KAPITEL 9

Einführung in ASP.NET Core

Dieses Kapitel bietet eine Einführung in ASP.NET Core. Die weiteren Kapitel in diesem Buchteil beschäftigen sich mit den Themen ASP.NET Core MVC, ASP.NET Core Web API und ASP.NET Core SignalR.

Klassisches ASP.NET oder ASP.NET Core?

In diesem Buch werden sowohl das klassische ASP.NET als auch das neue ASP.NET Core vorgestellt. Es stellt sich die Frage, was Sie einsetzen sollten.

ASP.NET Core bietet als Vorteile:

- Höhere Performance als das klassische ASP.NET.
- Plattformunabhängigkeit (bei Betrieb auf .NET Core).
- Einige schöne neue Funktionen. Insbesondere die Tag Helper und die View Components sind dazu geeignet, Funktionen und Seitenteile besser zu kapseln und damit die Übersichtlichkeit und Wiederverwendbarkeit zu erhöhen.
- Eine größere Zukunftssicherheit, da Microsoft das klassische ASP.NET nicht mehr wesentlich weiterentwickeln wird.

Allerdings hat ASP.NET Core auch Nachteile:

- Es gibt noch nicht so viele Drittanbieterkomponenten dafür wie für das klassische ASP.NET
- Die Dokumentation ist noch sehr unzureichend.
- Es ist nur der Betrieb auf .NET Core möglich: Es fehlen unter anderem einige Klassen, die es im klassischen .NET Framework, aber nicht in .NET Core gibt.

Für neue Projekte ist ASP.NET Core daher auf jeden Fall in Betracht zu ziehen.

Die Migration bestehender klassischer ASP.NET-Projekte sollte wohlüberlegt sein, da dies einen größeren Migrationsaufwand bedeutet. Die Migration von ASP.NET Webforms, ASP.NET Dynamic Data und ASP.NET AJAX kommt einer Neuprogrammierung gleich, da es diese Programmiermodelle in ASP.NET Core nicht mehr gibt. Aber auch die Umstellung von klassischem ASP.NET MVC auf ASP.NET Core MVC ist ein größerer Aufwand, da sich viele Programmierschnittstellen geändert haben.

Einführung in die Core-Welt

Nach über zwei Jahren der öffentlichen Entwicklung sind .NET Core 1.0 nebst ASP.NET Core 1.0 und Entity Framework Core 1.0 am 27. Juni 2016 erschienen. Aktuell zum Redaktionsschluss dieses Buchs gibt es im RTM-Status die Version 2.0 und als Vorschauversion die Version 2.1. Die Version 2.1 ist dann während der Satzphase dieses Buchs am 30.5.2018 erschienen.

Die Erstankündigung der Core-Produkte fand im Mai 2014 auf der TechEd-Konferent in den USA statt – damals noch unter den Schlagwörtern »Cloud-optimized .NET«, »Project K« und »ASP.NET vNext« sowie »Entity Framework 7«. Im November 2014 erfolgte dann die konkrete Benennung in .NET Core 5 und ASP.NET 5, die aber am 19. Januar 2016 auf .NET Core 1.0, ASP.NET Core 1.0 und Entity Framework Core 1.0 korrigiert wurde. Damit will Microsoft deutlich machen: Die Core-Produkte sind nicht die nächste Version des bisherigen .NET Framework 4.x beziehungsweise ASP.NET MVC 5.x und Entity Framework 6.x, sondern eine Neuentwicklung, die eine "Parallelwelt" zu den bisherigen .NET-Produkten bildet (siehe Abbildung 9-1).

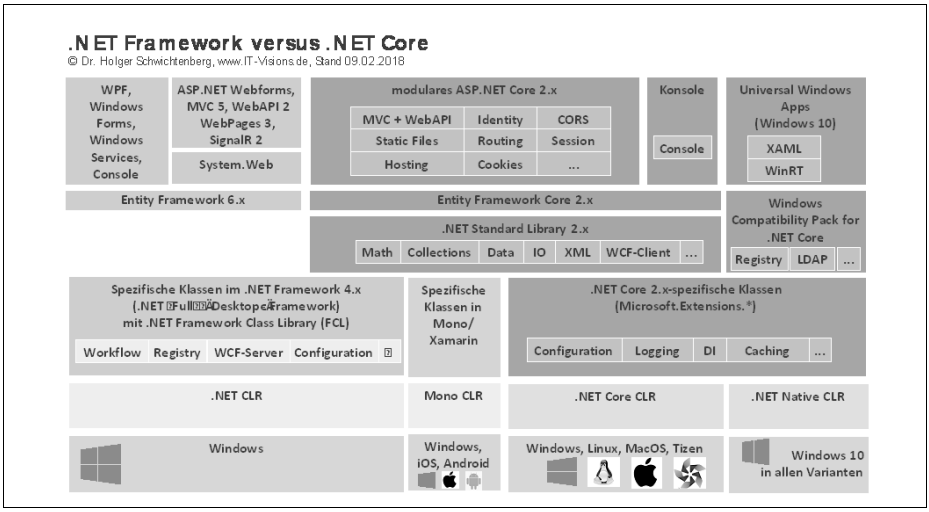


Abbildung 9-1: Die Core-Produkte sind eine »Parallelwelt« zu dem bisherigen .NET Framework.

Ein Teil von .NET Core 1.0 ist schon im Juli 2015 als Teil von Windows 10 erschienen: Auch die dortigen Windows Universal Apps nutzen mit der .NET Native CLR eine Variante der Core CLR und einiger Core-Bibliotheken.

Universal Windows Platform Apps in Windows 10 (alias: UWP Apps) sind derzeit die einzige Anwendungsart in der Core-Welt, mit der man eine grafische Benutzeroberfläche erstellen kann. Der Rest der Core-Produkte fokussiert auf Webanwendungen und REST-Webservices. Ein plattformübergreifendes GUI-Framework gibt es nicht und ist auch nicht in Sicht. Mit ASP.NET Core erstellt man das GUI – wie bisher – in HTML und JavaScript.

Die Core-Produkte sind grundsätzlich plattformunabhängig (was .NET eigentlich immer schon konnte, wofür es aber außer in Form von Mono und Xamarin nie eine Implementierung gab).

Neben Windows (inklusive dem kleinsten Windows, das es gibt, dem »Windows Nano Server«) läuft .NET Core auch auf macOS und den Linux-Varianten Ubuntu, Fedora, Debian, CentOS, Oracle Linux, OpenSUSE sowie RHEL. Auf der Website *https://www.microsoft.com/net/download* kann von .NET Core wahlweise die reine Laufzeitumgebung (Runtime) oder das Software Development Kit (SDK) mit kommandozeilenbasierten Werkzeugen bezogen werden.

Eine Besonderheit bei ASP.NET Core und Entity Framework Core ist, dass diese beiden Produkte nicht auf das .NET Core Framework angewiesen sind. Sie laufen auch unter dem klassischen .NET Framework mit dem vollen Funktionsumfang. Entity Framework läuft auch auf Mono und Xamarin.

.NET Core ist stark modularisiert in viele, zum Teil sehr kleine Nuget-Pakete. Wie stark Microsoft modularisiert hat, zeigt ein Blick auf *https://github.com/dotnet/corefx/tree/master/src*: Die Klassen des Namensraums System.IO, die bisher komplett in der mscorlib.dll steckten, sind nun auf 14 Assemblies (und Nuget-Pakete) aufgeteilt. Die Idee dahinter ist: Man sollte nur die Klassen in sein Projekt binden und zur Laufzeit in den Arbeitsspeicher RAM laden, die man wirklich braucht. Zum Beispiel: Die Klasse `System.IO.FileSystemWatcher` benutzen eher wenige Entwickler. Das zugehörige Nuget-Paket `System.IO.FileSystem.Watcher` besteht nur aus ganz wenigen Klassen: Neben der `FileSystemWatcher`-Klasse sind dies nur Klassen für die Ereignisparameter der Dateisystemüberwachung. Auch das Paket `System.IO.FileSystem.DriveInfo` besteht nur aus drei öffentlichen Klassen: `System.IO.DriveInfo`, `System.IO.DriveType` und `System.IO.DriveNotFoundException`. Dabei fällt außerdem auf, dass Paketname und Namensraum abweichen: Der Namensraum der IO-Klassen ist wie bisher `System.IO`. Bei den Paketnamen hat Microsoft noch `FileSystem` eingefügt.

