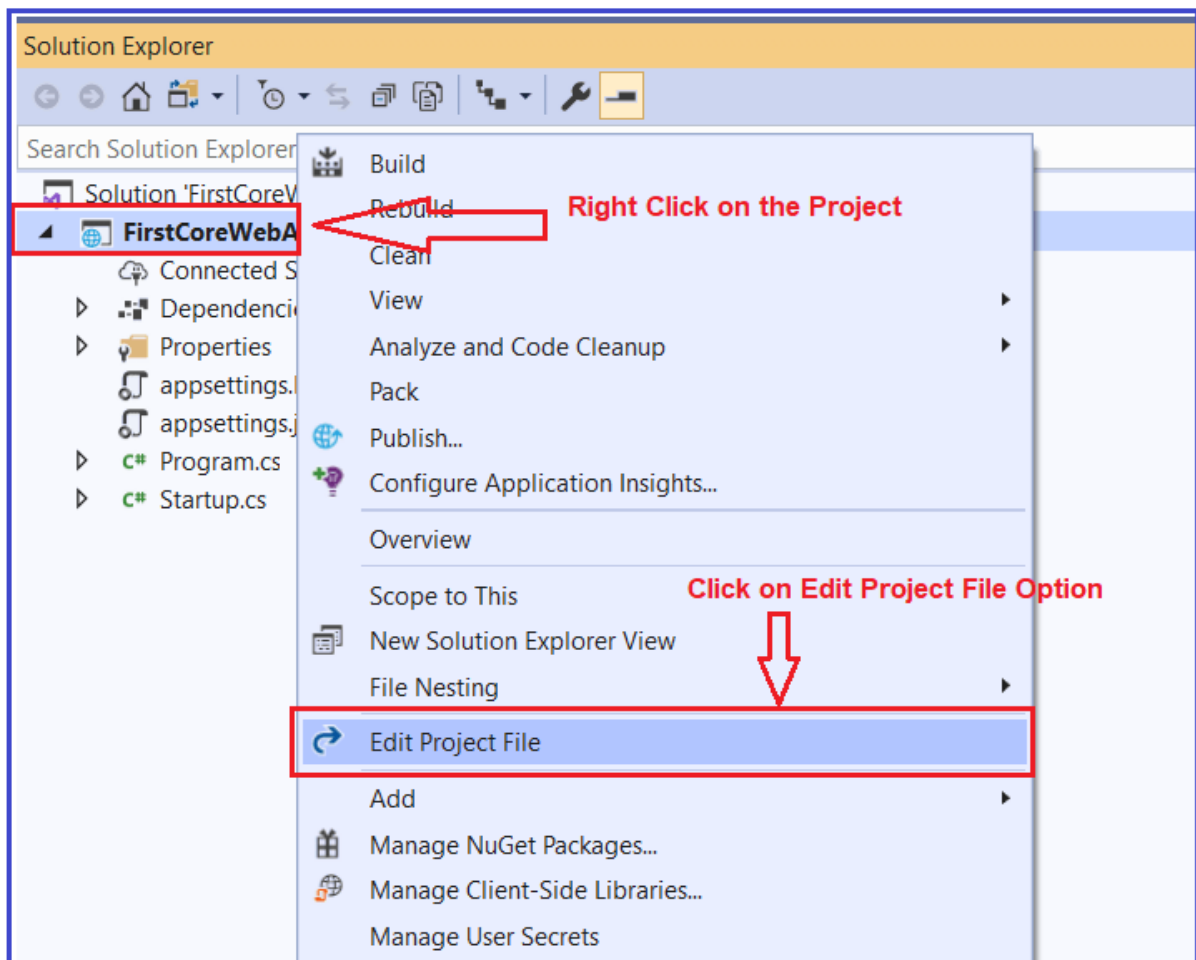
De ASP.NET Core Project File bevat geen referenties naar files of folders.

In ASP.NET Core, bepaalt het File System welke folders en files tot het project behoren. Alle files en folders in de project root folder zijn behoren in principe tot het project. Deze worden dan ook getoond in de Solution Explorer.

Indien je een bestand/folder onder de project folder toevoegt via de File Explorer, dan is dat bestand/folder onderdeel van je project.

Hoe de ASP.NET Core Project File aanpassen:

Om de ASP.NET Core project file aan te passen, rechtsklik met de muis op de project name in de Solution Explorer en selecteer “Edit Project File” van het context menu :



In de code-editor wordt de .csproj getoond in VS 2019:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

Inhoud van de ASP.NET Core Project File:

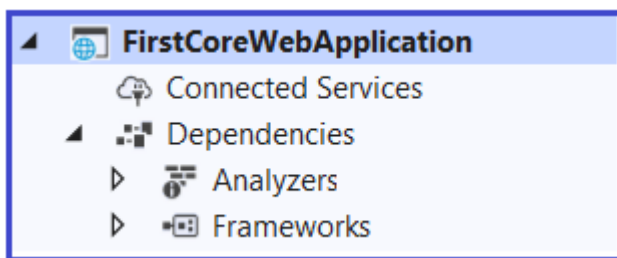
De volgende ASP.NET Core Project File elementen kan je terugvinden :

TargetFramework:

De TargetFramework element in de project file duidt het target framework aan van je applicatie. De Target Framework Moniker (TFM) wordt gebruikt, bv framework netcoreapp3.1. De netcoreapp3.1 is de Moniker voor .NET Core 3.1.

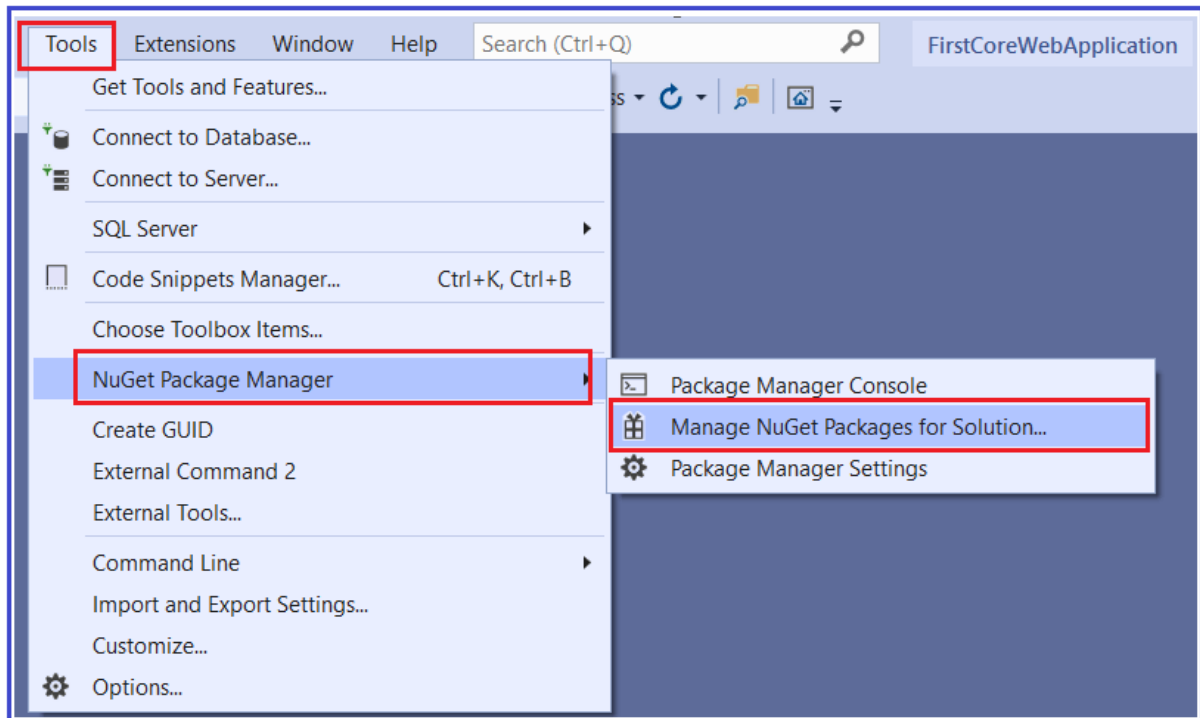
NuGet Packages toevoegen aan ASP.NET Core app:

In de Dependencies sectie in de solution explorer kan je de geïnstalleerde packages zien. Wanneer we packages installeren, worden deze (en hun dependency packages) hier ook getoond:



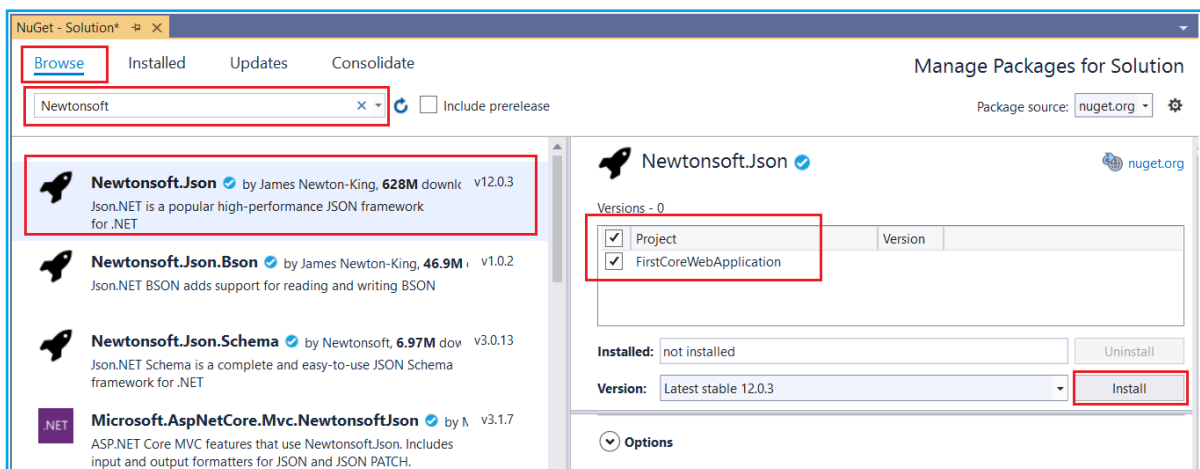
Installeer via de NuGet Package Manager een NuGet Package.

Ga naar tools => NuGet Package Manager => Manage NuGet Packages for Solution:

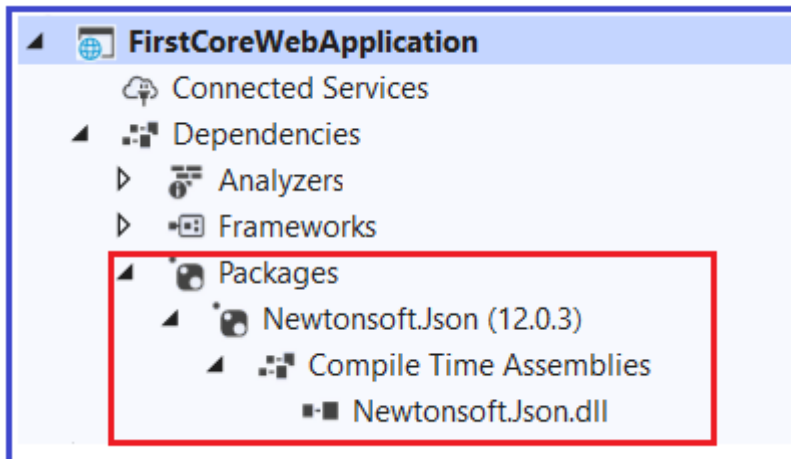


Voeg de Newtonsoft.Json Nugetpackage toe:

Selecteer de Browse tab , zoek naar Newtonsoft en installeer de laatste stabiele versie van **Newtonsoft.Json** :

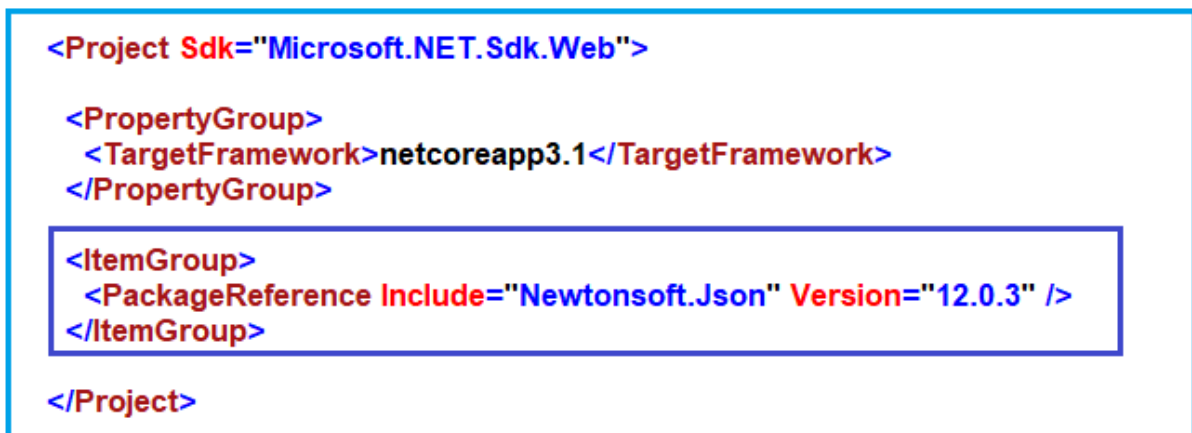


Eens de package geïnstalleerd is, kan je deze onder Dependencies/Packages zien staan met versienummer in de Solution Explorer :



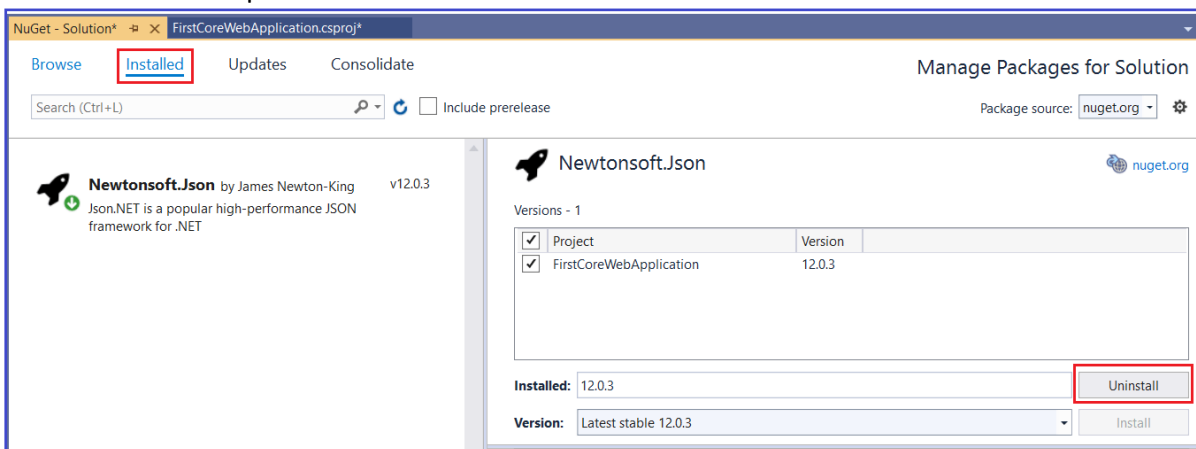
Ook alles waarnaar deze package refereert, wordt hier getoond.

In het project file (.csproj), zie je ook de PackageReference naar Newtonsoft.Json staan:



Verwijderen van Package in ASP.NET Core:

Via de Nuget Package Manager Kan je via de installed tab terug packages verwijderen door ze te selecteren en dan op 'Uninstall' te klikken:



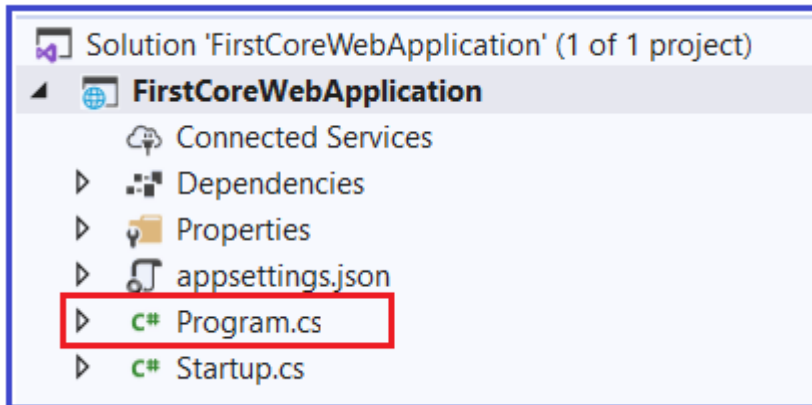
De dependencies in de .csproj worden dan ook verwijderd en in de solution explorer worden ze dan niet meer getoond.

5. ASP.NET Core Main Methode

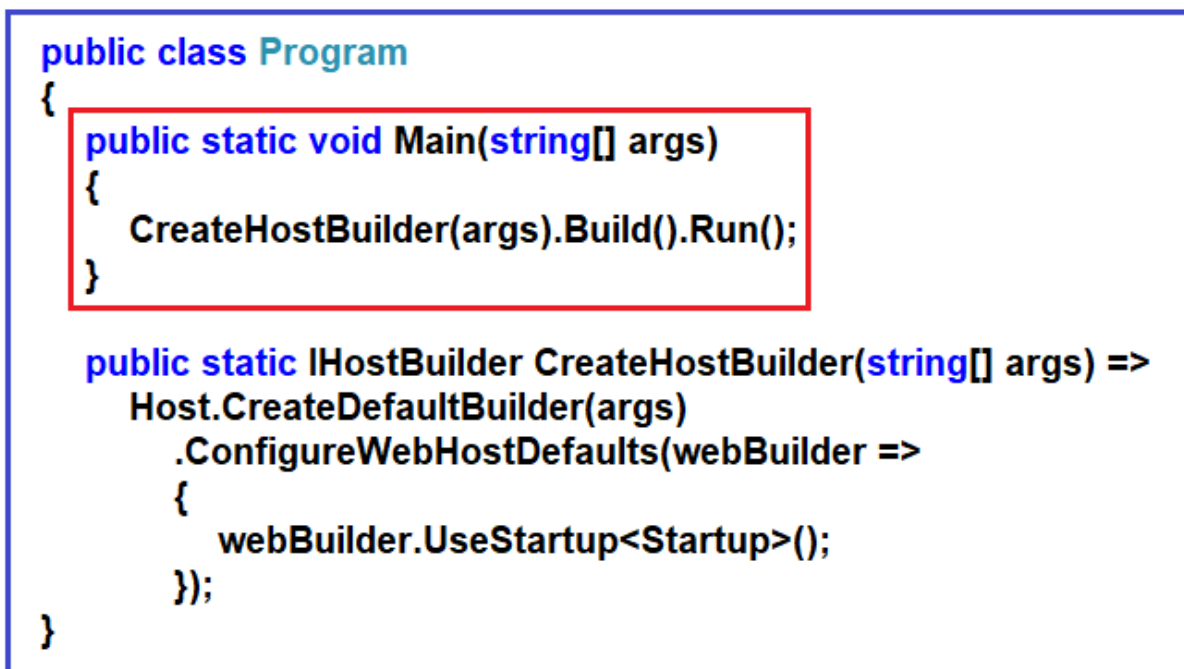
1. Rol van de ASP.NET Core Main Methode.
2. Waarom Main() method in ASP.NET Core?
3. Wat er achter de schermen gebeurt wanneer je een .NET core applicatie runt?