.NET Core ist aber nicht so extrem modularisiert wie die Welt von *node.js*, wo es zum Teil Node-Packages-Manager-Pakete gibt, die aus nur wenigen Zeilen Programmcode bestehen.

.NET Standard

.NET Standard (ursprünglich auch .NET Platform Standard) ist eine Mitte 2016 eingeführte Spezifikation für den Funktionsumfang einer .NET-Klassenbibliothek, die von verschiedenen .NET-Varianten implementiert werden kann.

.NET Standard 2.0 (verabschiedet am 9.8.2017) wird realisiert von:

- .NET Framework ab Version 4.6.1
- .NET Core ab Version 2.0
- Mono ab Version 5.4

- Xamarin.iOS ab Version 10.14
- Xamarin.Mac ab Version 3.8
- Xamarin.Android ab Version 7.5
- Universal Windows Platform (UWP) ab Version 6.0 (Windows 10 Herbst 2017 Creators Update)

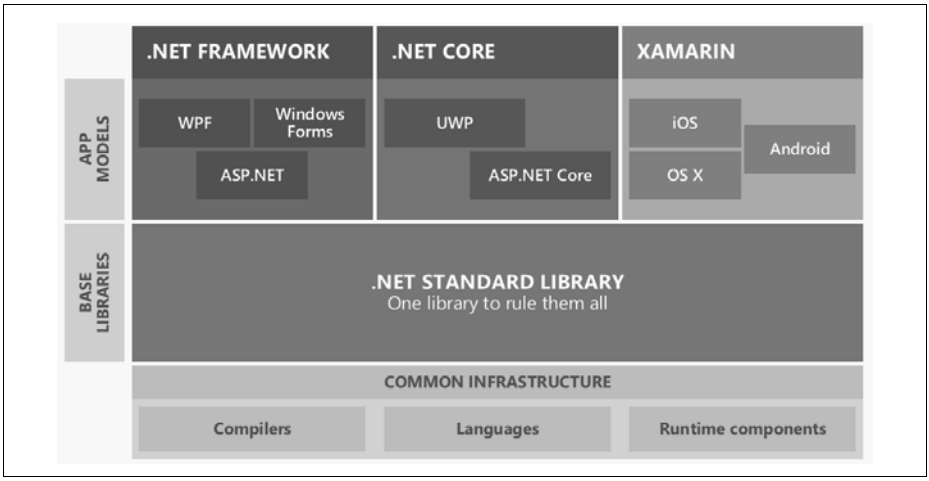


Abbildung 9-2: .NET Standard umfasst .NET-Basisklassen (Quelle des Bilds: Microsoft) für verschiedene .NET-Varianten.

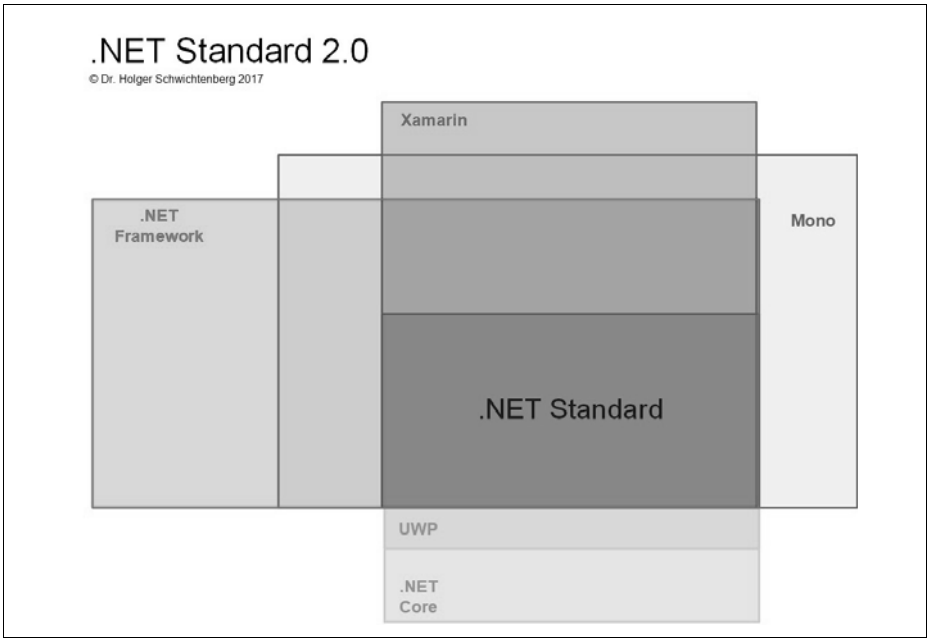


Abbildung 9-3: .NET Standard ist der gemeinsame Nenner verschiedener .NET-Implementierungen.

.NET Standard selbst ist kein installierbares Produkt, sondern nur eine Spezifikation. Es gibt in Visual Studio trotzdem eine Projektvorlage *Class Library (.NET Standard)*. Diese erlaubt, eine Klassenbibliothek anzulegen, die deklariert, dass sie nur Klassen aus einer in den Projekteigenschaften einstellbaren Version des .NET Standards verwendet. .NET Standard 2.0 verewigt sich in der Projektdatei *.csproj* mit dem Tag `<TargetFramework>netstandard2.0</TargetFramework>`.

Die folgende Abbildung zeigt eine .NET-Standard-2.0-Klassenbibliothek mit Namen `ITV.AppUtil.NETStandard`, die von zahlreichen Projektarten (klassische .NET-Konsolenanwendung, .NET-Core-Konsolenanwendungen, WPF-Anwendung, UWP-App, Xamarin-Android-App) verwendet wird.

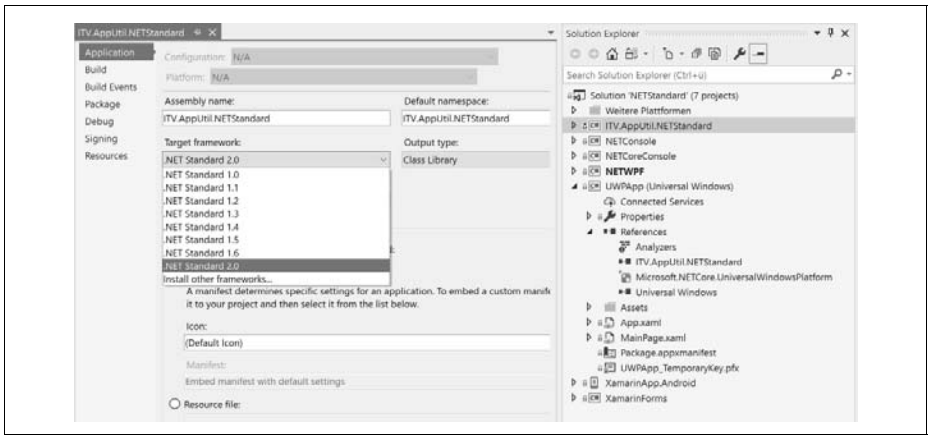


Abbildung 9-4: Eine .NET-Standard-Klassenbibliothek, die von mehreren .NET-Varianten und -Anwendungsarten verwendet wird

.NET Standard ist der Nachfolger des Konzepts der Portable Class Library (PCL), die in Visual Studio 2012 eingeführt wurden. Auch PCLs konnten in verschiedenen .NET-Varianten verwendet werden; die Schnittmenge war aber sehr klein.

Es gibt weiterhin spezifische Klassenbibliotheken für alle .NET-Varianten. Sie sollen aber bevorzugt .NET-Standard-Klassenbibliothek verwenden, wenn es für Ihren Anwendungsfall möglich ist, weil Sie die Klassen ohne Migration in anderen .NET-Varianten verwenden können. Beispiel: Sie starten eine Webanwendung mit ASP.NET Core auf .NET Full Framework. Verwenden Sie eine Klassenbibliothek in .NET Standard, können Sie diese Klassenbibliothek ohne Änderungen weiterverwenden, wenn Sie später ASP.NET Core auf .NET Core betreiben wollen.