Creëer een Web applicatie van de Empty Template (zie hoofdstuk 2)

In de solution explorer zie je de volgende folder structuur :



Dubbel-klik op Program.cs. In het code-venster zie je de inhoud van de file:



De class Program bevat de methode **public static void Main()** . We weten van uit de module Leren Programmeren dat bij de Console app de Main methode de “entry-point” is bij uitvoering van de app.

Waarom een Main() methode in ASP.NET Core?

De ASP.NET Core Web Applicatie start eigenlijk op als een Console Applicatie, dus bij opstart van de ASP.NET Core Web application, wordt eveneens de Main() methode als eerste uitgevoerd. De Main() methode configureert ASP.NET Core en start het op, op dit moment wordt de applicatie . een ASP.NET Core web applicatie.

Als je kijkt wat er in de Main() methode staat, zie je dat het de CreateHostBuilder() methode aanroept met de command line arguments :

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

De **CreateHostBuilder()** methode geeft een object terug die de **IHostBuilder** interface implementeert. De **Host** is een static class die kan worden gebruikt om een instantie van HostBuilder aan te maken met pre-geconfigureerde defaults.

De CreateDefaultBuilder() methode maakt dus een object aan van de klasse HostBuilder. Intern wordt **Kestrel (Internal Web Server for ASP.NET Core)** geconfigureerd, en eveneens IISIntegration, and enkele andere configuraties.

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

De Main() methode, roept op dit IHostBuilder object de **Build()** methode aan, die een web host aanmaakt. De asp. core web applicatie wordt dan gehost door deze Web Host.

De **Run()** methode wordt dan aangeroepen, die uiteindelijk de **web applicatie opstart** en deze luistert naar binnenkomende HTTP requests.

De **CreateHostBuilder()** methode roept de static **CreateDefaultBuilder()** methode aan op de **Host** class. De CreateDefaultBuilder() methode creëert een web host met de default configuraties.

Voor het aanmaken van een web host, doet de CreateDefaultBuilder() methode verschillende zaken. In het volgende hoofdstuk gaan we hier dieper op in.

Startup Class

De Startup class is gedefinieerd in de Startup.cs file. Open deze file in VS 2019. De Startup class is wordt geconfigureerd via de UseStartup() extension methode van de IHostBuilderclass. De Startup class heeft volgende 2 methoden:

ConfigureServices

Configure

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello World!");
            });
        });
    }
}
```

De **ConfigureServices()** methode van de Startup class configureert de services die vereist zijn door de applicatie.

De **Configure()** methode van de Startup class zet een pijplijn op van de request-verwerking (request processing) van de applicatie.

6. ASP.NET Core InProcess Hosting

1. Welke taken verricht de methode `CreateDefaultBuilder()`?
2. Wat is InProcess hosting in ASP.NET Core?
3. Hoe configureer je InProcess hosting in ASP.NET Core?
4. Hosten van ASP.NET Core Application InProcess hosting Model
5. Wat gebeurt er als we Hosting model als InProcess zetten?
6. Hoe werkt InProcess hosting in ASP.NET Core?
7. Wat is IIS Express?

We werken terug met een asp.net core web app aangemaakt met de Empty project template:

```
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>();
            });
}
```

Wanneer we een ASP.NET core applicatie uitvoeren, dan zoekt de **.NET Core runtime** naar de **Main() methode**. De `Main()` methode is de entry point van de .net core applicatie wanneer ze wordt opgestart.

De `Main()` methode van de class `Program` roept de static `CreateHostBuilder()` methode aan. Dan roept de `CreateHostBuilder()` methode de static `CreateDefaultBuilder()` methode op de `Host` class. De `CreateDefaultBuilder()` methode configureert de web host die onze web app zal hosten met default configuraties.

Welke taken verricht de `CreateDefaultBuilder()` methode?

De `CreateDefaultBuilder()` methode verricht o.a. de volgende taken :

1. **Opzetten van de webserver**
2. **Laden van de host en applicatieconfiguratie van verschillende configuratiebronnen**
3. **Configuratie van logging**

Een ASP.NET core Web applicatie kan op 2 manieren worden gehost:

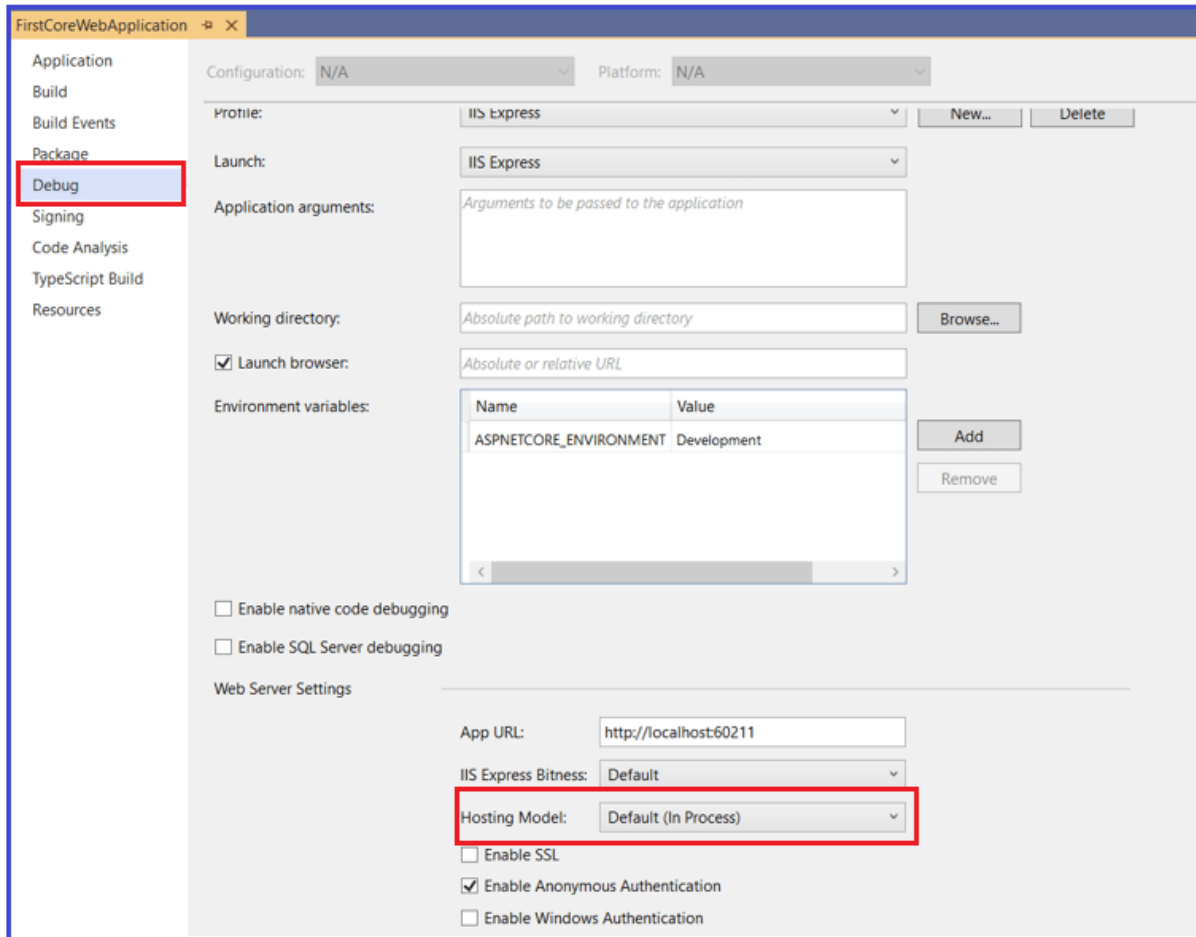
InProcess hosting
OutOfProcess hosting

Bij default wordt bij creatie van een web app vanaf een Template in VS

20199 InProcess hosting gezet. Dit laat **hosting** toe van de applicatie in IIS of IIS Express

Hoe kan je dit verifiëren?

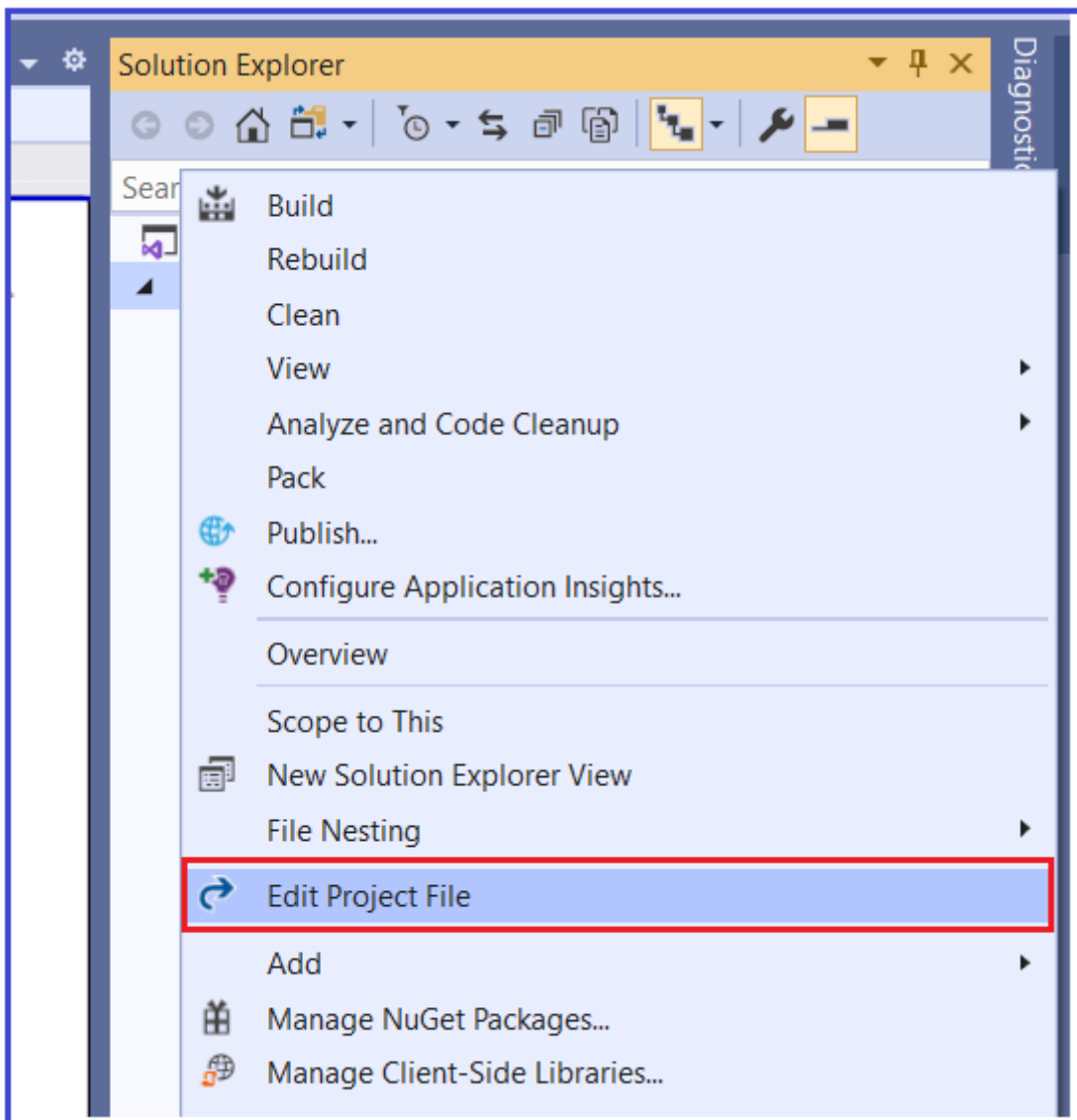
Via de Project properties van je web applicatie. Rechtsklik op je project in solution explorer en selecteer de properties option van het context menu. In het properties venster, Selecteer Debug en kijk wat er in de Hosting Model drop-down lijst staat:



Configureren van InProcess hosting in ASP.NET Core

Om InProcess hosting voor ASP.NET Core Web applicaties te configureren, is er één enkele setting: **<AspNetCoreHostingModel>**

Voeg dit element toe aan je applicatie project file (.csproj) met de waarde **InProcess**.



Once you open the Application Project file then modify it as shown below. As you can see, here we add **<AspNetCoreHostingModel>** element and set its value to **InProcess**. The order possible value for this element is **OutOfProcess**.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  </PropertyGroup>
</Project>
```

Wat gebeurt er als we InProcess hosting zetten?

De `CreateDefaultBuilder()` zegt dat de setting voor hosting als `InProcess` staat (`AspNetCoreHostingModel` element in de project file).

De `CreateDefaultBuilder()` methode roept intern de `UseIIS()` methode aan en host de web-applicatie binnen de IIS worker proces (dit is `w3wp.exe` voor IIS en `iisexpress.exe` voor IISExpress).

Het **InProcess** hosting model is **performanter** in het verwerken van de requests dan het `OutOfProcess` hosting model.

Wat is IIS Express?

IIS Express is een lightweight, self-contained versie van IIS. Het is geoptimaliseerd voor web applicatie development. Belangrijk is om te weten dat we IIS Express enkel in development gebruiken, nooit in productie. In productie gebruiken we meestal IIS.

OutOfProcess hosting

Voor **OutOfProcess** hosting, zijn er 2 web servers:

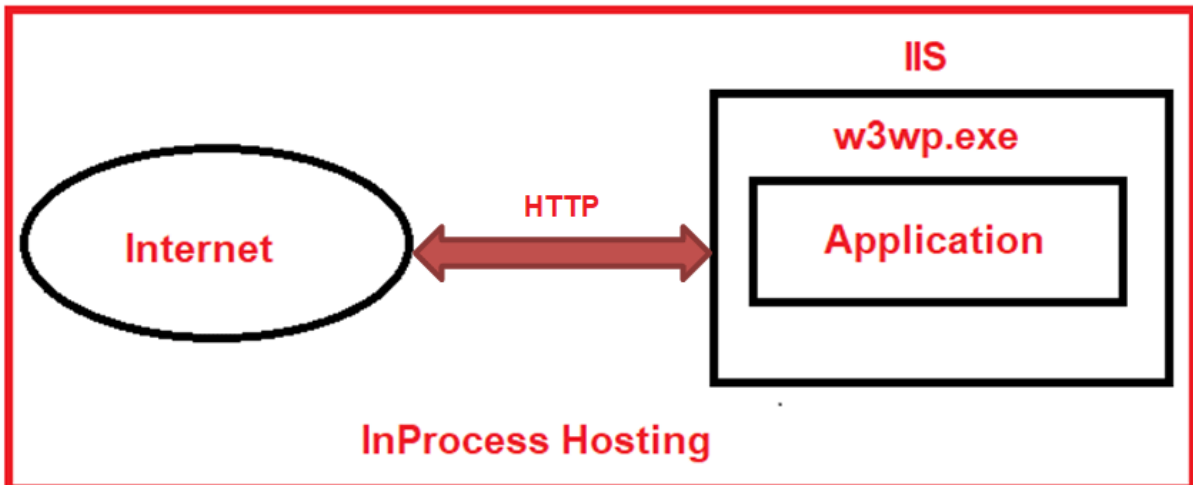
1. een interne web server
2. een externe web server.

De internal web server is wordt **Kestrel** genoemd en de externe web server kan

IIS, Nginx, of Apache zijn Met het `InProcess` hosting model, is er maar één web server nl IIS.

Bij de outofprocess hosting model moeten de requests worden doorgegeven tussen de interne en externe web servers.

Daarom is het `InProcess` hosting model performanter dan het `OutOfProcess` hosting model.



7. Kestrel Web Server in ASP.NET Core

1. [Wat is een Kestrel Web Server?](#)
2. [Hoe Kestrel Web Server configureren?](#)
3. [Runnen van .NET Core application met Kestrel Web Server](#)
4. [Runnen van .NET Core Application met .NET Core CLI.](#)

Wat is een Kestrel Web Server?

ASP.NET Core is een cross-platform framework.

Kestrel is de cross-platform web server voor de **ASP.NET Core applicatie**. Deze webserver ondersteunt dus alle platformen en versies die ASP.NET Core ondersteunt. Kestrel is de standaard interne web server in een ASP .NET Core application.