Tabelle 9-1 zeigt die in .NET Standard enthaltenen Namensräume und die Anzahl der »Programmierschnittstellen«, wie Microsoft es nennt. Damit meint Microsoft nicht Klassen, sondern tatsächlich einzelne Klassenmitglieder. Die Namensräume entsprechend den Namensräumen aus dem klassischen .NET Framework.

Tabelle 9-1: Namensräume und Programmierschnittstellen in .NET Standard 2.0 (Quelle: Microsoft https://github.com/dotnet/standard/blob/master/docs/versions/netstandard2.0.md)

Namensraum	Programmierschnittstellen
Microsoft.Win32.SafeHandles	32
System	1,087
System.CodeDom.Compiler	14
System.Collections	292
System.Collections.Generic	17
System.Collections.ObjectModel	1
System.Collections.Specialized	241
System.ComponentModel	1,499
System.ComponentModel.Design	520
System.ComponentModel.Design.Serialization	150
System.Configuration.Assemblies	13
System.Data	1,399
System.Data.Common	701
System.Data.SqlTypes	882
System.Diagnostics	772
System.Diagnostics.CodeAnalysis	2
System.Diagnostics.Contracts	89
System.Diagnostics.Contracts.Internal	3
System.Diagnostics.SymbolStore	144
System.Diagnostics.Tracing	2
System.Drawing	681
System.Dynamic	173
System.Globalization	188
System.IO	275
System.IO.IsolatedStorage	104
System.IO.MemoryMappedFiles	64
System.IO.Pipes	124
System.Linq	351
System.Linq.Expressions	41
System.Net	1,271
System.Net.Cache	50
System.Net.Mail	279
System.Net.Mime	69
System.Net.NetworkInformation	692
System.Net.Security	147
System.Net.Sockets	164
System.Net.Web-Sockets	165

Tabelle 9-1: Namensräume und Programmierschnittstellen in .NET Standard 2.0 (Quelle: Microsoft https://github.com/dotnet/standard/blob/master/docs/versions/netstandard2.0.md) (Fortsetzung)

Namensraum	Programmierschnittstellen
System.Numerics	344
System.Reflection	348
System.Reflection.Emit	3
System.Resources	89
System.Runtime	12
System.Runtime.CompilerServices	164
System.Runtime.ConstrainedExecution	22
System.Runtime.ExceptionServices	6
System.Runtime.InteropServices	96
System.Runtime.InteropServices.ComTypes	15
System.Runtime.Remoting.Messaging	55
System.Runtime.Serialization	463
System.Runtime.Serialization.Formatters	13
System.Runtime.Serialization.Formatters.Binary	28
System.Runtime.Serialization.Json	64
System.Runtime.Versioning	33
System.Security	117
System.Security.Authentication	11
System.Security.Authentication.ExtendedProtection	40
System.Security.Claims	205
System.Security.Cryptography	684
System.Security.Cryptography.X509Certificates	67
System.Security.Permissions	87
System.Security.Principal	34
System.Text	56
System.Text.RegularExpressions	29
System.Threading	313
System.Threading.Tasks	66
System.Timers	36
System.Web	32
System.Xml	1,011
System.Xml.Linq	6
System.Xml.Resolvers	20
System.Xml.Schema	924
System.Xml.Serialization	935
System.Xml.XPath	244
System.Xml.Xsl	137

Nicht alle Klassen und alle Klassenmitglieder aus diesen Namensräumen aus dem klassischen .NET Framework sind Bestandteil von .NET Standard. Welche Klassen und Klassenmitglieder dies im Einzelnen sind, würde viele Hundert Buchseiten füllen. Dies können Sie nachsehen unter (*github.com/dotnet/standard/blob/master/docs/versions/netstandard2.0.md*. Grundsätzlich kann man aber festhalten: .NET Standard umfasst nur die nicht visuellen Basisklassen. Nicht in .NET Standard enthalten sind die Anwendungsframeworks und UI-Klassen von Windows Forms, Windows Presentation Foundation (WPF) und dem klassische ASP.NET. Ebenso nicht enthalten sind folgende Funktionen:

- Caching
- Windows Workflow Foundation (WF)
- WCF-Server
- System.ServiceProcess
- System.Media
- Registry (Microsoft.Win32)
- System.CodeDom
- LINQ-to-SQL
- WMI (System.Management)
- System.Configuration
- MSMQ (System.Messaging)

Windows Compatibility Pack für .NET Core

Das Windows Compatibility Pack for .NET Core umfasst zahlreiche Klassen aus dem klassischen .NET Framework (.NET "Full" Framework), die es dort zum Teil schon seit Version 1.0 gibt, die aber bisher nicht im .NET Standard 2.0 und damit auch nicht in .NET Core enthalten sind. Das hat bisher die Migration klassischen .NET-Framework-Programmcodes auf das neue .NET Core verhindert beziehungsweise erschwert. Trotz seines Namens laufen Teile dieser Bibliothek nicht nur auf Windows, sondern auch auf Linux und macOS.

Microsoft hat das WCP auf der .NET Conf im September angekündigt (*www.heise.de/developer/meldung/Microsoft-kuendigt-Windows-Compatibility-Pack-fuer-NET-Core-an-3835840.html*. Seit dem 30. Mai 2018 ist die erste Version unter der Nummer 2.0 Verfügbar. Dabei handelt es sich um ein Nuget-Metapaket mit Namen »Microsoft.Windows.Compatibility« (*www.nuget.org/packages/Microsoft.Windows.Compatibility*) (siehe Abbildung 9-59.5). Das neue Paket lässt sich in .NET-Core-2.0- und .NET-Standard-2.0-Projekten referenzieren.

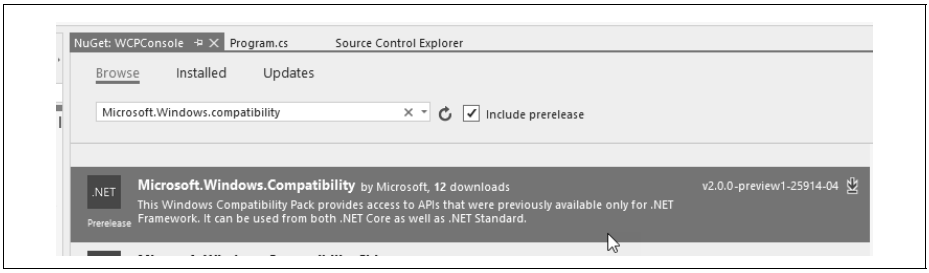


Abbildung 9-5: Einbindung des Windows Compatibility Pack for .NET Core via Nuget

Das Metapaket umfasst zahlreiche Einzelpakete (siehe Abbildung 9-6). Ein Entwickler kann wahlweise die benötigten Einzelpakete oder das Metapaket referenzieren, wobei im letzteren Fall die Deployment-Werkzeuge von .NET Core und Visual Studio dafür sorgen, dass nur die tatsächlich verwendeten Einzelpakete verbreitet werden.