Standaard gebruikt VS 2019 **IIS Express** voor het hosten en runnen van een ASP.NET Core applicatie. De naam van het process dat wordt gerund is dus **IIS Express** (zie vorig hoofdstuk)

Runnen van de asp.net core web app met Kestrel Web Server

Open via de solution explorer de [launchSettings.json](#) file onder de Properties folder van je project:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:60211",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "FirstCoreWebApplication": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000"
    }
  }
}
```

IIS Server (i.e. IISExpress) will use the below URL to run your application
http://localhost:60211

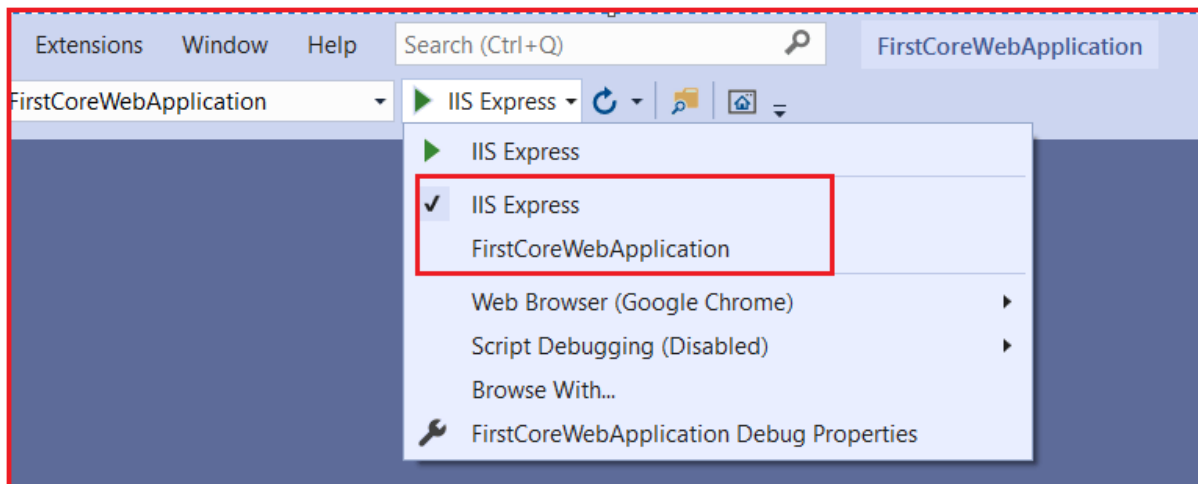
This Profile Settings for IISExpress

This profile is used by Kestrel Server and it will use the below URL
http://localhost:5000

In de Profiles sectie zijn er 2 sub-secties :

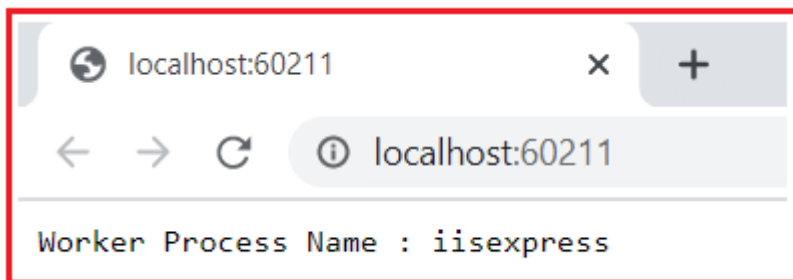
- IIS Express (IIS Server)
- Kestrel server

Wanneer je bij het runnen van je app kiest voor IIS Express, dan wordt deze opgestart met IIS Express, als je de naam van je project kiest in het opstartmenu: wordt je app opgestart met Kestrel web server:



Runnen van je web applicatie met IIS Express:

Indien je opstart met IIS Express, dan zal er gebruik gemaakt worden van de URL en poort-nummer die gespecificeerd staat in de sectie iisSettings van de launchSettings.json file:

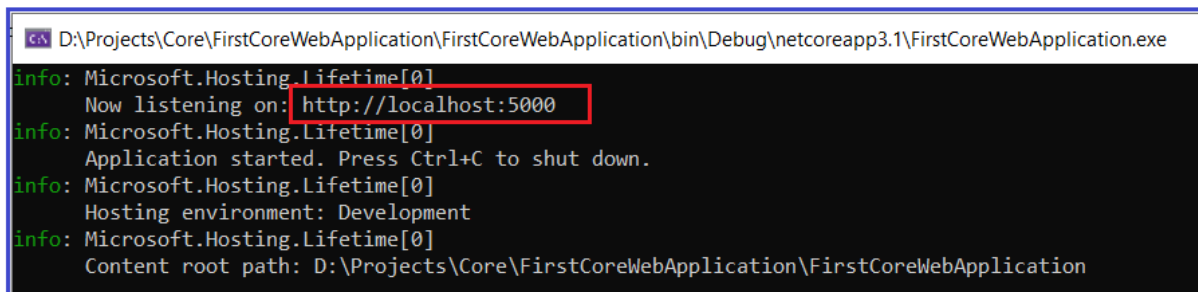


Runnen van je web applicatie met Kestrel Web Server:

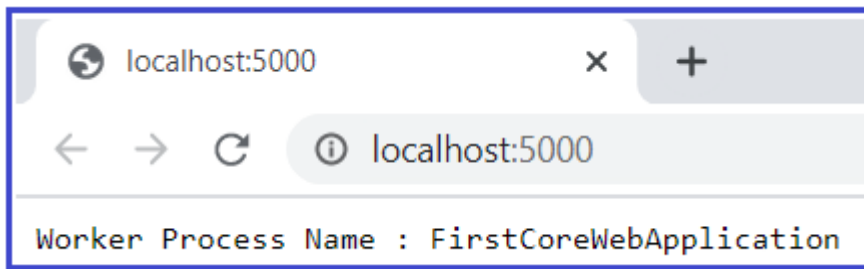
Selecteer bij het runnen de naam van je project uit het opstartmenu:



Als je nu opstart, zie je dat een command prompt wordt opgestart. De app wordt gehost door de Kestrel server. De URL en poort nummer worden nu uit de 2de sectie gehaald onder profile van de launchSettings.json file.



Er wordt ook een browser geopend en deze toont dezelfde url en poort-nummer:



Runnen van asp.net core web app via .NET Core CLI?

Een ASP.NET Core applicatie kan ook worden opgestart vanaf de command line dmv de .NET Core CLI. **CLI is de afkorting voor Command Line Interface.**

Wanneer we de app opstarten via de .NET Core CLI, gebruikt de .NET Core runtime Kestrel als webserver.

Runnen van dot net core application dmv .NET Core CLI Commando:

Open de command prompt en typ “**dotnet** —” en druk op enter :

```
Command Prompt
Microsoft Windows [Version 10.0.19041.329]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Users\HP>dotnet --
```

Nu zie je alle mogelijk commando's die je kan gebruiken :

```
Command Prompt

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.

Run 'dotnet [command] --help' for more information on a command.
```

Gebruik van de CLI

Je kan bv een nieuw project aanmaken via het **new** commando, een project kan je compileren via het **build** commando, je kan je project publiceren via het **publish** commando. En nog veel meer...

Runnen van .NET Core application dmv .NET Core CLI

In de Windows Command Prompt en ga naar de folder waar je .csproj van je project staat. Bv: **"D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication"**

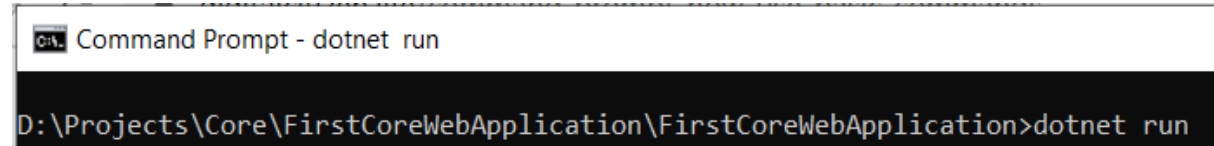
```
Command Prompt

C:\Users\HP>D:

D:\>cd D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication

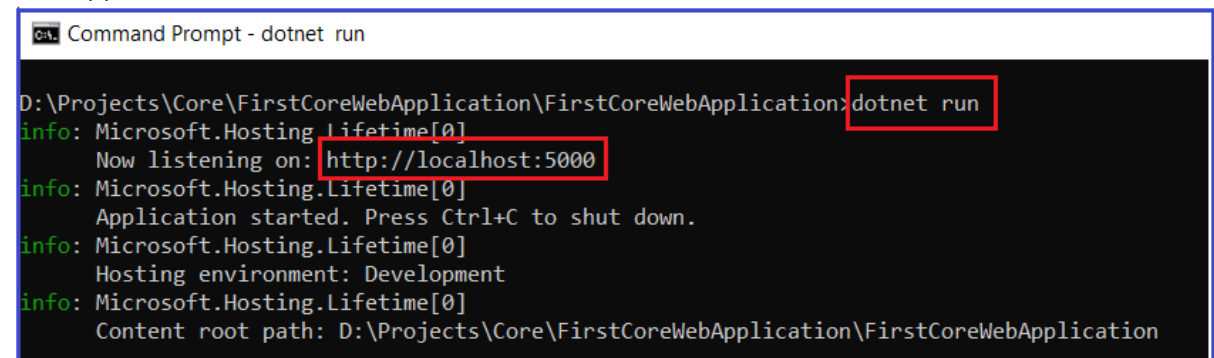
D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication>
```

Voer het commando “dotnet run” uit:



```
Command Prompt - dotnet run
D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication>dotnet run
```

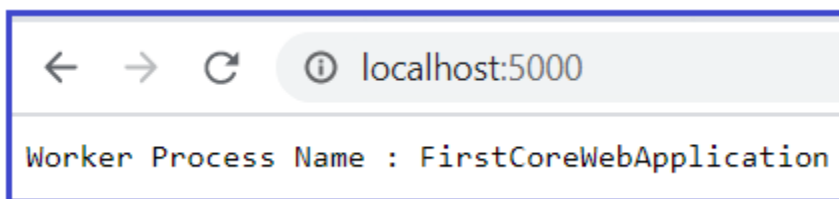
Nu zal de .NET Core CLI je app compileren en runnen. Er wordt eveneens de URL getoond van je web app:



```
Command Prompt - dotnet run
D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication>dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication
```

.Standaard luistert je app naar **http://localhost:5000**. Dit is de url die in launchSettings.json file voor Kestrel staat gedefinieerd

Open een browser en navigeer naar **http://localhost:5000** URL :



Aanpassen van het poortnummer:

Je kan eveneens het poortnummer voor de Kestrel Server aanpassen in launchsettings.json.

Bv: wijzig het poortnummer naar 60222.

```

{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:60211",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "FirstCoreWebApplication": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:60222"
    }
  }
}

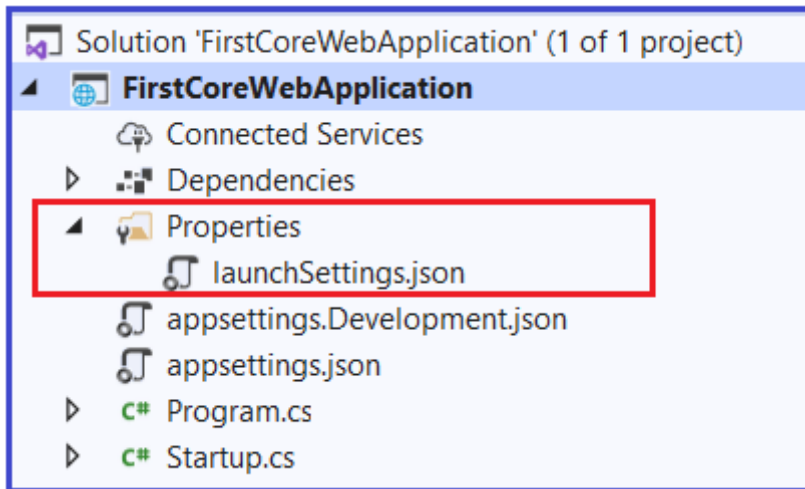
```

Bewaar je wijzigingen en run opnieuw je app met de Kestrel Server. Je zal nu zien dat de url gewijzigd is in de command prompt

8. ASP.NET Core launchSettings.json file

Maak een Empty ASP.NET Core Web Application met naam **FirstCoreWebApplication**

De launchSettings.json file kan je terugvinden in de solution explorer onder Properties :



LaunchSettings.json bestand in ASP.NET Core

De settings die in dit bestand gedefinieerd staan, worden gebruikt bij het opstarten van je ASP.NET core applicatie.

launchSettings.json file wordt enkel gebruikt op je locale development machine. Dit bestand moet dus niet worden gepubliceerd wanneer je applicatie op de productie server wordt gedeployed.

Indien je settings nodig hebt in productie, worden deze in de appsettings.json file gedefinieerd.

ASP.NET Core Applicatie Profile settings in launchSettings.json:

Open launchSettings.json file, standaard staan volgende settings gedefinieerd in een ASP.NET Core 3.1 applicatie:.

```
{  
  
    "iisSettings": {  
  
        "windowsAuthentication": false,  
  
        "anonymousAuthentication": true,  
  
        "iisExpress": {  
  
            "applicationUrl": "http://localhost:50409",  
  
            "sslPort": 0  
  
        }  
  
    },  
  
    "profiles": {
```

```

    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "FirstCoreWebApplication": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

In de bovenstaande **launchSettings.json** file zien we in binnen de sectie Profile 2 secties:

- IIS Express
- FirstCoreWebApplication:


```

"profiles": {
  "IIS Express": {
    "commandName": "IISExpress",
    "launchBrowser": true,
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
  "FirstCoreWebApplication": {
    "commandName": "Project",
    "launchBrowser": true,
    "applicationUrl": "http://localhost:5000",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  }
}

```

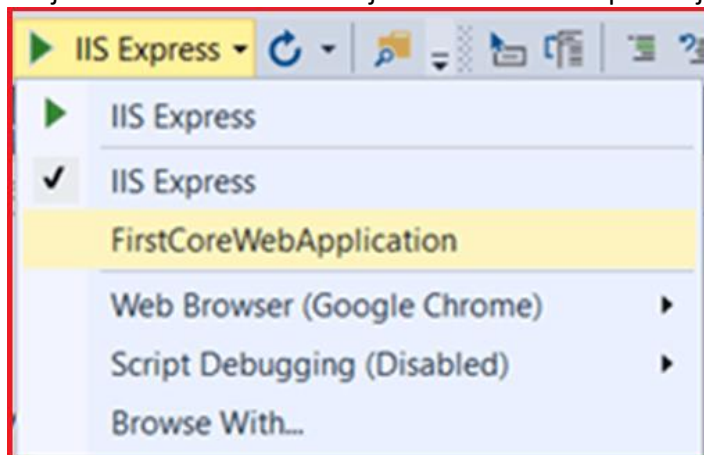
This settings is for IIS Express

This setting is for Kestrel Web Server.

Wanneer je je app runt via **CTRL + F5** of gewoon **F5** dan wordt bij default het profiel met **"commandName": "IISExpress"** gebruikt.

Indien je de ASP.NET Core applicatie runt via .NET Core CLI (dotnet run command), dan wordt het profiel **"commandName": "Project"** gebruikt.

Als je via de IDE VS 2019 kan je kiezen met welk profiel je opstart :

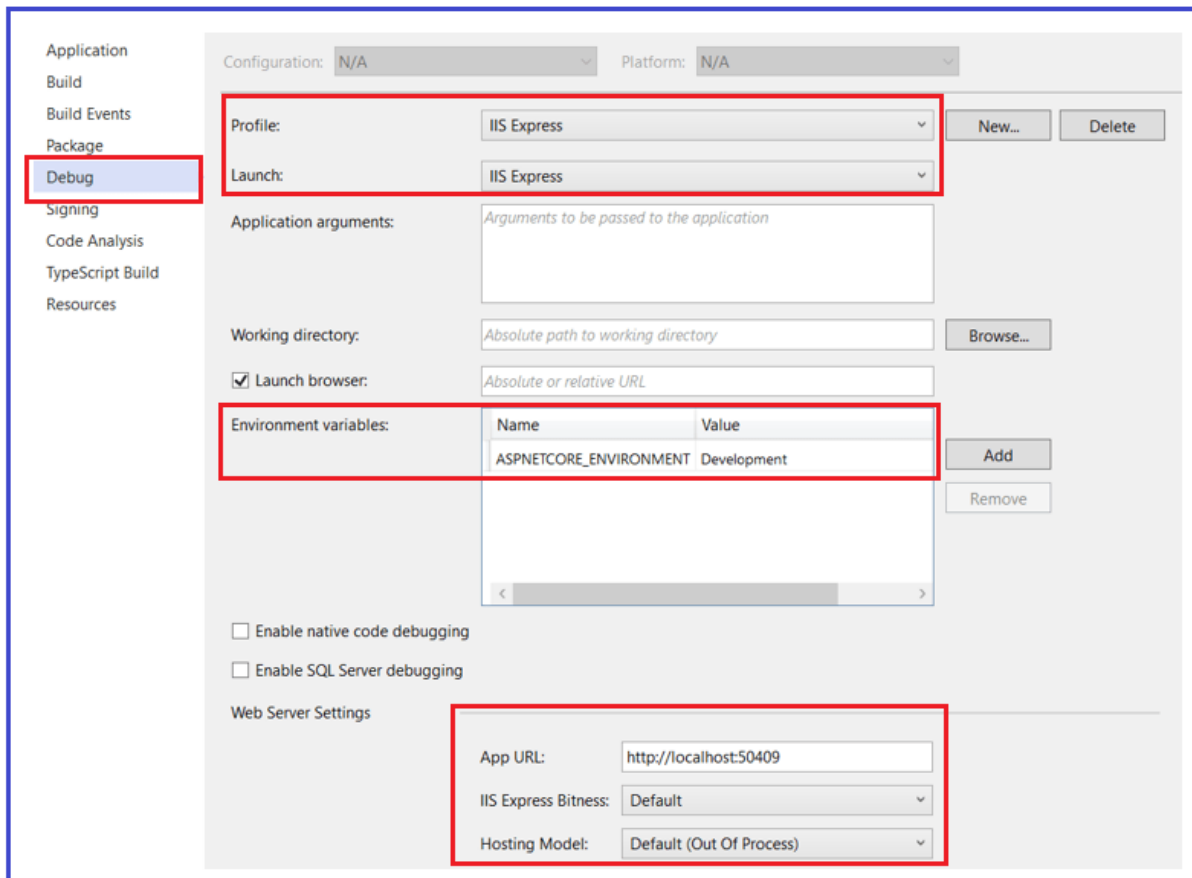


De volgende mogelijkheden zijn aanwezig :

1. **IISExpress**
2. **IIS**
3. **Project**

CommandName	AspNetCoreHostingModel	Internal Web Server	External Web Server
Project	Hosting Setting Ignored	Only one web server is used - Kestrel	
IISExpress	InProcess	Only one web server is used - IIS Express	
IISExpress	OutOfProcess	Kestrel	IIS Express
IIS	InProcess	Only one web server is used - IIS	
IIS	OutOfProcess	Kestrel	IIS

Via de IDE Visual Studio kan je eveneens het default profiel instellen
Open de properties view van je project :



Als je hier aanpassingen verricht, wordt **launchSettings.json** aangepast. Hier zie je eveneens de **Environment Variable "ASPNETCORE_ENVIRONMENT"** die op **"Development"** staat. Deze waarde kan verzet worden op **Staging** of **Production**, afhankelijk waar je app runt. Je kan hier ook zelf environment Variables toevoegen. Deze environment variables zijn beschikbaar binnen je app. Hiermee kan je conditioneel code uitvoeren op basis van de waarde van de environment variable:

bv Kijk in de Configure() methode van de Startup.cs file.

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

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Worker Process Name : " +
                System.Diagnostics.Process.GetCurrentProcess().ProcessName);
        });
    });
}
```

Deze checkt de of de environment variable is Development indien dit het geval is, wordt de Developer Exception Page gebruikt.

9. ASP.NET Core Startup Class

De Startup Class in ASP.NET Core Applicatie:

Een ASP.NET Core application heeft een Startup class. Deze wordt als eerste gestart (na Main method in program) wanneer de app opstart.

De startup class wordt dmv van **UseStartup<T>()** extension methode geconfigureerd:

```
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>();
            });
}
```

De class "Startup" wordt bij default aangemaakt bij aanmaak van een ASP.NET Core app.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method to configure the
    HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                //context.Response.Redirect("/default.html");
                await context.Response.WriteAsync("Hello World!");
            });
        });
    }
}
```

In de Startup class zijn er 2 methoden: **ConfigureServices** en **Configure**. **Configure** methode is vereist. Code in de **ConfigureService** methode is optioneel.