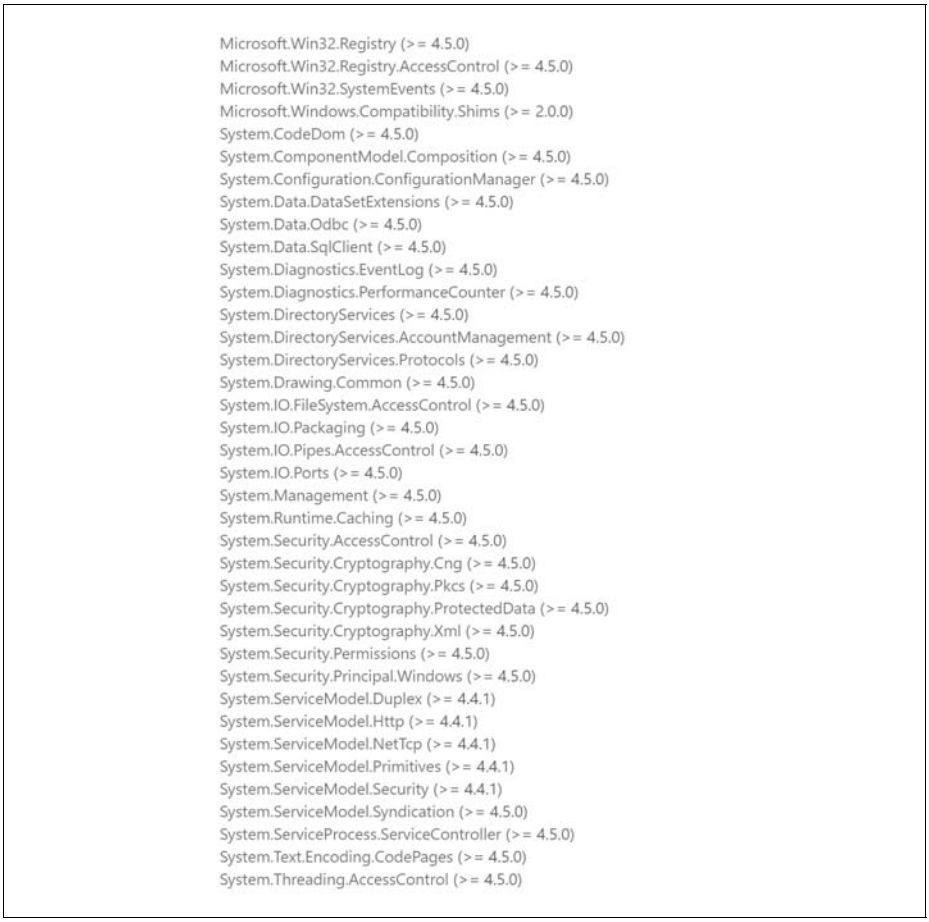


Abbildung 9-6: Inhalt des Metapakets

Zu den Funktionen des WCP gehören der Zugriff auf die Windows-Registry, das Code Document Object Model (CodeDOM), das Komprimieren von Dateien, das Lesen und Verändern von Dateiberechtigungen, die Bereitstellung von Webservices mit der Windows Communication Foundation (WCF) und die direkte Datenbankprogrammierung mit den Klassen `DataSet` und `DataReader` gegen einen Microsoft-SQL-Server.

Wie Abbildung 9-7 zeigt, plant Microsoft, weitere Klassen in das Bibliothekpaket aufzunehmen, zum Beispiel ODBC-Datenbankzugriffe, Grafikbearbeitung und Caching sowie die Windows Management Instrumentation (WMI) für den Zugriff auf Systeminformationen. Laut einem Blogeintrag von Microsoft umfasst das Windows Compatibility Pack 20.000 Funktionen (*blogs.msdn.microsoft.com/dotnet/2017/11/16/announcing-the-windows-compatibility-pack-for-net-core)*. Dies bezieht sich aber wohl auf die Endausbaustufe und nicht auf die aktuelle Preview-Version.

Component	Status	Windows-Only	Component	Status	Windows Only
Microsoft.Win32.Registry	Available	Yes	System.Management	Coming	Yes
Microsoft.Win32.Registry.AccessControl	Available	Yes	System.Runtime.Caching	Coming	
System.CodeDom	Available		System.Security.AccessControl	Available	Yes
System.ComponentModel.Composition	Coming		System.Security.Cryptography.Cng	Available	Yes
System.Configuration.ConfigurationManager	Available		System.Security.Cryptography.Pkcs	Available	Yes
System.Data.DatasetExtensions	Coming		System.Security.Cryptography.ProtectedData	Available	Yes
System.Data.Odbc	Coming		System.Security.Cryptography.Xml	Available	Yes
System.Data.SqlClient	Available		System.Security.Permissions	Available	
System.Diagnostics.EventLog	Coming	Yes	System.Security.Principal.Windows	Available	Yes
System.Diagnostics.PerformanceCounter	Coming	Yes	System.ServiceModel.Duplex	Available	
System.DirectoryServices	Coming	Yes	System.ServiceModel.Http	Available	
System.DirectoryServices.AccountManagement	Coming	Yes	System.ServiceModel.NetTcp	Available	
System.DirectoryServices.Protocols	Coming		System.ServiceModel.Primitives	Available	
System.Drawing	Coming		System.ServiceModel.Security	Available	
System.Drawing.Common	Available		System.ServiceModel.Syndication	Coming	
System.IO.FileSystem.AccessControl	Available	Yes	System.ServiceProcess.ServiceBase	Coming	Yes
System.IO.Packaging	Available		System.ServiceProcess.ServiceController	Available	Yes
System.IO.Pipes.AccessControl	Available	Yes	System.Text.Encoding.CodePages	Available	Yes
System.IO.Ports	Available	Yes	System.Threading.AccessControl	Available	Yes

Abbildung 9-7: Verfügbarkeit von Klassen im Windows Compatibility Pack for .NET Core

Anders als der Name Windows Compatibility Pack suggeriert, sind einige der Teilbibliotheken nicht nur auf Windows, sondern auch auf den anderen von .NET Core unterstützten Plattformen (Linux und macOS) verfügbar. Einige Funktionen – wie zum Beispiel der Windows-Registry-Zugriff – ergeben dort auch keinen Sinn.

Wenn ein Entwickler Teile des Windows Compatibility Pack verwendet, die nur für Windows implementiert sind, verliert er die Plattformunabhängigkeit. Die Anwendung lässt sich dann zwar noch auf macOS oder Linux starten, es kommt aber beim Zugriff auf diese APIs zum Laufzeitfehlertyp `PlatformNotSupportedException`. Entwickler können mit der Bedingung `if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows)) { … }` die Zugriffe auf diese APIs nur dann ausführen, wenn die Anwendung unter Windows läuft.

Das neue Windows Compatibility Pack vereinfacht insbesondere den Umstieg vom klassischen .NET Framework auf .NET Core für Entwickler, die zumindest vorerst weiterhin ihre Anwendung nur für Windows anbieten wollen. Der Einsatz des Windows Compatibility Pack kann auch ein erster Schritt in Richtung Linux und Mac sein. Der Entwickler kann auf diesen Betriebssystemen zunächst einen Teil der Funktionen ausblenden und dann schrittweise andere Implementierungen für diese Funktionen realisieren.

Open Source

Alle Softwareprodukte der Core-Familie sind Open Source. Es kommen die Lizenzen der MIT, der Apache Software Foundation beziehungsweise die Creative Commons 4.0 zum Einsatz. Die Entwicklung findet öffentlich auf GitHub statt, und Microsoft hat dabei auch signifikante Beiträge von fast 10.000 externen Entwicklern in die Software einfließen lassen (*blogs.msdn.microsoft.com/dotnet/2016/06/27/announcing-net-core-1-0*).

Links zu den GitHub-Repositories:

- .NET Core: *https://github.com/dotnet/core*
- .NET Core CLR: *https://github.com/dotnet/coreclr*
- .NET-Core-Basisklasse: *https://github.com/dotnet/corefx*
- ASP.NET Core: *https://github.com/aspnet/Home*
- Entity Framework Core: *https://github.com/aspnet/EntityFrameworkCore*

Dokumentation

Die .NET-Core-Dokumentation findet man nicht wie bisher im MSDN, sondern auf Microsofts neuer Dokumentationswebsite *docs.microsoft.com*. Hier findet man inzwischen auch die Dokumentation des klassischen .NET Framework, des klassischen ASP.NET und des klassischen Entity Framework.

	Englische Dokumentation	Deutsche Übersetzung
.NET Core	*https://docs.microsoft.com/en-us/dotnet/*	*https://docs.microsoft.com/de-de/dotnet/*
ASP.NET Core	*https://docs.microsoft.com/en-us/aspnet/*	*https://docs.microsoft.com/de-de/aspnet/*
Entity Framework Core	*https://docs.microsoft.com/en-us/ef/*	*https://docs.microsoft.com/de-de/ef/*
.NET-Klassenbibliothek	*https://docs.microsoft.com/en-us/dotnet/api/*	*https://docs.microsoft.com/de-de/dotnet/api/*

Die deutsche Übersetzung ist wegen der vielen Übersetzungsfehler nicht empfehlenswert!