ConfigureServices() methode in ASP.NET Core Startup class

Het **Dependency Injection patroon** wordt standaard gebruikt in de ASP.NET Core architectuur. **ASP.NET Core bevat een built-in IoC container.**

In de **ConfigureServices** methode kan je je **dependent classen registreren met de built-in IoC container**. Nadat een dependent class hier is geregistreerd, kan deze overal binnen je applicatie worden gebruikt. Je moet enkel een constructor parameter voorzien in de classes waar je de geregistreerde dependent class wil aanspreken. De IoC container zal deze automatisch injecteren (via *constructor injection*)

ASP.NET Core verwijst naar de dependent class als zijnde een "Service".

De **ConfigureServices** methode voorziet de **IServiceCollection** parameter om services te kunnen registreren in de IoC container. Bv, indien je RazorPages of MVC Services wil toevoegen aan je ASP.NET Core application, dan gebruik je deze parameter om deze te registreren:

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

Configure() methode in ASP.NET Core Startup class

In de **Configure** methode kan de **request pijplijn** worden gedefinieerd voor onze asp.net core applicatie met behulp van een instantie van **IApplicationBuilder** die reeds voorzien is in de built-in IoC container.

ASP.NET Core heeft verschillende middleware componenten die kunnen worden toegevoegd aan de request pipeline. **De methode Configure wordt bij elke request uitgevoerd.** In deze methoden voeg je enkel de middleware componenten toe die noodzakelijk zijn voor je application.

Standaard worden in een "Empty" ASP.NET Core applicatie de volgende 3 **middleware componenten** toegevoegd in de request pijplijn:

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

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

10. ASP.NET Core appsettings.json file

1. Welke soorten Configuratie-bronnen zijn beschikbaar voor een ASP.NET Core applicatie?
2. Wat is de ASP.NET Core appsettings.json file?
3. Hoe configuratie -informatie aanspreken in een ASP.NET Core Applicatie?
4. Wat is de configuratie-uitvoeringsvolgorde van een ASP.NET Core Application?
5. Wat is de standaard volgorde van uitlezen van configuratie-bronnen?
6. Hoe de Config waarde van de Command Line doorgeven in een ASP.NET Core Applicatie?

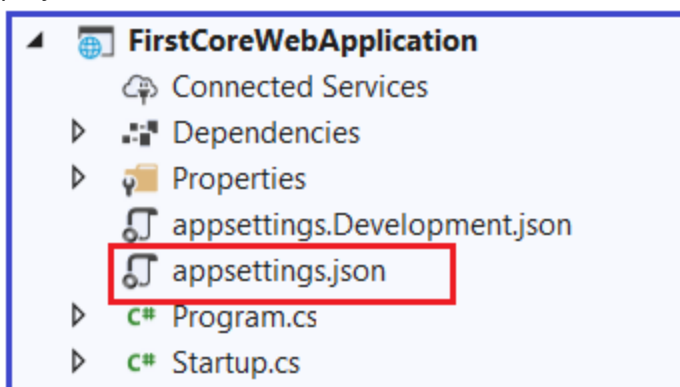
Welke soorten Configuratie-bronnen zijn beschikbaar voor een ASP.NET Core applicatie?

In ASP.NET Core, de applicatie configuratie settings kunnen van **verschillende configuratie-bronnen** komen, zoals:

1. **Bestanden** (appsettings.json, appsettings.{Environment}.json, waar de {Environment} is het huidige hosting environment (Development, Staging of Production)
2. **User secrets**
3. **Environment variabelen**
4. **Command-line argumenten**

Wat is de ASP.NET Core appsettings.json File?

Bij creatie van een ASP.NET Web App genereert VS 2019 een appsettings.json bestand in het project:



De appsettings.json file is een applicatie configuratie file waarin settings worden bijgehouden zoals **database connections strings**, application scope global variables, ...

Indien je de ASP.NET Core appsettings.json file opent, die je reeds een aantal default settings:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Voeg een key met de naam **MyCustomKey** toe aan appSettings.json.

In een JSON file, wordt alles gedefinieerd in key-value paren:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "MyCustomKey": "MyCustomKey waarde uit appsettings.json"
}
```

Hoe configuratie informatie in de ASP.NET Core applicatie opvragen?

Om configuratie info binnen de Startup class op te vragen, hebben we de **IConfiguration service** nodig, die reeds voorzien is door het ASP.NET Core Framework. We injecteren de IConfiguration service via de constructor van de Startup class.

Hiervoor moeten we volgende wijzigingen in de Startup.cs aanbrengen:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```
public class Startup
{
    private IConfiguration _config;
    // Hier gebruiken we Dependency Injection om een Configuration object te injecteren
    public Startup(IConfiguration config)
    {
        _config = config;
    }

    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                // context.Response.Redirect("/default.html");
                //await context.Response.WriteAsync("Hello World! ");
            });
        });
    }
}
```

```

        await context.Response.WriteAsync(_config["MyCustomKey"]);
    });
}
}

```

Uitleg van de bovenstaande code:

Eerst creëren we een private variable van type **IConfiguration _config** (de IConfiguration interface behoort tot de namespace *Microsoft.Extensions.Configuration* namespace). Dan kunnen we d.m.v constructor dependency injection het IConfiguration object injecteren en toewijzen aan de private variabele **_config**:

```

private IConfiguration _config;
public Startup(IConfiguration config)
{
    _config = config;
}

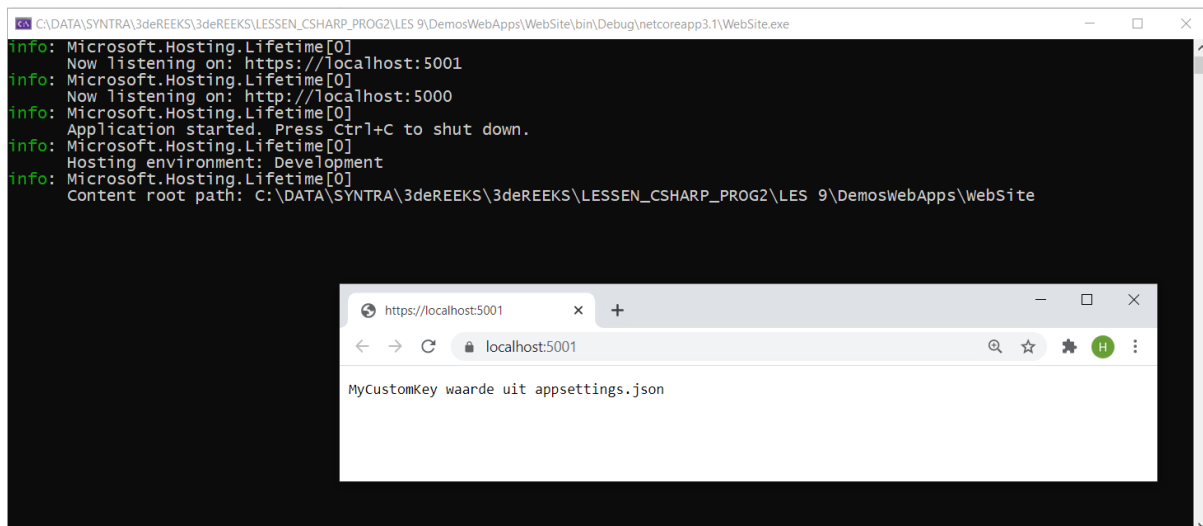
```

We kunnen nu in de Startup class de waarde van de setting **MyCustomKey** opvragen via deze **IConfiguration** service instantie:

```

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync(_config["MyCustomKey"]);
    });
});

```



Dependency Injection Design Patroon

In vorige versies van ASP.NET was Dependency Injection optioneel. Indien je dit wou toevoegen, moest je een framework installeren, zoals bv Ninject, StructureMap, IUnity container,...

In ons ASP.NET Core 3.1 applicatie is er built-in support voor Dependency Injection.

Het **Dependency Injection Design Patroon** laat developers toe om los-gekoppelde systemen te ontwikkelen die gemakkelijk uitbreidbaar en te testen zijn.

Wat is de Configuratie-uitvoeringsvolgorde in een ASP.NET Core Applicatie?

In een ASP.NET core app wordt standaard een appsettings.Development.json file aangemaakt bij creatie van het project:

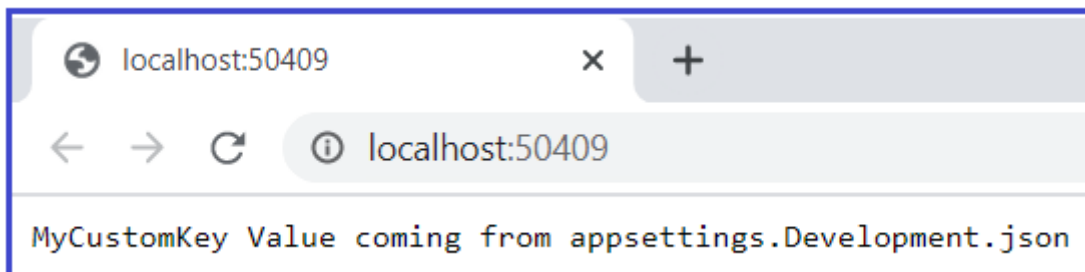


Pas de appsettings.Development.json aan als volgt:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "MyCustomKey": "MyCustomKey waarde uit appsettings.Development.json"
}
```

We gebruiken hier dezelfde setting MyCustomKey als in appsettings.json file.

Start je app op en je krijgt het volgende resultaat :



Zoals je ziet wordt de MyCustomKey waarde uit appsettings.Development.json file teruggegeven.

Wat is de standaard volgorde van uitlezen van configuratie-bronnen?

In de volgende volgorde worden de configuraties uitgelezen :

1. **appsettings.json**,
2. **appsettings.{Environment}.json** bv **appsettings.development.json**
3. **User secrets**
4. **Environment variabelen**
5. **Command-line argumenten**

Momenteel hebben we een setting **MyCustomKey** op twee plaatsen gedefinieerd:
appsettings.json and **appsettings.development.json**.

Voeg nu dezelfde setting **"MyCustomKey"**: **"MyCustomKey waarde uit Environment Variable van launchesSettings.json"** toe via je project properties:

The screenshot shows the 'Debug' tab of the Visual Studio Project Properties dialog. The 'Environment variables' section is expanded, showing a table with two columns: 'Name' and 'Value'. The table contains two entries: 'ASPNETCORE_ENVIRONMENT' with value 'Development' and 'MyCustomKey' with value 'MyCustomKey waarde uit Environment'. The 'MyCustomKey' entry is selected. To the right of the table are 'Add' and 'Remove' buttons. Below the table are checkboxes for 'Enable native code debugging' and 'Enable SQL Server debugging'. At the bottom, there is a 'Web Server Settings' section with an 'App URL' field containing 'https://localhost:5001;http://localhost:5000'.

aan de IIS Express profile sectie van de launchSettings.json file :

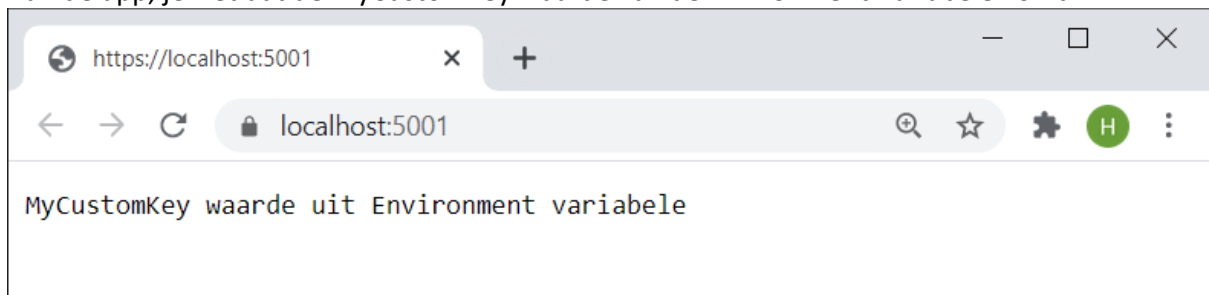
```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:59798",
      "sslPort": 44314
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "MyCustomKey": "MyCustomKey waarde uit Environment variabele"
      }
    }
  }
}
```

```

    }
  },
  "WebSite": {
    "commandName": "Project",
    "launchBrowser": true,
    "applicationUrl": "https://localhost:5001;http://localhost:5000",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development",
      "MyCustomKey": "MyCustomKey waarde uit Environment variabele"
    }
  }
}
}
}

```

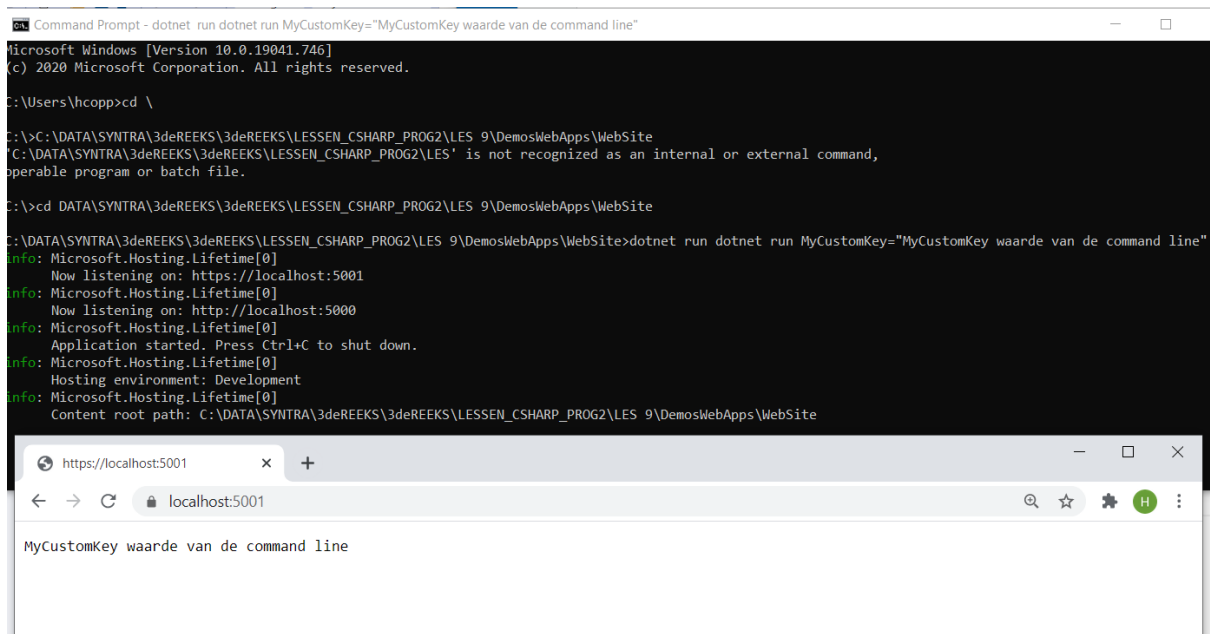
Run de app, je ziet dat de MyCustomKey waarde van de Environment Variabele komt :



Hoe de config-waarde doorgeven via de Command Line in een ASP.NET Core Applicatie?

Open een command prompt en ga naar de locatie waar je .csproj file van je project staat.

Voer het commando **dotnet run MyCustomKey="MyCustomKey waarde van de command line"**;



Open de browser en typ de juiste url. Je krijgt nu de waarde terug die je ingegeven hebt via de command line.

11. ASP.NET Core Middleware Componenten

1. **Wat zijn ASP.NET Core Middleware Componenten?**
2. **Waar gebruiken we de Middleware Componenten in een ASP.NET Core applicatie?**
3. **Hoe Middleware Componenten in een ASP.NET Core applicatie configureren?**
4. **Hoe werken de Middleware Componenten in ASP.NET Core?**
5. **Wat is de volgorde van uitvoering van Middleware Componenten in ASP.NET Core?**
6. **Wat zijn Request Delegates in ASP.NET Core?**
7. **Waarvoor dienen de Use, Run, and Map methoden in ASP.NET Core?**
8. **Wat is de UseDeveloperExceptionPage Middleware Component?**
9. **Hoe Middleware Componenten configureren via de Run() en Use() extension methoden?**
10. **Wat is het verschil tussen de MapGet en Map methoden?**

Wat zijn ASP.NET Core Middleware Componenten?

De ASP.NET Core Middleware Componenten zijn de software componenten (technisch gezien zijn componenten gewoon C# klassen) die in de applicatie pijplijn worden geschakeld om HTTP requests te verwerken en responses terug te geven. Elke middleware component in ASP.NET Core Application vervult de volgende taken:

1. **Beslist of al dan niet de HTTP Request wordt doorgegeven naar de volgende component** in de pijplijn dmbto the next component in the pipeline. Het doorgeven gebeurt via de **next()** methode in de middleware component.
2. Kan werk uitvoeren vóór en na de volgende component in de pijplijn.

In ASP.NET Core zijn er reeds vele built-in Middleware componenten beschikbaar die je rechtstreeks kan gebruiken.

Waar gebruiken we Middleware Componenten in een ASP.NET Core applicatie?

De volgende veel gebruikte built-in Middleware componenten kunnen bv in een ASP.NET Core applicatie aan de pijplijn worden toegevoegd:

1. Middleware component voor **authenticatie** van een user
2. Middleware component voor **logging** van request en response
3. Middleware component om **errors** af te handelen
4. Middleware component om requests voor **static files** af te handelen, zoals bv images, Javascript, CSS files,
5. Middleware component voor **Authorisatie** van users voor toegang toe specifieke resources

Deze Middleware componenten zijn de meest gebruikte componenten die worden ingeschakeld in de request processing pipeline van een ASP.NET Core application. Er worden HTTP Handlers en HTTP Modules gebruikt om de request processing pipeline op te zetten. Deze pipeline zal bepalen hoe de HTTP request en response zal worden verwerkt.

Hoe Middleware Componenten in ASP.NET Core applicatie configureren?

In een ASP.NET Core applicatie, worden de Middleware componenten geconfigureerd in de **Configure()** methode van de Startup class.

Bij een asp.net web app aangemaakt met een Empty Template bevat de **Configure methode** in **Startup class** te volgende code;

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.  
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
    }  
  
    app.UseRouting();  
  
    app.UseEndpoints(endpoints =>  
    {  
        endpoints.MapGet("/", async context =>  
        {  
            await context.Response.WriteAsync("Hello World!");  
        });  
    });  
}
```

So, whenever you want to configure any middleware components in any type of .net core applications, then you need to configure it within the Configure() method of the Startup class by calling the **Use*** methods on the IApplicationBuilder object. As you can see in the above image, the configuration() method sets up the request processing pipeline with just three middleware components are as follows.

1. **UseDeveloperExceptionPage() Middleware component**
2. **UseRouting() Middleware component**
3. **UseEndpoints() Middleware component**

Before understanding the above three built-in Middleware components. Let us first understand what are Middleware components and how exactly these Middleware components work in an ASP.NET Core application.

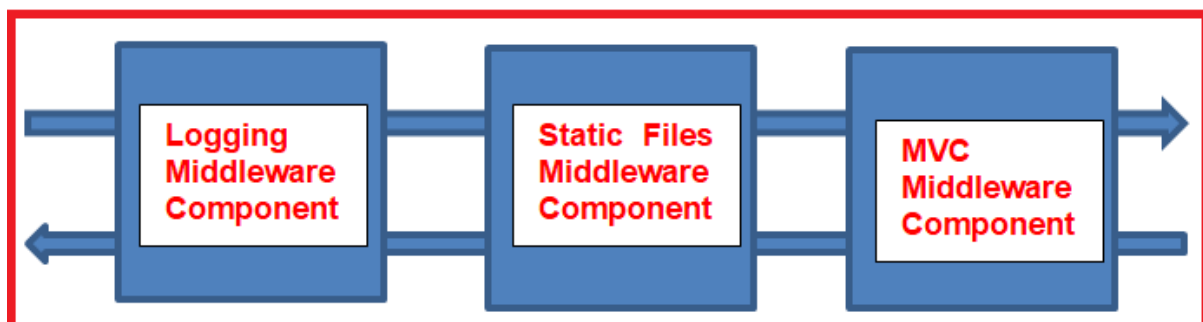
Hoe werken Middleware Components in ASP.NET Core?

In ASP.NET Core applicatie heeft de Middleware component toegang tot de binnenkomende HTTP Request en uitgaande HTTP Response.

Een Middleware component in ASP.NET Core kan:

1. De binnenkomende HTTP request afhandelen door een HTTP response te genereren.
2. De binnenkomende HTTP request verwerken, deze aanpassen en dan doorgeven aan de volgende middleware component in de pijplijn
3. De uitgaande HTTP response verwerken, deze aanpassen en dan deze ofwel doorgeven aan de volgende middleware component of aan de ASP.NET Core web server.

Het volgende toont een schema van middleware componenten die in de request processing pipeline van een ASP.NET Core applicatie zijn:



In het bovenstaande schema is er eerst een logging middleware component. Deze logt bv enkel de request tijd en geeft dan de HTTP request door naar de volgende middleware component. De 2de component is de Static Files Middleware component die een response geeft voor een request van een static file (image, css, javascript bv)

Elke component kan beslissen of de request wordt doorgegeven naar de volgende component ofwel dit niet te doen, dit wordt dan “short-circuiting de request pipeline”. Dit laatste gebeurt door de Static files component wanneer er een binnenkomende HTTP request is voor een image, css of javascript.

De laatste component in het schema is een MVC Middleware component.

Een middleware component kan zowel een HTTP request als response in de pijplijn verwerken of wijzigen. Bv, kan de 1ste logging middleware component eveneens de tijd loggen wanneer de response wordt teruggestuurd naar de eindgebruiker.

Wat is de volgorde van uitvoering van de Middleware Componenten in een ASP.NET Core Applicatie?

De **volgorde van uitvoering van de middleware componenten** is belangrijk, want ASP.NET Core middleware componenten worden uitgevoerd **in dezelfde volgorde als dat ze gedefinieerd staan in de pipeline (Configure methode van Startup class)**.

Niet alle middleware componenten heb je voor alle web apps nodig.

Indien je bv een static web applicatie met enkel statische HTML pagina's en images maakt, dan heb je bv enkel de component “StaticFiles” in de pipeline nodig.

Maar als je een beveiligde dynamische data-driven web app wil ontwikkelen, zal je meerdere middleware componenten in de pijplijn moeten schakelen, zoals bv Logging Middleware, Authentication middleware, Authorization middleware, MVC middleware,

Wat zijn Request Delegates in ASP.NET Core?

Request delegates worden in ASP.NET core gebruikt om de request pipeline te bouwen, dwz dat request delegates gebruikt worden om elke inkomende HTTP request af te handelen. In ASP.NET Core kan je de Request delegates configureren via de **Run, Map, en Use** extension methoden. Je kan een request delegate specificeren via een in-line anonymous methode (in-line middleware genoemd) of je kan de request delegates specificeren via een herbruikbare class. Deze herbruikbare classes en in-line anonymous methoden worden middleware of middleware componenten genoemd. Elke middleware component in de request processing pipeline is verantwoordelijk voor de beslissing voor het aanroepen van de volgende component in de pipeline of short-circuiting the pipeline.

Waarvoor dienen de Use en Run methoden in een ASP.NET Core Web Applicatie?

In ASP.NET Core kan je de methoden “**Use**” en “**Run**” gebruiken om de Inline Middleware component te registreren in de Request processing pipeline. De “**Run**” extension methode laat toe om de eind-component te definiëren (deze middleware zal geen volgende middleware componenten aanroepen). De “**Use**” extension methode laat toe om componenten te definiëren die de volgende component in de request processing pipeline zal aanroepen. Indien je de code in de **Configure methode** bekijkt, zie je dat deze een instantie van het type **IApplicationBuilder** interface en deze instantie wordt samen gebruikt met de methoden Use en Run om de Middleware componenten te configureren.

In de **configure** methode, staan 3 middleware componenten reeds geregistreerd in de request processing pipeline dmv de `IApplicationBuilder` instantie:

1. **UseDeveloperExceptionPage() Middleware component**
2. **UseRouting()**
3. **UseEndpoints() Middleware component**

DeveloperExceptionPage Middleware Component:

In de `configure` method, wordt de **UseDeveloperExceptionPage()** middleware component geregistreerd in de pipeline. Deze middleware component wordt enkel ingeschakeld wanneer de hosting environment in "development" staat. Deze middleware component zorgt dat bij een onafgehandelde exception de lijn code toont die de exception veroorzaakt.

UseRouting() Middleware Component:

Deze middleware component voegt Endpoint Routing Middleware toe aan de pipeline dwz dat het de URL (of binnenkomende HTTP Request) mapt op een specifieke resource.

UseEndpoints() Middleware Component:

In deze middleware worden routing beslissingen genomen met behulp van de **Map** of **MapGet** extension methoden. Met deze extension methoden kan een URL pattern zoals "/" worden gespecificeerd. "/" patroon betekent enkel het domein-naam. Elke request met enkel de domain naam als URL zal worden verwerkt door deze component.

De default code van **UseEndpoints** middleware component bij creatie van een "Empty" web applicatie is als volgt:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
});
```

In plaats van **MapGet**, kan je eveneens de **Map** methode gebruiken:

```
app.UseEndpoints(endpoints =>
{
    endpoints.Map("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
});
```

Run de app en je krijgt de tekst «Hello World ! » in de browser te zien

Wat is het verschil tussen MapGet en Map methoden?

De **MapGet** methode is handelt enkel de **GET** HTTP Requests af, terwijl de **Map** methode is alle types van HTTP requests afhandelt, zoals bv **GET, POST, PUT, DELETE,**

Hoe de Middleware Componenten configureren dmv de Run() extensie methode?

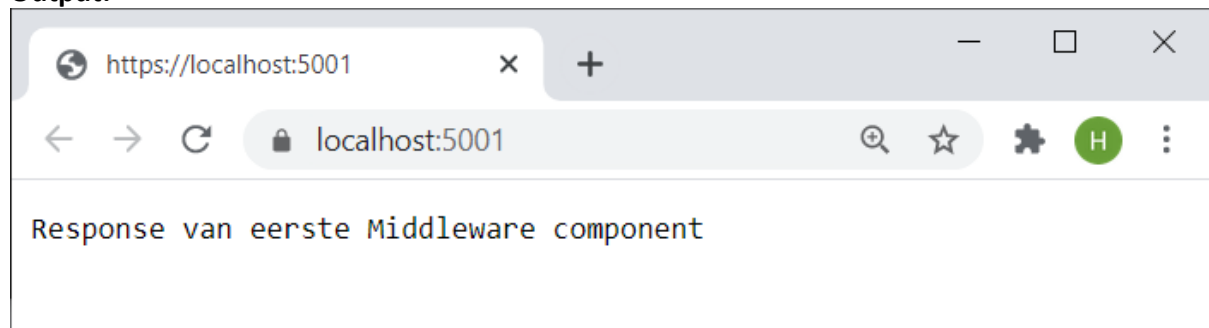
Hoe schrijven en configureren we **custom** middleware componenten dmv de **Run** extension methode.

In de volgende code, wordt bv een custom middleware component gebruikt die via de Run in de Configure methode aangemaakt en geconfigureerd wordt. Deze custom component doet niet anders dan een tekst als HTTP Response teruggeven:

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Response van eerste Middleware component");
    });
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            // context.Response.Redirect("/default.html");
            //await context.Response.WriteAsync("Hello World! ");
            await context.Response.WriteAsync(_config["MyCustomKey"]);
        });
    });
}
```

Output:



Via de aanroep van de **Run() extension** methode op het **IApplicationBuilder** object wordt de custom middleware component geregistreerd in de HTTP request verwerkings-pijplijn.

Dit is de definitie van de **Run** methode:

```
...public static class RunExtensions
{
    ...public static void Run(this IApplicationBuilder app, RequestDelegate handler);
}
```

De Run() methode, is geïmplementeerd als een extension methode van de IApplicationBuilder interface.

De 2de parameter van de Run() methode neemt een input parameter van type **RequestDelegate**.

Dit is de definitie van RequestDelegate:

```
...public delegate Task RequestDelegate(HttpContext context);
```

De RequestDelegate is een delegate dat een input parameter van het type **HttpContext** neemt.

Een middleware component in een ASP.NET Core applicatie heeft zowel toegang tot de HTTP Request als Response dankzij het HttpContext object.

Het HttpContext object bevat zowel de informatie van de http Request als de http response die zal worden teruggegeven

In onze custom middleware component (zie hierboven), geven we de Requestdelegate door via een inline anonymous methode met een lambda expressie. En we geven het HttpContext object als input parameter aan de request delegate:

```
app.Run(async
    HttpContext object
    (context) =>{
        await context.Response.WriteAsync("Hello world!");
    }
);
    Anonymous Delegate
```

Opmerking: In plaats van een anonymous methode kan de request delegate ook gedefinieerd worden in een aparte class.

Nog een middleware component toevoegen.

Voeg de volgende lijnen code toe aan configure om een 2de middleware component toe te voegen :

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Response van eerste Middleware component");
    });
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Response van 2de Middleware");
    });
}
```

```

        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                // context.Response.Redirect("/default.html");
                //await context.Response.WriteAsync("Hello World! ");
                await context.Response.WriteAsync(_config["MyCustomKey"]);
            });
        });
    }
}

```

Wanneer we de app runnen, krijgen we echter nog steeds de tekst **“Response van eerste Middleware”**

Hoe komt dit? Dit komt omdat de eerste middleware component de Run() extensie methode gebruikt en wordt hierdoor een eind-component, dwz de eerste component roept niet de volgende component aan in de pijplijn.

Configureren van middleware component via de Use extensie methode

Hoe kan je nog de volgende component in de pijplijn aangroepen? Dit kan via de **Use** extensie methode.

Wijzig de **Run** van de eerste component naar **Use**, voeg een 2de parameter **next** toe aan de anonymous methode en voeg de lijn **await next()**; toe op het einde van de eerste component:

```

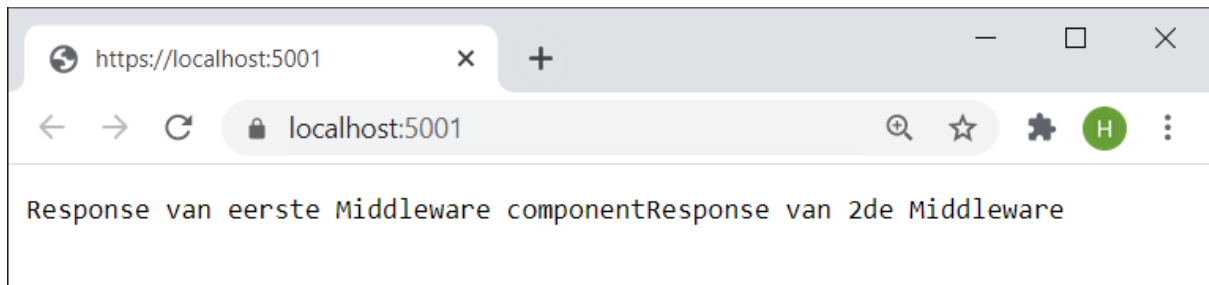
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Response van eerste Middleware component");
        await next();
    });
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Response van 2de Middleware");
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            // context.Response.Redirect("/default.html");
            //await context.Response.WriteAsync("Hello World! ");
            await context.Response.WriteAsync(_config["MyCustomKey"]);
        });
    });
}

```

Run de app en nu zie je dat eerst de text van de eerste component en daarna de text van de 2de component wordt teruggegeven :



Uitleg over Use extension methode:

De **Use** extension methode voegt een in-line gedefinieerde middleware delegate toe aan de request pijplijn.

De **Use** methode is als volgt gedefinieerd:

```
public static class UseExtensions
{
    public static IApplicationBuilder Use(
        this IApplicationBuilder app,
        Func<HttpContext, Func<Task>, Task> middleware
    );
}
```

De **Use** methode is eveneens gedefinieerd als extension methode van de **IApplicationBuilder** interface. Dit is de reden waarom we deze kunnen aanroepen op de **IApplicationBuilder** instantie. De **Use** methode neemt 2 input parameters.

De eerste parameter is het **HttpContext** object waarmee deze toegang heeft tot zowel de HTTP request als HTTP response.

De tweede parameter heeft een **Func** type dit is een generic built-in delegate die zowel de Http request kan afhandelen, ofwel de volgende middleware compenten kan aanroepen in de request pijplijn.

Indien je een HTTP request van een middleware component naar de volgende middleware component wil doorgeven, dan moet je de **next()** methode aanroepen.

12. De ASP.NET Core Request Processing Pipeline

1. **Wat is de ASP.NET Core Request Processing Pipeline?**
2. **Hoe meerdere middleware componenten maken en registreren in de pipeline?**
3. **Wat is de uitvoeringsvolgorde van middleware componenten in de request processing pipeline?**

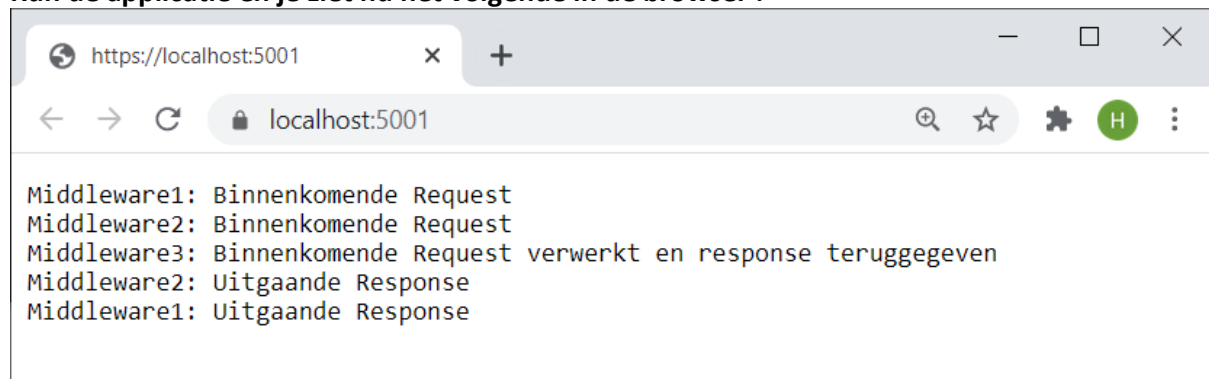
Wat is de ASP.NET Core Request Processing Pipeline?

Als voorbeeld plaatsen we **3 middleware componenten in de pipeline** door de volgende code in de **Configure()** methode van de **Startup** class te plaatsen:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Middleware1: Binnenkomende Request\n");
        await next();
        await context.Response.WriteAsync("Middleware1: Uitgaande Response\n");
    });
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("Middleware2: Binnenkomende Request\n");
        await next();
        await context.Response.WriteAsync("Middleware2: Uitgaande Response\n");
    });
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Middleware3: Binnenkomende Request
verwerkt en response teruggegeven\n");
    });
}
```

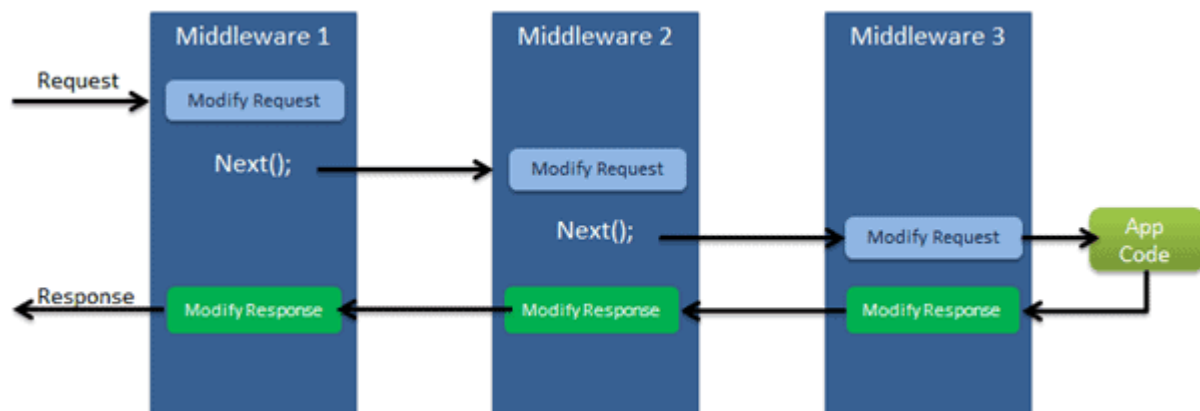
De eerste 2 middleware componenten zijn geregistreerd dmv de **Use()** extension methode en roepen dan via de **next()** methode de volgende component aan. De 3de component is geregistreerd via de **Run()** extension methode. Aangezien het een **eind-** component is, dwz deze component zal geen volgende component meer aanroepen.

Run de applicatie en je ziet nu het volgende in de browser :



Wat is de uitvoeringsvolgorde van de ASP.NET Core Request Processing Pipeline?

Het volgende diagram illustreert de volgorde van uitvoering van de componenten in de request pijplijn :



Bron: <https://www.tutorialsteacher.com/core/aspnet-core-middleware>

De binnenkomende HTTP request zal eerst ontvangen worden door de eerste middleware component d.i. Middleware1 , deze geeft de tekst “**Middleware1: Binnenkomende Request**” in de response stream. Dit is de eerste tekst die in de browser verschijnt.

De 1ste component roept de 2de aan via next() method , d.i. Middleware2.

De 2de component geeft de tekst “**Middleware2: Binnenkomende Request**” Deze is de 2de tekst die in de browser verschijnt.

De 2de component roept de 3de aan via next(), d.i. Middleware3.

De 3de component is een eindcomponent, deze geeft de tekst . “**Middleware3: Binnenkomende Request verwerkt en response teruggegeven**”.Dit is de 3de lijn die in de browser verschijnt.