Auch fast zwei Jahre nach dem Erscheinen der ersten Version der Core-Produkte ist die Dokumentation immer noch unzureichend:

- Die Dokumentation der Core-Produkte ist insgesamt deutlich knapper als die Dokumentation der klassischen .NET-Produkte.
- An einigen Stellen stehen nur Platzhalter »This topic hasn't been written yet«.

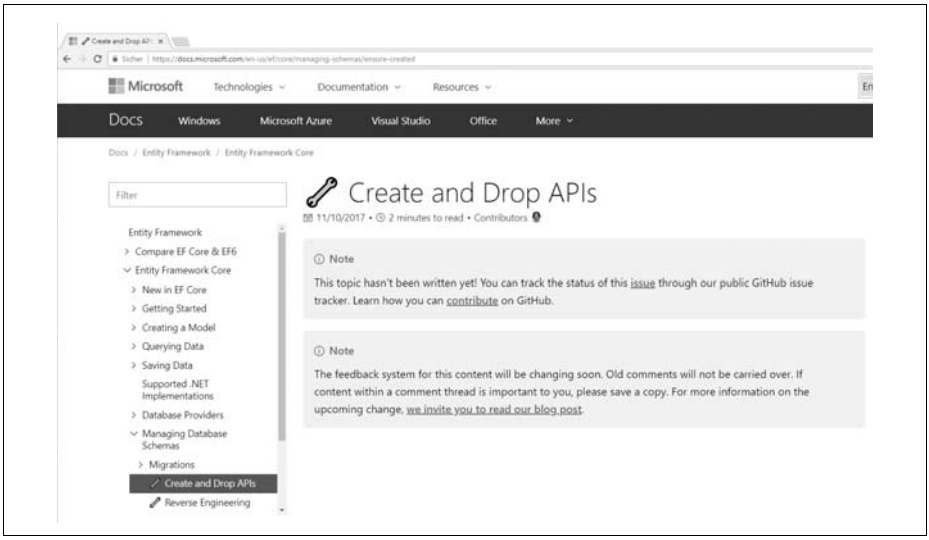


Abbildung 9-8: Screenshot aus der Dokumentation, erstellt am 12.3.2018. Seit 10.11.2017 hat sich an der Seite nichts mehr getan (docs.microsoft.com /ef/core/managing-schemas/ensure-created).

- Während es früher zu jedem einzelnen .NET-Klassenmitglied einen Beschreibungstext und in der Regel auch ein Beispiel gab, findet man zu den in .NET Core hinzugefügten Klassen weder Beispiele noch durchgehend Beschreibungstext.

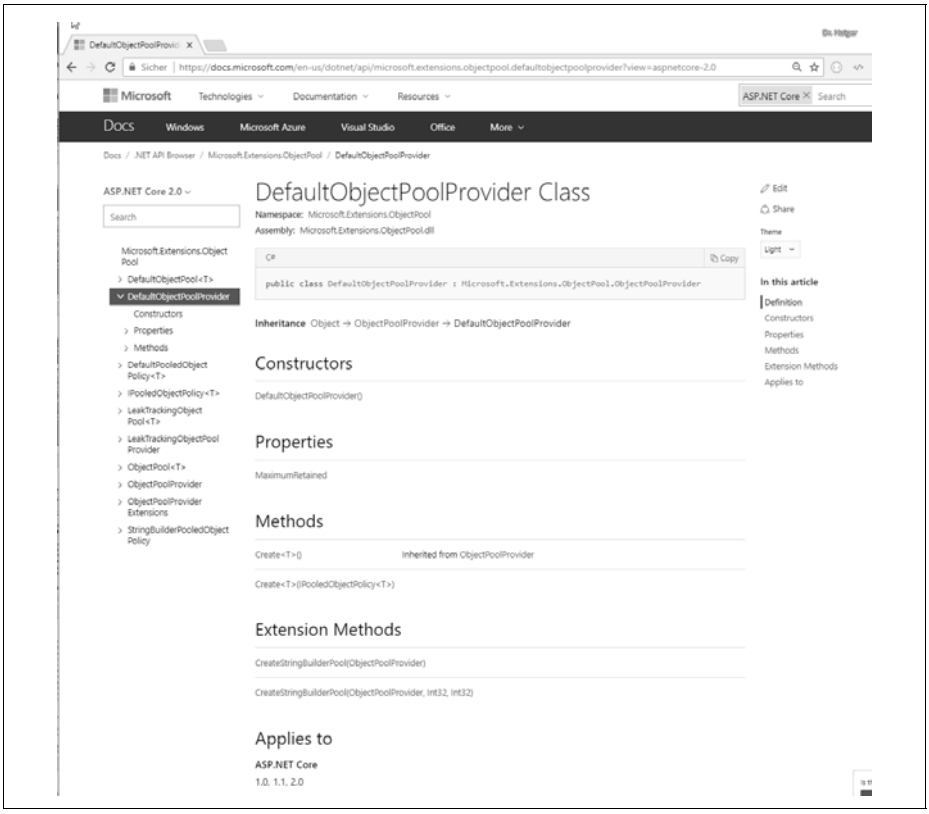


Abbildung 9-9: Es existieren keine Hilfetexte und keine Beispiele in der Dokumentation vieler neuer Klassen in .NET Core, ASP.NET Core und Entity Framework Core (hier am Beispiel der Klasse DefaultObjectPoolProvider*) (docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.objectpool.defaultobjectpoolprovider).*

- Die Dokumentationen für die Klassen sind oft nur aus dem Quellcode generiert. Und auch im Quellcode findet man selten (XML-)Kommentare.

```
// Copyright (c) .NET Foundation. All rights reserved.
// Licensed under the Apache License, Version 2.0. See License.txt in the project root for license information.

using System;
using System.Threading;

namespace Microsoft.Extensions.ObjectPool
{
    public class DefaultObjectPool<T> : ObjectPool<T> where T : class
    {
        private readonly T[] _items;
        private readonly IPooledObjectPolicy<T> _policy;

        public DefaultObjectPool(IPooledObjectPolicy<T> policy)
            : this(policy, Environment.ProcessorCount * 2)
        {
        }

        public DefaultObjectPool(IPooledObjectPolicy<T> policy, int maximumRetained)
        {
            if (policy == null)
            {
                throw new ArgumentNullException(nameof(policy));
            }

            _policy = policy;
            _items = new T[maximumRetained];
```

Abbildung 9-10: Quellcode der Klasse DefaultObjectPoolProvider *– ohne Kommentare (github.com /aspnet/Common/tree/dev/src))*

Sich in .NET Core einzuarbeiten, bedeutet daher an einigen Stellen, den Quellcode von Microsoft auf GitHub zu studieren. Manchmal hilft ein Blick in die von Microsoft geschriebenen Unit Tests, um die vorgesehene Verwendung eines APIs zu verstehen.

Werkzeuge

Microsoft legt bei .NET Core Wert darauf, dass alle wesentlichen Entwicklerfunktionen per Kommandozeilenwerkzeug unabhängig von einem bestimmten Editor ausgeführt werden können. Dazu gehört unter anderem das Anlegen von Projekten, das Laden des Nuget-Pakets, die Übersetzung, die Ausführung von Tests und das Erstellen eines Deployments.

Die Unabhängigkeit von einem bestimmten Editor ermöglicht dem Entwickler, den Editor frei zu wählen, beziehungsweise den Entwicklern von Editoren, .NET in den Editor zu integrieren.

Darüber hinaus gibt es aber auch komfortable Editoren für .NET-Core-Anwendungen.

Kommandozeilenwerkzeug dotnet

Microsoft stellt im .NET Core SDK das Kommandozeilenwerkzeug *dotnet* (unter Linux und macOS) beziehungsweise *dotnet.exe* (unter Windows) bereit. Der offizielle Name ist .NET Core CLI (CLI steht für Command Line Interface).

Das Werkzeug *dotnet* ist ein Über-Werkzeug (ein Rahmenwerkzeug) über zahlreiche weitere Werkzeuge; daher ist es erweiterbar. Das Wort nach *dotnet* drückt den eigentlichen Befehl aus. Sofern nicht durch einen Parameter anders angegeben, bezieht sich der *dotnet*-Befehl immer auf das aktuelle Dateisystemverzeichnis.