Vanaf dit moment wordt de pijplijn omgekeerd. Vanaf de eind-component wordt nu de control teruggegeven naar de 2de component, deze geeft dan de volgende tekst “**Middleware2: Uitgaande Response**” de 2de component geeft dan de controle terug door aan de eerste die dan de laatste lijn tekst “**Middleware1: Uitgaande Response**” teruggeeft.

Samenvatting:

1. De ASP.NET Core request processing pipeline bestaat uit een reeks van middleware componenten die opeenvolgende worden aangeroepen.
2. Elke middleware component kan bepaalde operaties vóór en na het aanroepen van de volgende component verrichten. Een middleware component kan ook beslissen om niet de volgende component aanroepen, dit wordt “short-circuiting de request pipeline” genoemd.
3. De middleware component in asp.net core heeft zowel toegang tot de binnenkomende Request als de uitgaande Response.
4. De volgorde waarin de middleware componenten worden geregistreerd in de **Configure** methode van de Startup class definieert de volgorde waarin de middleware componenten zullen worden uitgevoerd bij binnenkomende HTTP requests. De componenten worden terug in omgekeerde volgorde uitgevoerd voor de response.
5. Deze volgorde van componenten is dus van belang voor de security, prestatie, en functionaliteit van de applicatie.

13. wwwroot folder in ASP.NET Core

Wat is de wwwroot folder in ASP.NET Core?

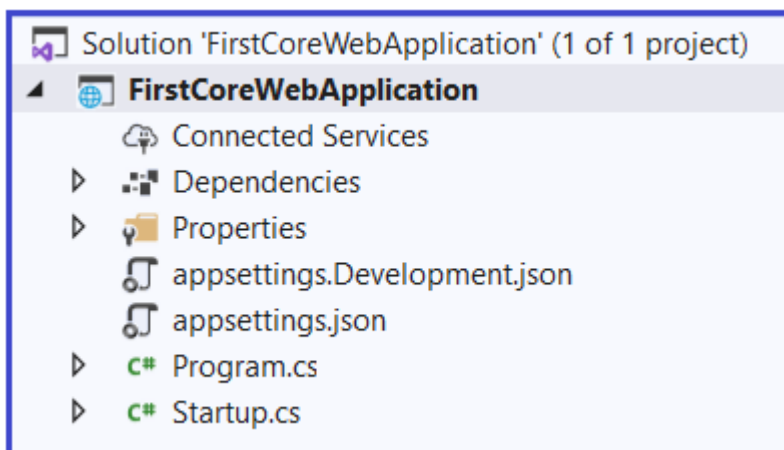
In een web app wordt de wwwroot folder als de “root” beschouwd. Deze folder moet in de root van je ASP.NET web app worden geplaatst. In ASP.NET Core Application kunnen Statische files in gelijk welke subfolder onder de webroot folder worden geplaatst en kunnen aangesproken worden dmv een relatief pad ten op zichte van deze wwwroot root folder.

Adding the wwwroot (webroot) folder in ASP.NET Core Application:

Wanneer je een ASP.NET Core Web Application met de MVC Template aanmaakt, wordt de wwwroot folder standaard aangemaakt in de root van je project folder.

Indien je een ASP .NET Core Application met de Empty template aanmaakt wordt de wwwroot folder echter niet automatisch aangemaakt en moet je de wwwroot folder zelf aanmaken.

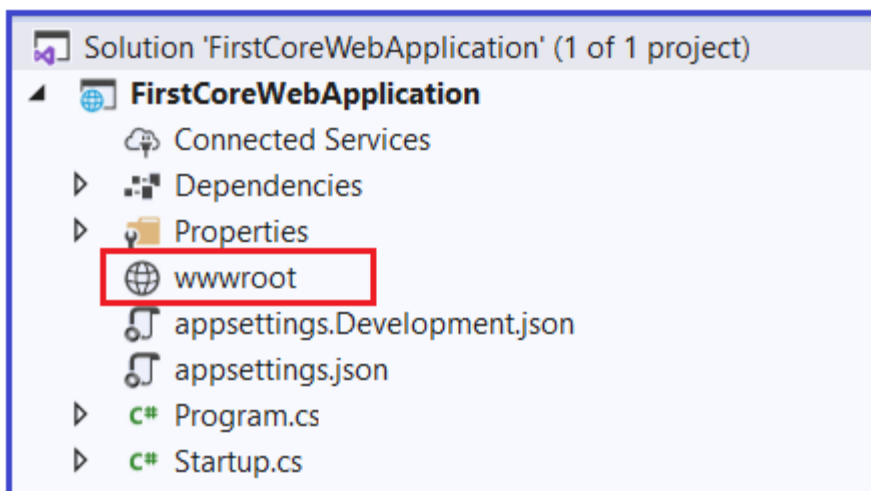
Maak een ASP.NET Core Application met de naam “FirstCoreWebApplication” met de **Empty** project template. De volgende project structuur zie je in de Solution Explorer:



Toevoegen van wwwroot (webroot) folder in ASP.NET Core:

Rechtsklik met de muis op de naam van je project in Solution Explorer en selecteer de **add => new folder** optie en geef de folder naam **wwwroot**.

Nu zie je wwwroot met een speciaal icoontje ervoor in de Solution Explorer :



Wat de wwwroot (webroot) folder bevat in ASP.NET Core:

In ASP.NET Core kunnen enkel statische files die in de webroot – wwwroot folder of subfolder van de wwwroot aangesproken worden dmv een HTTP request. Alle ander files kunnen niet standaard worden aangesproken.

Het is wel mogelijk om dit manueel aan te passen.

Meestal worden standaard de volgende subfolders en static files onder wwwroot folder geplaatst:



Nu kunnen statiche files zoals CSS, js, lib vanaf de basis URL via hun bestandsnaam worden aangesproken, bv via een browser.

Bv de site.js file in de js folder kan aangesproken worden via de

url: **https://localhost:<port>/js/site.js**

Opmerking: Om statische files te kunnen aanspreken, moet je de middleware component in de Configure() methode van de Startup class registreren via [app.UseStaticFiles\(\)](#)

Is het mogelijk om de wwwroot Folder te hernoemen?

Ja, dit is mogelijk, bijvoorbeeld ipv wwroot wil je als folder naam bv **MyWebRoot**, maar dan moet je de volgende code aanpassen in de Program.cs :

```
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>().UseWebRoot("MyWebRoot");
            });
}
```

14. Statische Files Middleware in ASP.NET Core

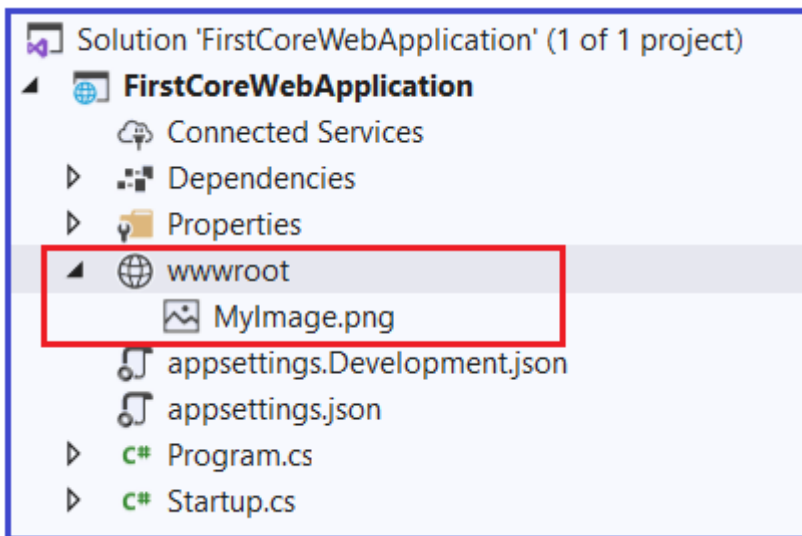
1. **Waar worden de statische files in ASP.NET Core geplaatst?**
2. **Hoe de Static Files Middleware configureren ASP.NET Core Web Application?**

Statische files zijn bv HTML, Images, CSS, JavaScript,.. maar ASP.NET Core kan deze niet bij default leveren. Er is een configuratie nodig in je ASP.NET Core project om toe te laten om toegang te geven.

Waar worden de statische files in een ASP.NET Core project geplaatst?

De default lokatie is **wwwroot** (de webroot) folder of een subfolder onder de wwwroot.

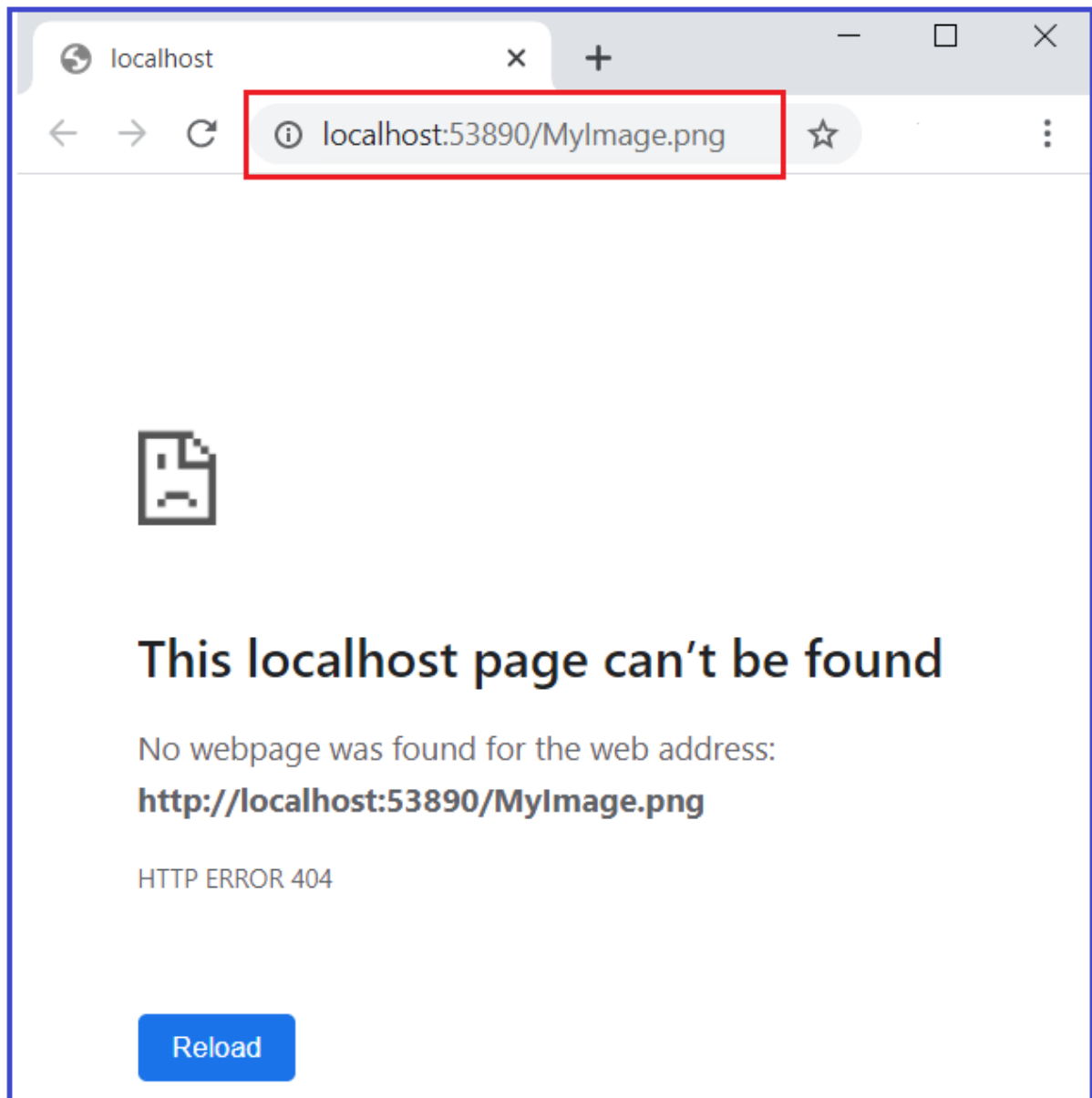
Bv voeg een image toe onder de **wwwroot**:



Run je app en probeer te navigeren naar de image via de volgende URL:

<http://localhost:<portnumber>/MyImage.png>

Dit lukt niet, je krijgt de volgende boodschap in de browser :



Dit komt omdat we de Static Files middleware component nog niet hebben geregistreerd in de Configure() methode in Startup.cs.

Hoe de Static Files Middleware component configureren in een ASP.NET Core Applicatie?

We hebben de middleware component **UseStaticFiles()** nodig in de Request pipeline om statische files, zoals bv images, html pagina's, javascript en .css te kunnen opvragen aan je ASP.NET core App via bv een browser.

De Request pipeline wordt gedefinieerd in de Configure() methode van de Startup class
Voeg de Static Files middleware component toe aan de request pijplijn via **UseStaticFiles()**:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    //Toevoegen van Static Files Middleware component om statische files bv HTML, images, .css en .js toe te
    laten
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            // context.Response.Redirect("/default.html");
            //await context.Response.WriteAsync("Hello World! ");
            await context.Response.WriteAsync(_config["MyCustomKey"]);
        });
    });
}
```

Run opnieuw de app en probeer nogmaals naar de

URL: <http://localhost:<portnumber>/MyImage.png> te navigeren via een browser. Nu wordt de image wel getoond in de browser

15. Configuratie van Default Page in ASP.NET Core

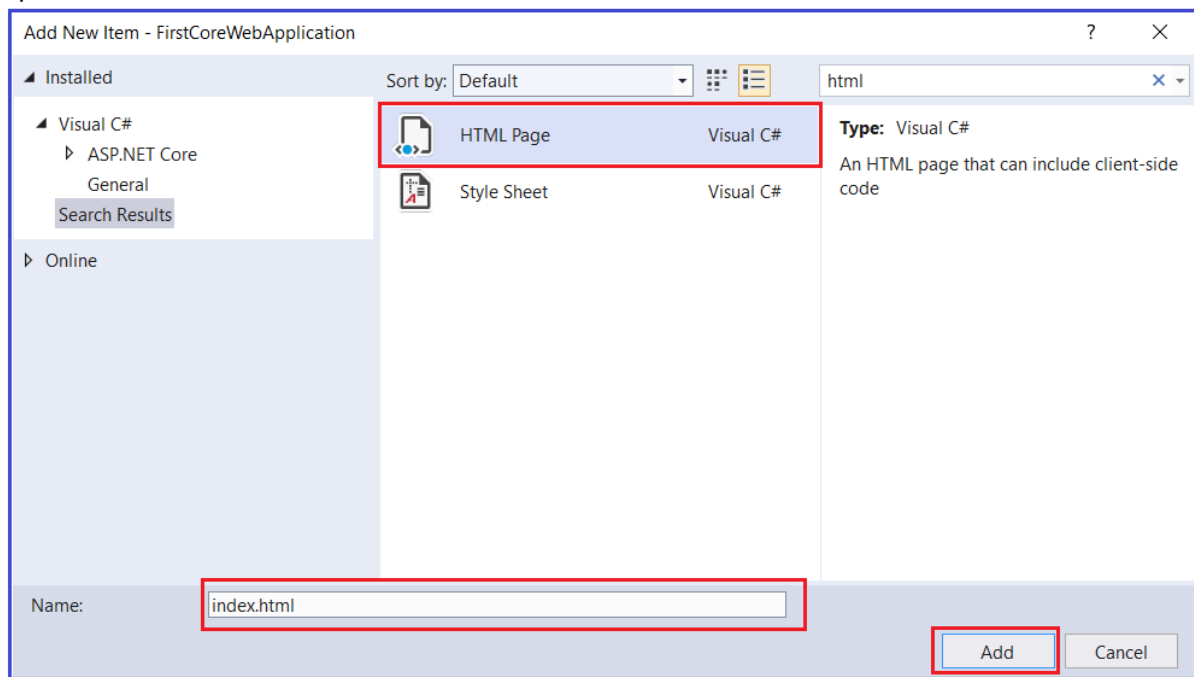
Hoe de Default Page configureren in ASP.NET Core:

Maak een ASP.NET Core Web applicatie aan met als naam *FirstCoreWebApplication* met de **Empty** project template

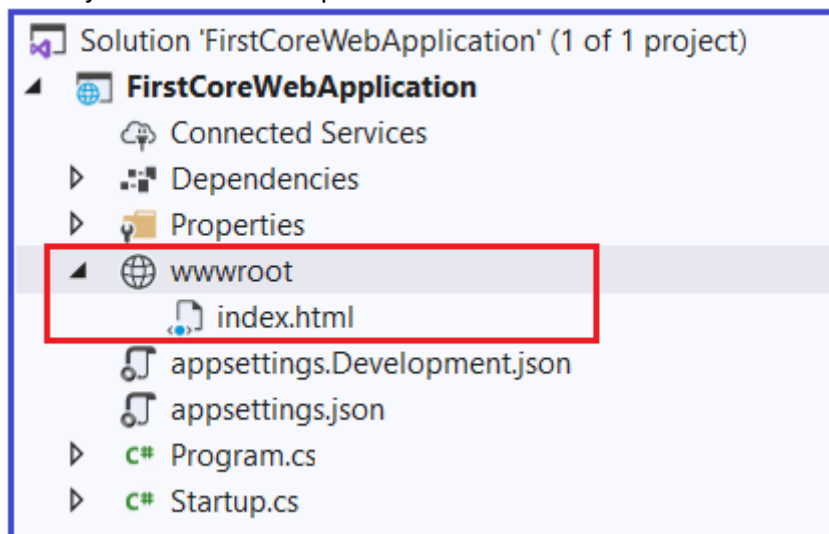
Voeg een folder wwwroot toe aan de root van je project

Voeg onder de wwwroot een HTML pagina toe:

Rechtsklik met de muis op de foldere **wwwroot** folder en selecteer **add => new item** om de “new item” window te openen. Selecteer **HTML template**, geeft als naam “**index.html**” en klik op de **Add** button:



Nu zie je in de solution Explorer onder de wwwroot de index.html :



Open de **index.html** onder de wwwroot folder en plaats hierin de volgende HTML code:

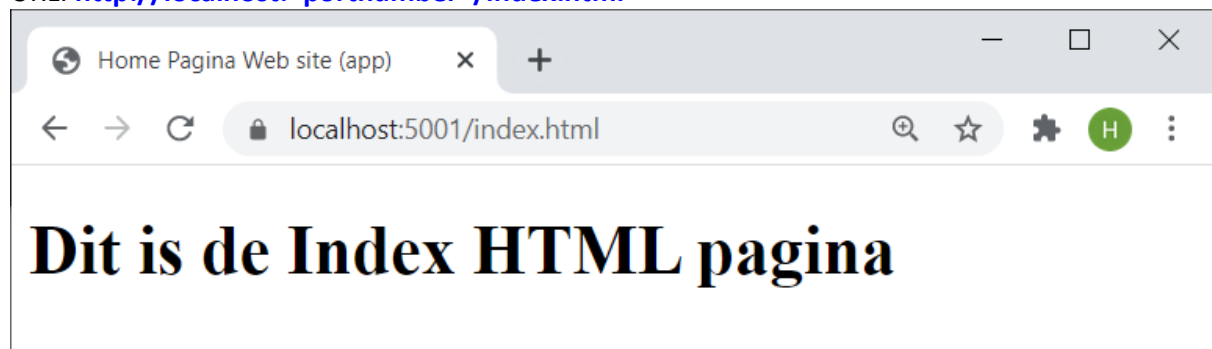
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Home Pagina Web site (app)</title>
</head>
<body>
    <h1> Dit is de Index HTML pagina </h1>
</body>
</html>
```

Static Files middleware component toevoegen aan de Configure Methode van de Startup class:

Voeg deze component toe aan de request pipeline dmv **app.UseStaticFiles()**

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    //Setting the Default Files
    app.UseDefaultFiles();
    //Adding Static Files Middleware to serve the static files
    app.UseStaticFiles();
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Request afgehandeld met eind-middleware component");
    });
}
```

Run de app en navigeer in een browser naar de URL: <http://localhost:<portnumber>/index.html>



Hoe een custom HTML Page als de Default Pagina zetten?

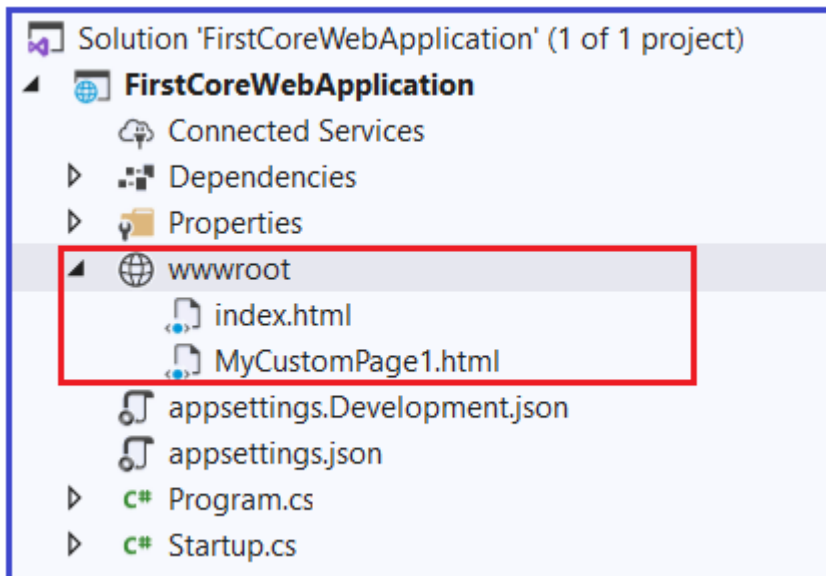
De **UseDefaultFiles()** middleware zoekt in de **wwwroot** folder naar de volgende files:

1. **index.htm**
2. **index.html**
3. **default.htm**
4. **default.html**

Dit is het standaardgedrag.

Hoe kan je de default pagina zetten op bv MyCustomPage1.html ?

Voeg een HTML pagina met naam MyCustomPage1.html toe onder de wwwroot:



Open MyCustomPage1.html file en plaats hierin de volgende code :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <h1>Mijn custom home pagina</h1>
</body>
</html>
```

Zet MyCustomPage1.html als Default Pagina:

Pas in Configure() methode van de Startup class de volgende code:

Via de DefaultFileNames property van de DefaultFilesOptions class kan je een default filenames toevoegen, bv "MyCustomPage1.html".

Dan pas roep je de UseDefaultFiles middleware component aan:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    //Specifeer MyCustomPage1.html als de default page
    DefaultFilesOptions defaultFilesOptions = new DefaultFilesOptions();
    defaultFilesOptions.DefaultFileNames.Clear();
    defaultFilesOptions.DefaultFileNames.Add("MyCustomPage1.html");
    //Zet de Default Files
    app.UseDefaultFiles(defaultFilesOptions);
    //Voeg de Static Files Middleware component toe:
    app.UseDefaultFiles();
    //Adding Static Files Middleware to serve the static files
    app.UseStaticFiles();
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Request afgehandeld met eind-middleware component");
    });
}
```

Run de app en je ziet dan nu MyCustomPage1.html als default in de browser wordt getoond:



16. Developer Exception Pagina Middleware in ASP.NET Core

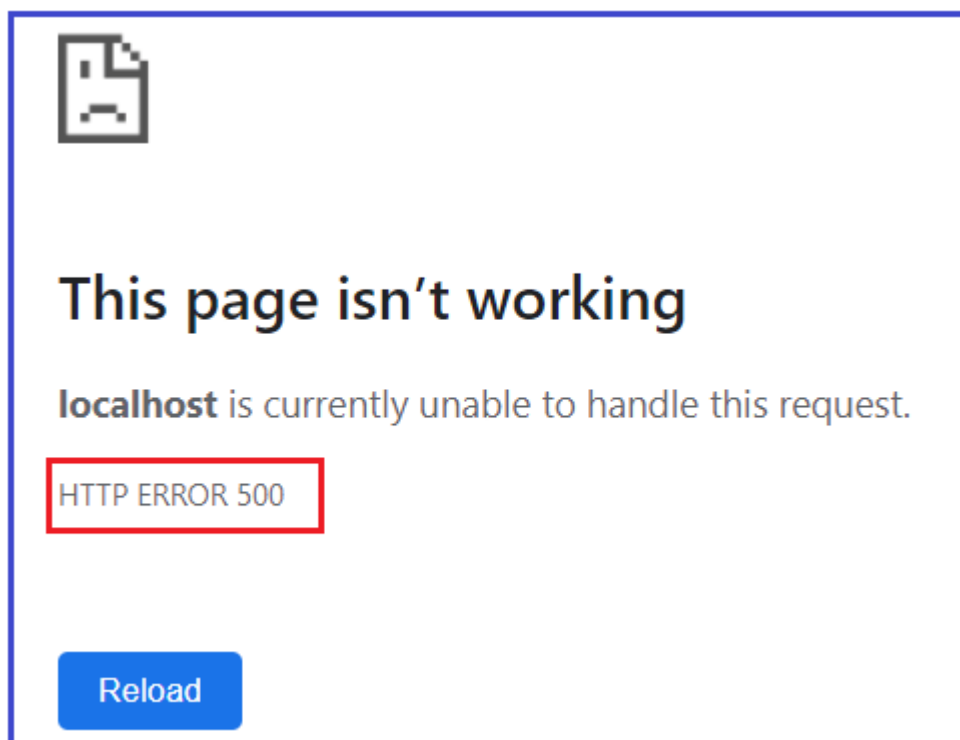
1. [Wat is de Developer Exception Page Middleware?](#)
2. [Hoe de Developer Exception Page Middleware gebruiken in een ASP.NET Core App?](#)
3. [Hoe de UseDeveloperExceptionPage Middleware customiseren in ASP.NET Core?](#)
4. [Waar moet de UseDeveloperExceptionPage Middleware worden geconfigureerd?](#)

Understanding Developer Exception Page Middleware in ASP.NET Core:

Maak of open de ASP.NET web app aangemaakt met de Empty Project template. Standaard geeft de ASP.NET Core application een status code terug bij een onafgehandelde exception. Pas de Configure() methode in Startup.cs als volgt aan :

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            throw new Exception("Fout bij het afhandelen van request");
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

Wanneer je de app runt, krijg je het volgende in een browser te zien :



Je krijgt de status code 500 te zien, wat betekent "Internal Server Error".

Als developer zou je graag de details willen weten van deze foutmelding, zodat je de fout kan verbeteren.

Hiervoor kan je de **DeveloperExceptionPage** Middleware component gebruiken.

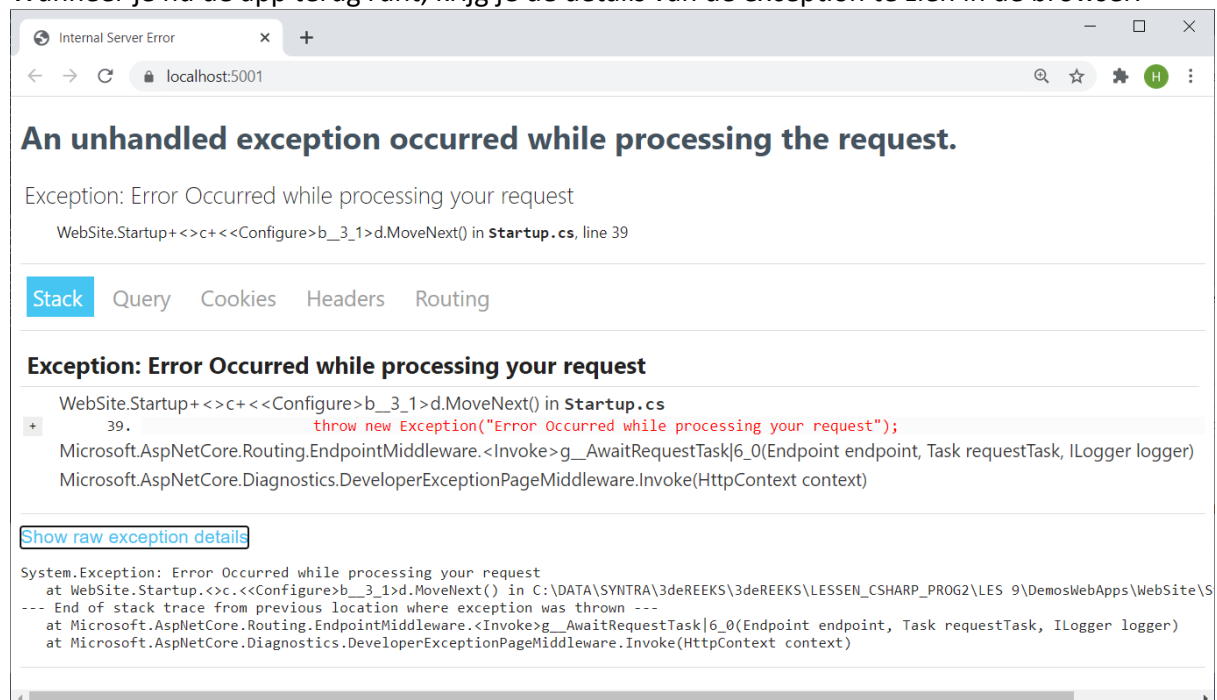
Hoe de DeveloperExceptionPage Middleware gebruiken?

Indien je wil dat je applicatie een pagina toont met de details van de unhandled exception, moet je de Developer Exception Page middleware in de request processing pipeline configureren.

Pas de Configure() methode van de Startup class aan om de Developer Exception Page middleware component toe te voegen die de unhandled exception zal verwerken en tonen in een pagina:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            throw new Exception("Error Occurred while processing your request");
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

Wanneer je nu de app terug runt, krijg je de details van de exception te zien in de browser:



De Developer Exception Page bevat 5 tabs: Stack, Queue, Cookies, Headers en Routing.

1. **Stack:** de Stack tab geeft de informatie van de stack trace waar de exception zich voordeed, de bestandsnaam en het lijn nummer van de code.
2. **Query:** de Query tab geeft informatie over de query strings.
3. **Cookies:** de Cookies tab toont de informatie over de cookies van de request.
4. **Header:** de Header tab geeft informatie over de request-headers die gestuurd zijn door de client (bv browser van surfer).
5. **Route:** de Route tab geeft informatie over het Route Patroon en de Route HTTP Verb (GET, POST,...) ,...

Op de cookies en querystring tabs zie je momenteel geen gegevens, Aangezien de request geen cookies of querystring gebruikt.

Opmerking: Zorg dat de Developer Exception Page Middleware enkel in development-mode wordt ingeschakeld. Dit kan je doen door dit conditioneel toe te voegen:

```
if (env.IsDevelopment())
```

Hoe een Middleware component customiseren in ASP.NET Core?

Customisatie is mogelijk door:

1. **UseDeveloperExceptionPage** => customiseren dmv **DeveloperExceptionPageOptions** object
2. **UseDefaultFiles** => customiseren via **DefaultFilesOptions** object
3. **UseStaticFiles** => customiseren via **StaticFileOptions** object
4. **UseFileServer** => customiseren via **FileServerOptions** object

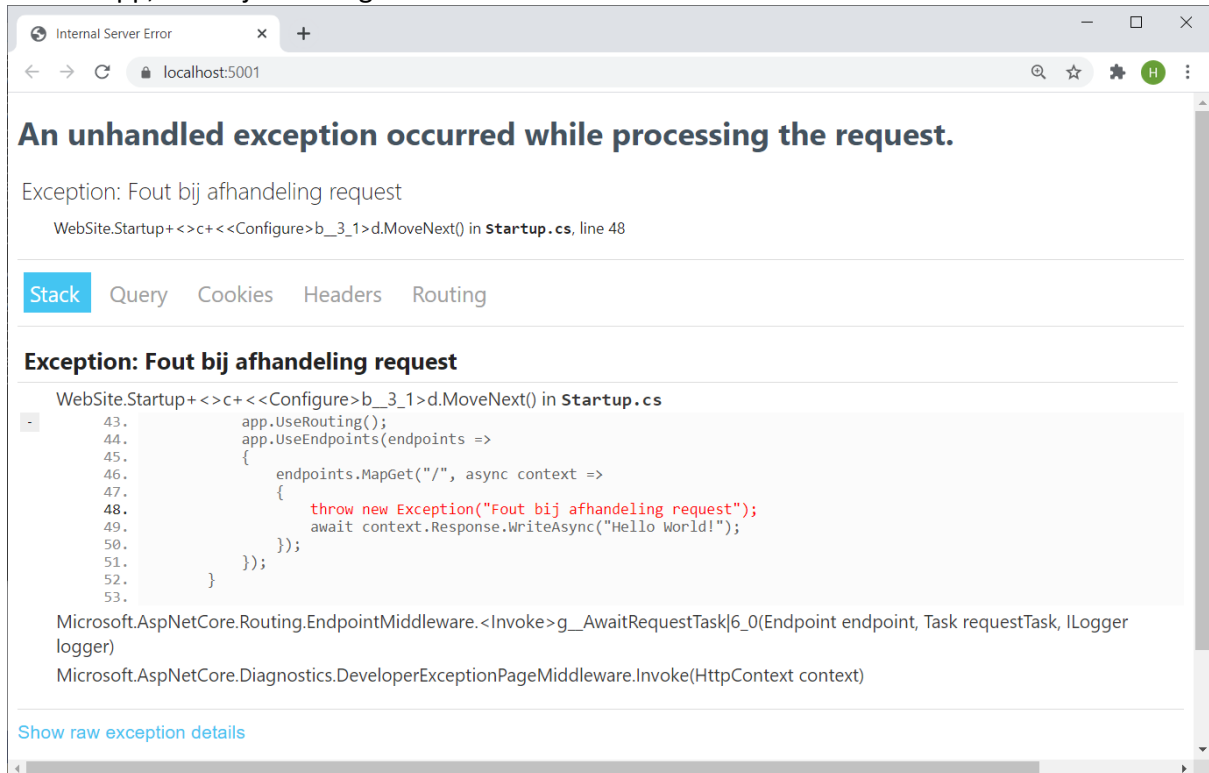
Voorbeeld van customisatie van de UseDeveloperExceptionPage() middleware:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        DeveloperExceptionPageOptions developerExceptionPageOptions = new
DeveloperExceptionPageOptions
        {
            SourceCodeLineCount = 5
        };
        app.UseDeveloperExceptionPage(developerExceptionPageOptions);
    }

    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            throw new Exception("Fout bij afhandeling request");
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

We gebruiken een property **SourceCodeLineCount** van de class **DeveloperExceptionPageOptions**. Deze specificeert het aantal lijnen code die moeten worden getoond vóór en na de lijn waar de exception zich voordeed.

Run de app, nu zie je het volgende in de browser:



Waar moet de UseDeveloperExceptionPage Middleware worden geconfigureerd?

Configureer de UseDeveloperExceptionPage() Middleware **zo vroeg mogelijk in de request processing pipeline**. Dus als eerste component in de Configure() methode van de Startup class

17. De .NET Core Command Line Interface

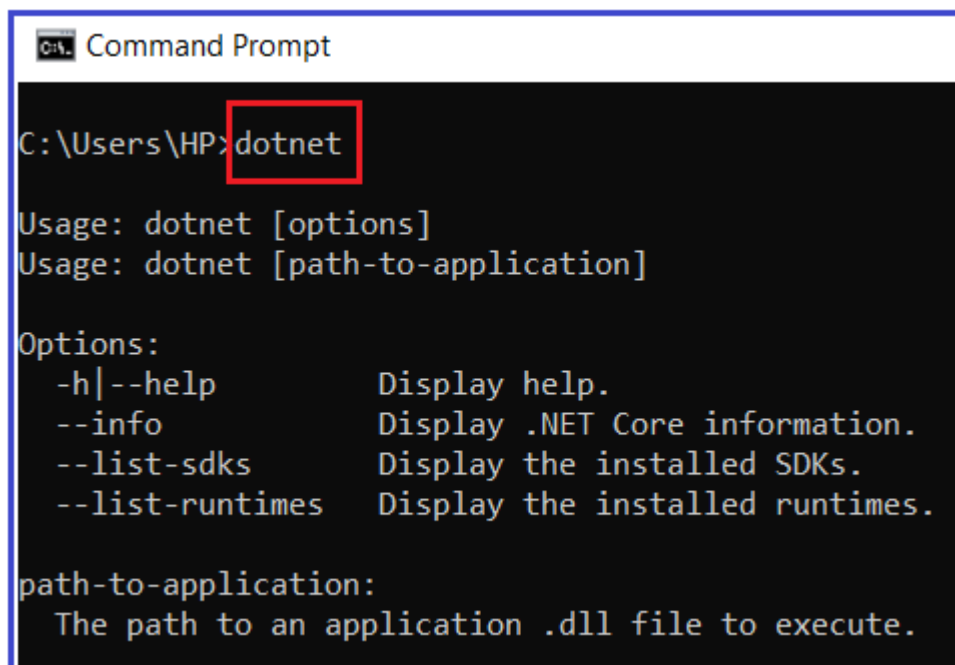
De .NET Core Command Line Interface:

De .NET Core CLI (Command Line Interface) is een cross-platform tool waarmee je .Net Core apps kan creëren, compileren, runnen, publiceren, NuGet packages installeren, restoren,... De .NET Core CLI command gebruikt voor web application het Out of Process hosting model, dwz het gebruikt de Kestrel web server om de app te runnen.

Tot nu toe hebben we steeds de IDE (Integrated Development Environment) Visual Studio 2019 gebruikt om projecten te creëren, te compileren en te runnen.

Bij installatie van de .NET Core SDK (Software Development Kit), wordt eveneens standaard de .NET Core CLI geïnstalleerd.

Om te verifiëren of de .NET CLI geïnstalleerd is op je locale machine, kan je in een command prompt (Windows) of terminal (Linux) "**dotnet**" enter typen. Als deze geïnstalleerd is, krijg je de help tekst te zien:



```
Command Prompt
C:\Users\HP>dotnet

Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help           Display help.
  --info              Display .NET Core information.
  --list-sdks         Display the installed SDKs.
  --list-runtimes     Display the installed runtimes.