Beispiele für den Einsatz von dotnet

Liste der Projektvorlagen:

```
dotnet new
```

Neue Konsolenanwendung mit *.csproj*-Datei anlegen:

```
dotnet new console --framework netcoreapp2.1 --language C#
```

Neues MVC-Webprojekt mit *.csproj*-Datei anlegen:

```
dotnet new mvc --framework netcoreapp2.1 --language C#
```

Neues Razor-Pages-Webprojekt mit datenbankbasierter Benutzerverwaltung anlegen:

```
dotnet new razor --framework netcoreapp2.1 --language C# --auth individual
```

Nuget-Paketinstallation:

```
dotnet restore
```

Übersetzen:

```
dotnet build
```

Übersetzen und Starten:

```
dotnet run
```

Unit Tests starten:

```
dotnet test $PSScriptRoot\Test\UnitTests\UnitTests.csproj
```

Framework-Dependent Deployments (FDD)/Portable Applications (PA):

```
dotnet publish -c release --framework netcoreapp2.0 -o t:\PA
```

FDD-Paket ausführen:

```
dotnet t:\PA\SCADemo.dll
```

Self-Contained Deployment (SCD)/Self-contained application (SCA) für Windows 10, 64-Bit:

```
dotnet publish -c release --runtime win10-x64 --self-contained --framework
netcoreapp2.0 -o t:\SCA
```

SCD-Paket ausführen:

```
t:\SCA\SCADemo.exe
```

Das Kommandozeilenwerkzeug *dotnet* (*dotnet.exe*) ist der Nachfolger diverser Werkzeuge (*k*, *klr*, *kre*, *kvm*, *kpm*, *dnwm*, *dnx*, *dnu*), die Microsoft in der Beta-Phase sowie der .NET Core 1.0 verwendet hat.

Das Dokument *docs.microsoft.com/dotnet/core/tools/extensibility* beschreibt, wie man selbst Erweiterungen für das .NET-Core-CLI-Tool entwickeln kann.

Editoren

Empfehlenswerte Editoren mit Eingabeunterstützung und Debugger für .NET-Core-Anwendungen sind:

- Visual Studio ab Version 2017 (verwenden Sie nicht Version 2015 – diese nutzt noch das alte Projektformat!)
- Visual Studio Code
- Visual Studio for Mac (ehemals Xamarin Studio)
- JetBrains Rider (*www.jetbrains.com/rider*)

Nur Visual Studio Code (VSCode) ist gänzlich kostenfrei. Von Visual Studio kann man eine kostenfreie Community Edition nur in kleineren Firmen und bei Open-Source-Projekten kostenfrei einsetzen (*https://www.visualstudio.com/vs/community*).

Dank eines Microsoft-Projekts namens OmniSharp (*www.omnisharp.net*) stehen inzwischen auch viele Hilfsfunktionen für das Programmieren mit C# in anderen Editoren wie Atom, Brackets, Emacs, Sublime und Vim zur Verfügung.

Erste Schritte mit ASP.NET Core (auf .NET Core)

Dieses Kapitel geht davon aus, dass Sie Visual Studio 2017 verwenden.

Dieses Kapitel beschreibt die Verwendung von ASP.NET Core auf .NET Core. In Kapitel 10 werden die (geringen) Unterschiede zu ASP.NET Core auf dem klassischen .NET »Full« Framework behandeln.

Installation

Wichtig ist, dass Sie in Visual Studio 2017 nicht nur den Workload *.NET Core Cross-Platform Development* wählen, sondern zusätzlich das .NET Core SDK in der aktuellen Version von (*www.microsoft.com/net/download/windows*) installieren.

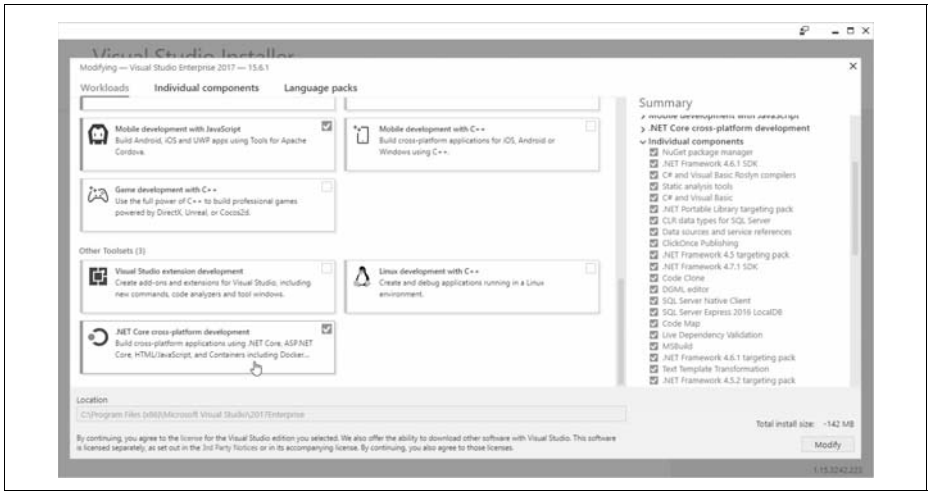


Abbildung 9-11: Installation des Workloads .NET Core Cross-Platform Development *in Visual Studio 2017*

Die Laufzeitumgebung und das SDK finden sich nach der Installation im Dateisystem in Windows unter *C:\Program Files\dotnet* mit dem zentralen Kommandozeilenwerkzeug *dotnet.exe*. Für den Betrieb von .NET-Core-Anwendungen wird dann nur die .NET-Core-Runtime, die im SDK enthalten ist, und für ASP.NET-Core-Webanwendungen zusätzlich das Paket *.NET Core Windows Server Hosting Bundle* (*aka.ms/dotnetcore-2-windowshosting*) benötigt, das die Verbindung zwischen den Internet Information Services (IIS) und ASP.NET Core herstellt.

Projekt anlegen

An der Konsole legt man ein neues Projekt an mit dem Kommandozeilenbefehl

```
dotnet new -t [Vorlage].
```

Beispiel: Anlegen eines neuen ASP.NET-MVC-Webprojekts:

```
dotnet new -t mvc
```

Die Liste der verfügbaren Vorlagen sieht man, wenn man nur `dotnet new` eingibt (siehe Abbildung 9-12).

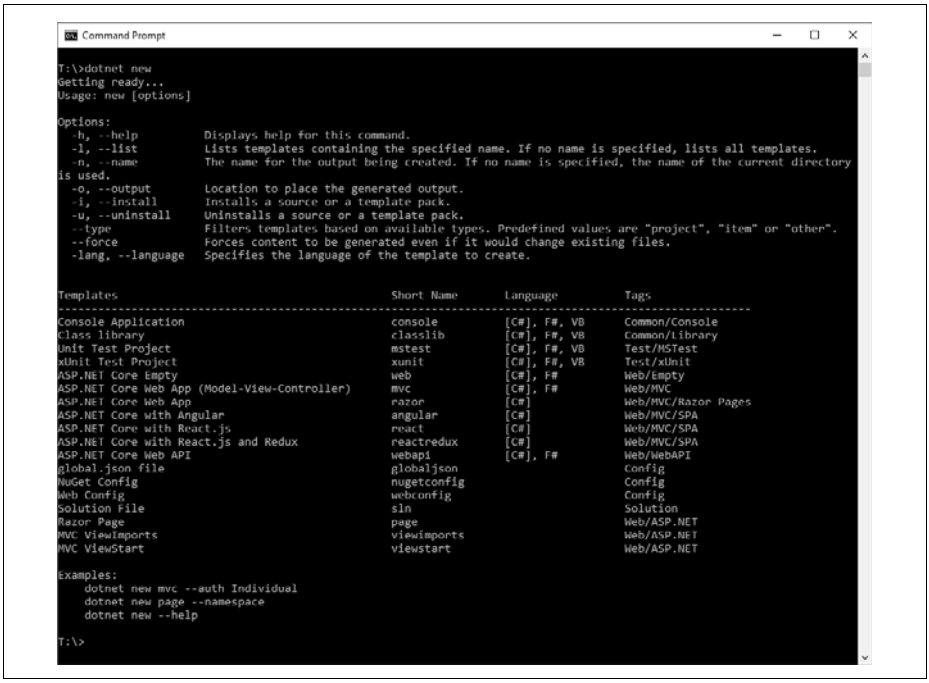


Abbildung 9-12: Projektvorlagen im .NET Core 2.0 SDK

Weitere Vorlagen für *dotnet new* findet man hier: *github.com/dotnet/templating/wiki/Available-templates-for-dotnet-new*. Auch der in der Web-Welt weit verbreitete Code-Generator Yeoman (*yeoman.io*) stellt Projektvorlagen für ASP.NET Core bereit.