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

.NET Core CLI Command Structure:

De .NET Core CLI command structuur is als volgt:

dotnet <commando> <argument> <option>

Opmerking: Alle .NET Core CLI commands beginnen met de driver **dotnet**. De driver start de uitvoering van het gespecificeerde commando.

Na de tekst dotnet moet het **commando** worden gespecificeerd om een bepaalde actie uit te voeren. Het commando kan gevolgd worden door één of meerder **arguments** en/of **options**.

Hoe een overzicht krijgen van alle .NET Core Commando's:

Open de command prompt en typ **dotnet help** en druk enter.

Alle mogelijke commando's worden dan getoond:

1. **add**: Voeg package of reference toe aan een .NET project.
2. **build**: Build(compileer) een .NET project.
3. **build-server**: Interageer met servers gestart met een build.
4. **clean**: Clean build outputs van een .NET project.
5. **help**: Toon de command-line help.
6. **list**: Toon lijst van project references van een .NET project.
7. **msbuild**: Run Microsoft Build Engine (MSBuild) commando's.
8. **new**: Creëer een nieuwe .NET project of file.
9. **nuget**: Levert additionele NuGet commando's.
10. **pack**: Creëer een NuGet package.
11. **publish**: Publiceer een .NET project voor deployment.
12. **remove**: Verwijder een package or reference from a .NET project.
13. **restore**: Herstel dependencies van een .NET project.
14. **run**: Build en run een .NET project output.
15. **sln**: Wijzig een Visual Studio solution file.
16. **store**: Bewaar de gespecificeerde assemblies in de runtime package store.
17. **test**: Run unit tests dmv de test runner van een .NET project.
18. **tool**: Installeer of beheer tools die .NET uitbreiden.
19. **vstest**: Run Microsoft Test Engine (VSTest) commando's.

Project Modificatie Commando's:

1. **add package**: Voeg package referentie toe aan project.
2. **add reference**: Voeg project-to-project (P2P) referentie toe.
3. **remove package**: Verwijder package referentie van project.
4. **remove reference**: Verwijder project referentie.
5. **list reference**: Geef lijst van alle project-to-project referenties.

Geavanceerde Commando's:

1. **nuget delete**: verwijder of un-list een package van de server.
2. **nuget locals**: Verwijder of geef lijst van alle NuGet resources.
3. **nuget push**: Plaats een package op de server en publiceer.
4. **msbuild**: Build (compileer) project en al zijn dependencies.
5. **dotnet install script**: Script om .NET Core CLI tools en shared runtime te installeren.

Creëer een nieuw project via een .NET Core CLI Commando:

Om een nieuw project te creëren, compileren en runnen via de CLI tool

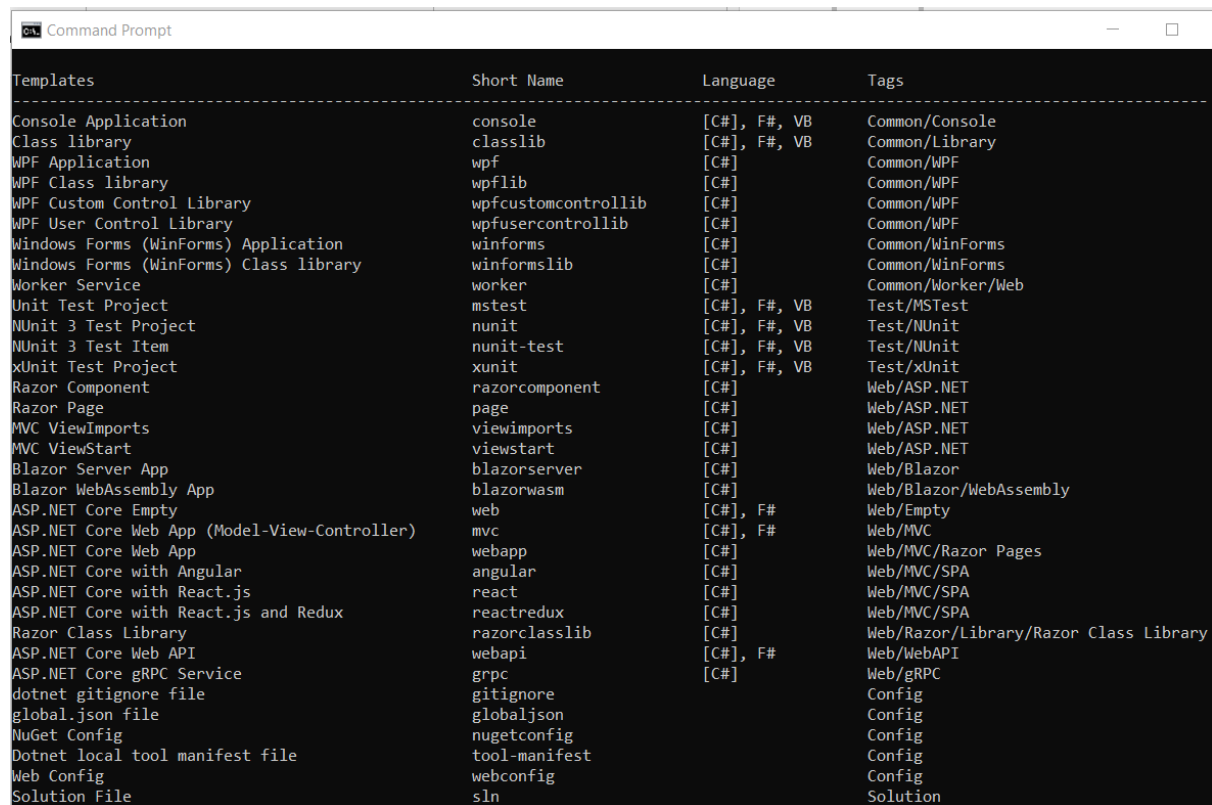
Om een nieuw .NET Core project aan te maken, moeten we het 'new' commando gebruiken gevolgd door de template naam als argument.

Hiermee kan je een console, class library, web, webapp, mvc, webapi, razor, angular, react, ... project aanmaken via CLI:

dotnet new <TEMPLATE>

Om de lijst van mogelijke templates te zien, typ :

dotnet new -l

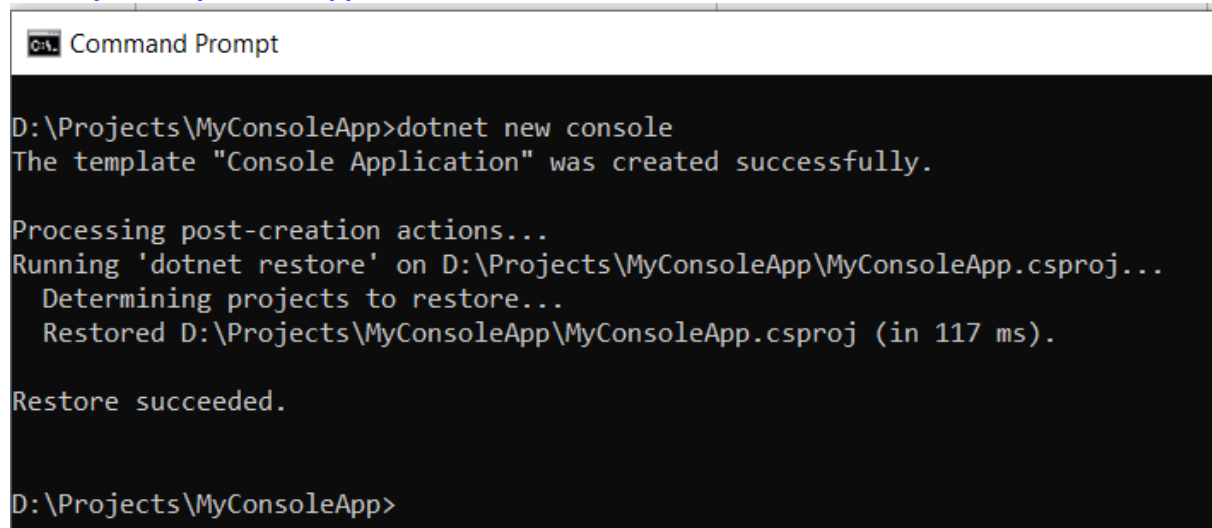


Templates	Short Name	Language	Tags
Console Application	console	[C#], F#, VB	Common/Console
Class library	classlib	[C#], F#, VB	Common/Library
WPF Application	wpf	[C#]	Common/WPF
WPF Class library	wplib	[C#]	Common/WPF
WPF Custom Control Library	wpfcustomcontrollib	[C#]	Common/WPF
WPF User Control Library	wpfusercontrollib	[C#]	Common/WPF
Windows Forms (WinForms) Application	winforms	[C#]	Common/WinForms
Windows Forms (WinForms) Class library	winformslib	[C#]	Common/WinForms
Worker Service	worker	[C#]	Common/Worker/Web
Unit Test Project	mstest	[C#], F#, VB	Test/MSTest
NUnit 3 Test Project	nunit	[C#], F#, VB	Test/NUnit
NUnit 3 Test Item	nunit-test	[C#], F#, VB	Test/NUnit
xUnit Test Project	xunit	[C#], F#, VB	Test/xUnit
Razor Component	razorcomponent	[C#]	Web/ASP.NET
Razor Page	page	[C#]	Web/ASP.NET
MVC ViewImports	viewimports	[C#]	Web/ASP.NET
MVC ViewStart	viewstart	[C#]	Web/ASP.NET
Blazor Server App	blazorserver	[C#]	Web/Blazor
Blazor WebAssembly App	blazorwasm	[C#]	Web/Blazor/WebAssembly
ASP.NET Core Empty	web	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/MVC
ASP.NET Core Web App	webapp	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA
ASP.NET Core with React.js and Redux	reactredux	[C#]	Web/MVC/SPA
Razor Class Library	razorclasslib	[C#]	Web/Razor/Library/Razor Class Library
ASP.NET Core Web API	webapi	[C#], F#	Web/WebAPI
ASP.NET Core gRPC Service	grpc	[C#]	Web/gRPC
dotnet gitignore file	gitignore		Config
global.json file	globaljson		Config
NuGet Config	nugetconfig		Config
Dotnet local tool manifest file	tool-manifest		Config
Web Config	webconfig		Config
Solution File	sln		Solution

Voorbeeld: Aanmaken van Console Application via .NET Core CLI

Ga in de command prompt naar de directory waar je je applicatie wil aanmaken , bv :

D:\Projects\MyConsoleApp>dotnet new console



```
D:\Projects\MyConsoleApp>dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on D:\Projects\MyConsoleApp\MyConsoleApp.csproj...
  Determining projects to restore...
  Restored D:\Projects\MyConsoleApp\MyConsoleApp.csproj (in 117 ms).

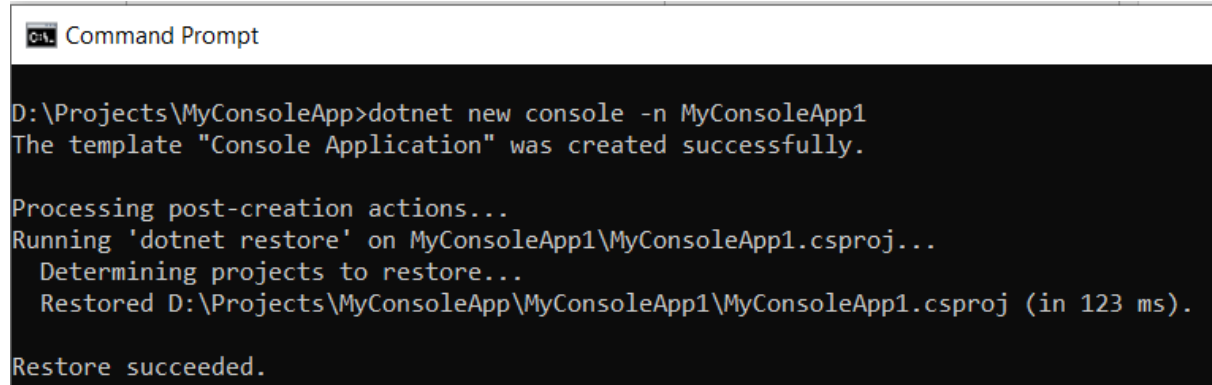
Restore succeeded.

D:\Projects\MyConsoleApp>
```

Om de naam van de Console app ‘MyConsoleApp1’ te speciëren, gebruikt **-n** or **–name** option bv.

D:\DotNetCoreApps>dotnet new console -n MyConsoleApp1

Nu wordt er een console app met naam MyconsoleApp1 aangemaakt:



```
C:\> Command Prompt

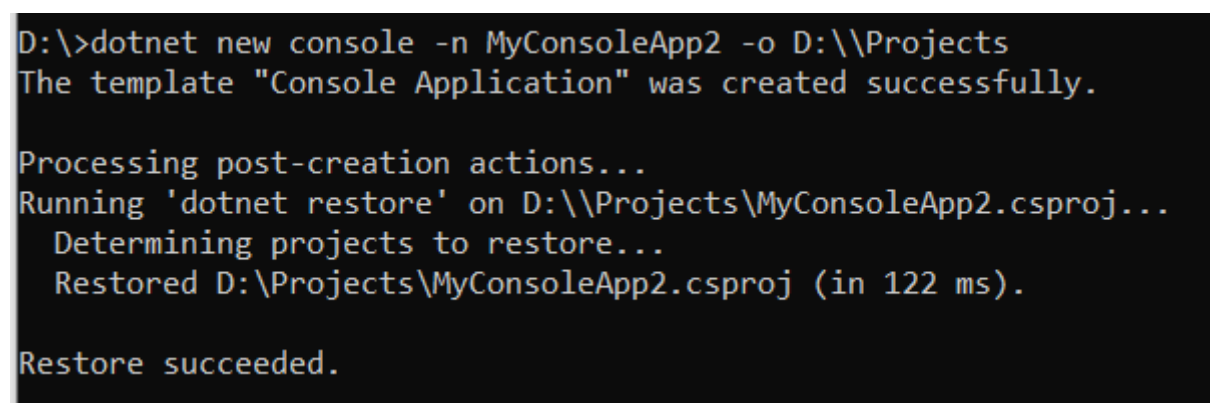
D:\Projects\MyConsoleApp>dotnet new console -n MyConsoleApp1
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on MyConsoleApp1\MyConsoleApp1.csproj...
  Determining projects to restore...
  Restored D:\Projects\MyConsoleApp\MyConsoleApp1\MyConsoleApp1.csproj (in 123 ms).

Restore succeeded.
```

Het volgende commando creëert een nieuwe console applicatie met naam MyConsoleApp2 in de \Projects directory. De **-o** or **-output** optie wordt gebruik om een output directory te specificeren waar het project moet komen.

dotnet new console -n MyConsoleApp2 -o D:\\Projects

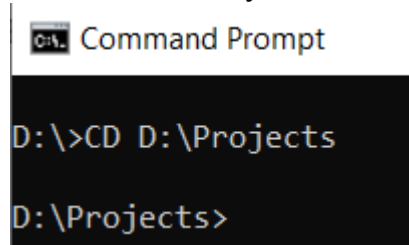


```
D:\>dotnet new console -n MyConsoleApp2 -o D:\\Projects
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on D:\\Projects\MyConsoleApp2.csproj...
  Determining projects to restore...
  Restored D:\Projects\MyConsoleApp2.csproj (in 122 ms).

Restore succeeded.
```

Ga naar de **D=>Projects** folder in de command prompt:



```
C:\> Command Prompt

D:\>CD D:\Projects

D:\Projects>
```

Controleer of de project files zijn aangemaakt met dir commando

Een NuGet package toevoege via .NET Core CLI:

Om de Package **Newtonsoft.json** te installeren, voer het volgende commando uit :

dotnet add package Newtonsoft.json

```
D:\Projects>dotnet add package Newtonsoft.json
Determining projects to restore...
Writing C:\Users\HP\AppData\Local\Temp\tmp6784.tmp
info : Adding PackageReference for package 'Newtonsoft.json' into project 'D:\Projects\MyConsoleApp2.csproj'.
info : Restoring packages for D:\Projects\MyConsoleApp2.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json 1146ms
info : Package 'Newtonsoft.json' is compatible with all the specified frameworks in project 'D:\Projects\MyConsoleApp2.csproj'.
info : PackageReference for package 'Newtonsoft.json' version '12.0.3' added to file 'D:\Projects\MyConsoleApp2.csproj'.
info : Committing restore...
info : Writing assets file to disk. Path: D:\Projects\obj\project.assets.json
log  : Restored D:\Projects\MyConsoleApp2.csproj (in 4.07 sec).
```

Open de .csproj file(project file) en controleer dat de NuGet Package referentie naar Newtonsoft.json is toegevoegd:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.json" Version="12.0.3" />
  </ItemGroup>

</Project>
```

Verwijderen van Nuget Package Referentie via .NET Core CLI:

De “**dotnet remove package**” commando verwijdert een NuGet package referentie van een project.

dotnet remove package Newtonsoft.json

Om de NuGet Package referentie te verwijderen, over het volgende commando uit:

“dotnet remove package Newtonsoft.json”

```
D:\Projects>dotnet remove package Newtonsoft.json
info : Removing PackageReference for package 'Newtonsoft.json' from project 'D:\Projects\MyConsoleApp2.csproj'.
```

Restore Packages via .NET Core CLI:

Om bestaande packages te herstellen of te updaten van een project, over het commando “**dotnet restore**” uit:

```
D:\Projects>dotnet restore
Determining projects to restore...
Restored D:\Projects\MyConsoleApp2.csproj (in 125 ms).
```

Build Project met .NET Core CLI:

Via het commando “**dotnet build**” kan je een .NET Core Project compileren:

```
D:\Projects>dotnet build
Microsoft (R) Build Engine version 16.7.0+b89cb5fde for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

Determining projects to restore...
All projects are up-to-date for restore.
MyConsoleApp2 -> D:\Projects\bin\Debug\netcoreapp3.1\MyConsoleApp2.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.97
```

Run .NET Core Project using .NET Core CLI Command:

Via het commando “**dotnet run**” kan je een project runnen:

```
D:\Projects>dotnet run
Hello World!
```


18. Referenties

<https://dotnettutorials.net/lesson/asp-net-core-environment-setup/>

<https://www.tutorialsteacher.com/core>

<https://www.yogihosting.com/category/ef-core/>

<https://dotnettutorials.net/course/asp-net-core-tutorials/>

<https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/>