Die gleichen Vorlagen, die *dotnet new* im Standard zeigt, findet man auch in Visual Studio 2017, wobei man die verschiedenen Vorlagen für ASP.NET in einem Unterdialog von ASP.NET Core Web Application findet (siehe Abbildung 9-13 und Abbildung 9-14).

Zu dem nachstehenden Dialog gelangt man auf zwei Wegen:

- *Add New Project/Visual C#/Web/ASP.NET Core Web Application*
- *Add New Project/Visual C#/.NET Core/ASP.NET Core Web Application*

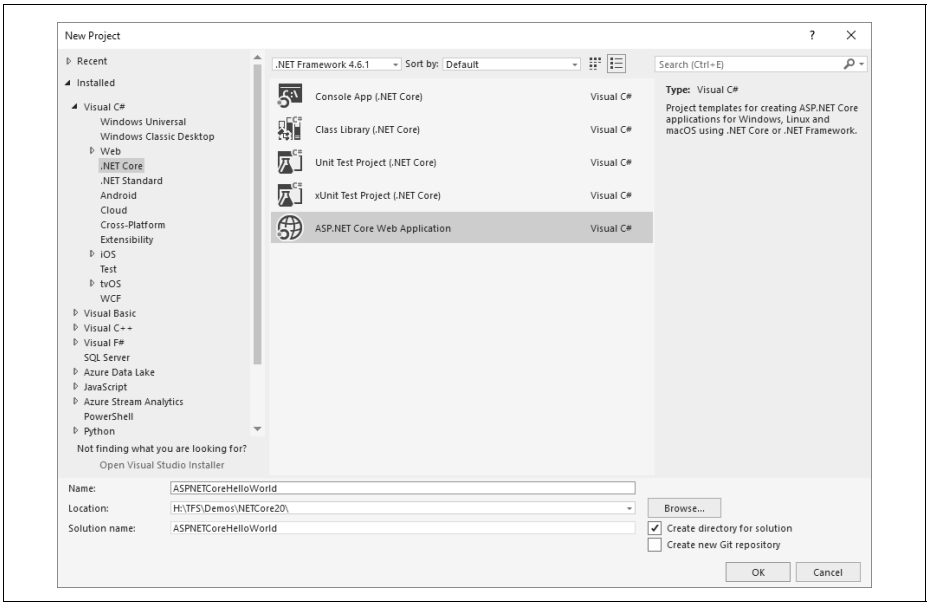


Abbildung 9-13: .NET-Core-Projektvorlagen in Visual Studio 2017

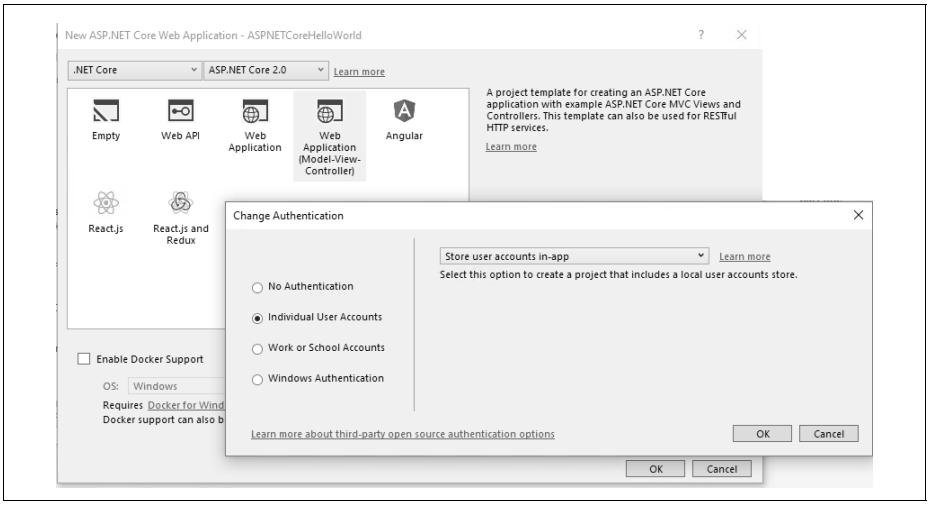


Abbildung 9-14: ASP.NET-Core-Projektvorlagen in Visual Studio 2017

Weitere Vorlagen für Visual Studio findet man im ASP.NET Core Template Pack (*marketplace.visualstudio.com/items?itemName=MadsKristensen.ASPNETCoreTemplatePack20173*).

Außer im Fall *Empty* und *Web API* bekommt der Entwickler jeweils eine komplette kleine Website mit Layoutvorlage (auf Basis von Twitter Bootstrap) und optional auch mit Benutzerverwaltung. Bei der Benutzerverwaltung wählt der Entwickler wie bisher zwischen Windows (inklusive Active Directory), Azure Active

Directory oder einer lokalen Datenbank (für den Datenbankzugriff nutzt die Projektvorlage Entity Framework Core und Microsoft SQL Server. Andere Datenbanktreiber lassen sich leicht einbinden).

Projektaufbau

Abbildung 9-15 zeigt das Ergebnis, wenn Sie ein Webprojekt mit der Vorlage *Web Application (Model View Controller)* und der Auswahl *Individual Accounts/Store user accounts in-app* bei *Change Authentication anlegen*.

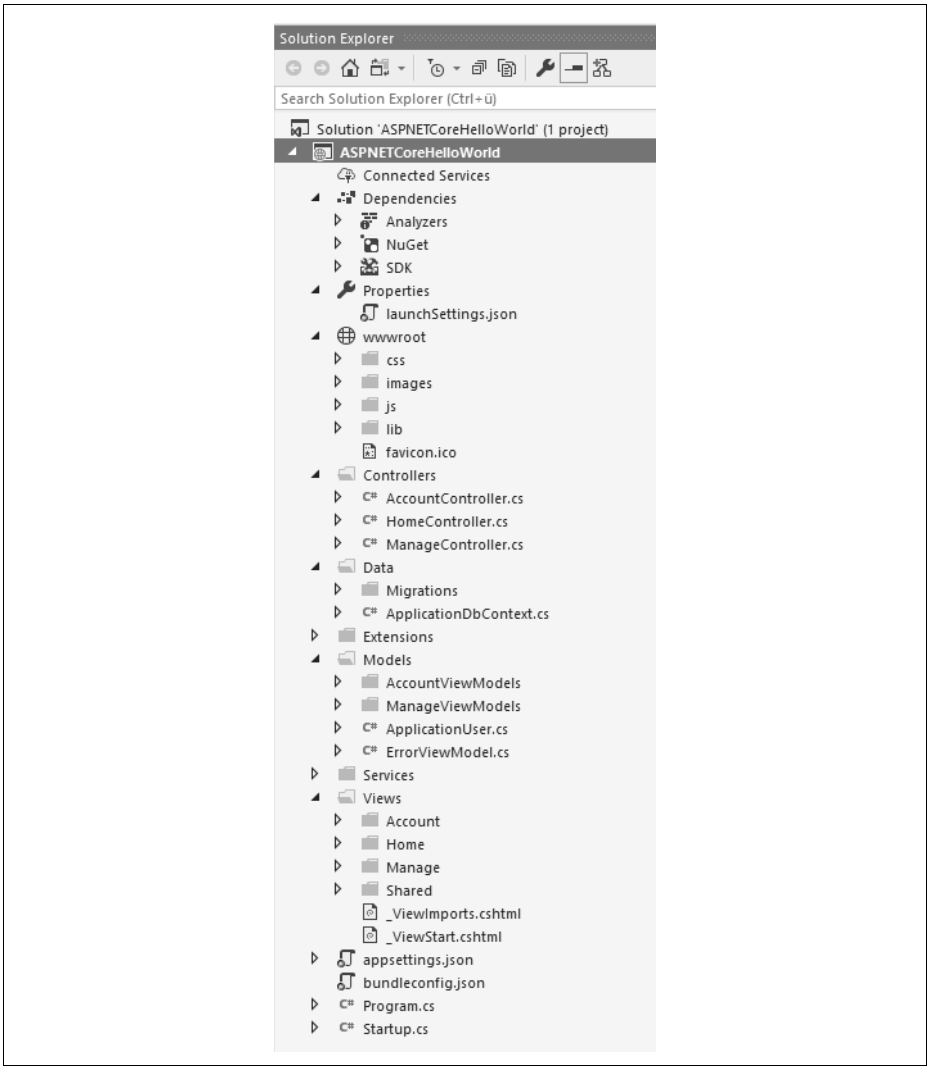


Abbildung 9-15: Aufbau eines MVC-Projekts mit ASP.NET Core auf .NET Core

In dem Projekt sind folgende Äste zu erkennen:

- */Connected Servies*: Eingebundene Cloud-Dienste (bei der vorliegenden Auswahl der lokalen Benutzerverwaltung sind im Standard keine Cloud-Dienste eingebunden).
- */Dependencies*: Generalisierung des bisherigen Astes *References* mit Unterästen für Abhängigkeiten zu .NET Core SDK, Visual Studio-Analyzer-DLLs und Nuget-Paketen. Bei Projektvorlagen, die auch ein Client-Projekt (zum Beispiel Vorlagen für Angular und React) beinhalten, erscheint hier auch noch der Node Paket Manager (NPM) im Ast */npm*. Wenn ein Bowser verwendet wird, gibt es auch */bowser*.
- */Properties/LauchSettings.json*: Einstellung für den Start der Anwendung aus Visual Studio heraus.
- */wwwroot*: Enthält alle statischen Dateien (*.html*, *.js*, *.css*, *.jpg*, *.csv*, *.json* etc.), die für die Verbreitung der Webanwendung gebraucht werden. Microsoft liefert hier die für die Projektvorlagen benötigten Dateien und Bibliotheken (Twitter Bootstrap und JQuery) im Ordner */wwwroot/lib* mit.
- */Controller*: MVC-Controller
- */Data*: Entity-Framework-Core-Kontextklasse für den Zugriff auf die lokale Benutzerdatenbank.
- */Extensions*: Erweiterungsmethoden.
- */Models*: Entitätsklassen für die lokale Benutzerverwaltung.
- */Services*: Dienst zum Senden von E-Mails für die lokale Benutzerverwaltung.
- */Views*: Views zu den Controllern.
- */appsetttings.json*: Anwendungskonfigurationsdatei mit Einstellungen der Verbindungszeichenfolge und Protokollierung (ersetzt *web.config*).
- */bundleconfig.json*: Einstellungen für das Bundling und Minifizieren von Dateien */wwwwroot*.
- *Program.cs*: Die Startdatei der Webanwendung mit einer `Main()`-Routine. Dieser Programmcode ist aber sehr knapp gehalten und delegiert die Konfiguration der Webanwendung auf *Startup.cs*.
- *Startup.cs*: Die eigentliche Konfiguration der Webanwendung.

Die Klasse Program

Das folgende Listing zeigt den Inhalt der Klasse `program.cs`. Hier werden einige Standardeinstellungen mit `CreateDefaultBuilder()` vorgenommen, und dann wird die Klasse `Startup` (*Startup.cs*) für den weiteren Startvorgang festgelegt.

Listing 9-1: Program.cs

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace ASPNETCoreHelloWorld
{
 public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

Dabei ist `CreateDefaultBuilder()` eine von Microsoft in ASP.NET Core 2.0 eingeführte Vereinfachung für mehrere bis dato explizit vorzunehmende Schritte, die Sie in Listing 9-2 sehen. Dazu folgende Erklärung:

- `UseKestrel()`: Verwendet den Standardwebserver, den ASP.NET Core mitliefert.
- `UseContentRoot(Directory.GetCurrentDirectory())`: Setzt den Wurzelordner der Webanwendung auf das aktuelle Verzeichnis.
- `ConfigureAppConfiguration()` mit `AddJsonFile("appsettings.json", …)`, `AddEnvironmentVariables()` sowie `AddCommandLine(args)`: Lädt die Konfigurationseinstellungen aus der Datei *appSettings.json* und die Umgebung die Kommandozeilenparameter.
- `ConfigureLogging()` mit `.AddConsole()` und `logging.AddDebug()`: Protokollierung an die Konsole und das Debug-Fenster.
- `UseIISIntegration()`: Integriert Kestrel in die Internet Information Services (IIS), wenn Kestrel von dort gestartet wurde.

Weitere Informationen und Optionen für den Start der ASP.NET-Core-Anwendung finden Sie unter *docs.microsoft.com/en-us/aspnet/core/fundamentals/hosting?tabs=aspnetcore2x*.

Listing 9-2: Funktionsweise von CreateDefaultBuilder() *(github.com/aspnet/MetaPackages/blob/dev/src/Microsoft.AspNetCore/WebHost.cs)*

```
   public static IWebHostBuilder CreateDefaultBuilder(string[] args)
                {
                    var builder = new WebHostBuilder()
                        .UseKestrel((builderContext, options) =>
                        {

options.Configure(builderContext.Configuration.GetSection("Kestrel"));
                        })
```

```
                        .UseContentRoot(Directory.GetCurrentDirectory())
                        .ConfigureAppConfiguration((hostingContext, config) =>
                        {
                            var env = hostingContext.HostingEnvironment;


                            config.AddJsonFile("appsettings.json", optional: true,
reloadOnChange: true)

.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
                                    reloadOnChange: true);


                            if (env.IsDevelopment())
                            {
                                var appAssembly = Assembly.Load(new
AssemblyName(env.ApplicationName));
                                if (appAssembly != null)
                                {
                                    config.AddUserSecrets(appAssembly, optional:
true);
                                }
                            }


                            config.AddEnvironmentVariables();


                            if (args != null)
                            {
                                config.AddCommandLine(args);
                            }
                        })
                        .ConfigureLogging((hostingContext, logging) =>
                        {

logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
                            logging.AddConsole();
                            logging.AddDebug();
                        })
                        .UseIISIntegration()
                        .UseDefaultServiceProvider((context, options) =>
                        {
                            options.ValidateScopes =
context.HostingEnvironment.IsDevelopment();
                        });


                    if (args != null)
                    {
                        builder.UseConfiguration(new
ConfigurationBuilder().AddCommandLine(args).Build());
                    }


                    return builder;
                }
```

Klasse Startup und Middleware

Die Datei *startup.cs* besteht aus zwei wesentlichen Methoden: `public void ConfigureServices(IServiceCollection services()` und `public void Configure(IApplicationBuilder app, IHostingEnvironment env)`:

- `ConfigureServices()` konfiguriert die per Dependency Injection injizierbaren Objekte.
- `Configure()` baut die sogenannte Middleware-Pipeline von ASP.NET Core auf und registriert dazu Middleware-Komponenten in einer bestimmten Reihenfolge beim übergebenen `IApplicationBuilder`. Der zweite Parameter, `IHostingEnvironment`, informiert über die Umgebung, in der die Anwendung läuft. Eine eingehende HTTP-Anfrage wird von der ersten konfigurierten Middleware-Komponente verarbeitet und dann an die nächste weitergeleitet. Jede dieser Komponenten kann die Anfrage direkt beantworten oder zur Beantwortung beitragen. Eine Middleware-Komponente kann die Anfrage auch an ein Web-Framework, wie MVC, weiterleiten. Die erzeugte Antwort leitet ASP.NET Core abermals durch die Pipeline. Nun hat jede Komponente erneut die Möglichkeit, die Antwort zu ergänzen. Beispielsweise könnten sie Kopfzeilen hinzufügen oder im Fehlerfall auf eine Fehlerseite umleiten. Nachdem ASP.NET Core wieder die erste Middleware-Komponente aufgerufen hat, steht die endgültige Antwort fest, und ASP.NET sendet diese zurück zum Aufrufer.

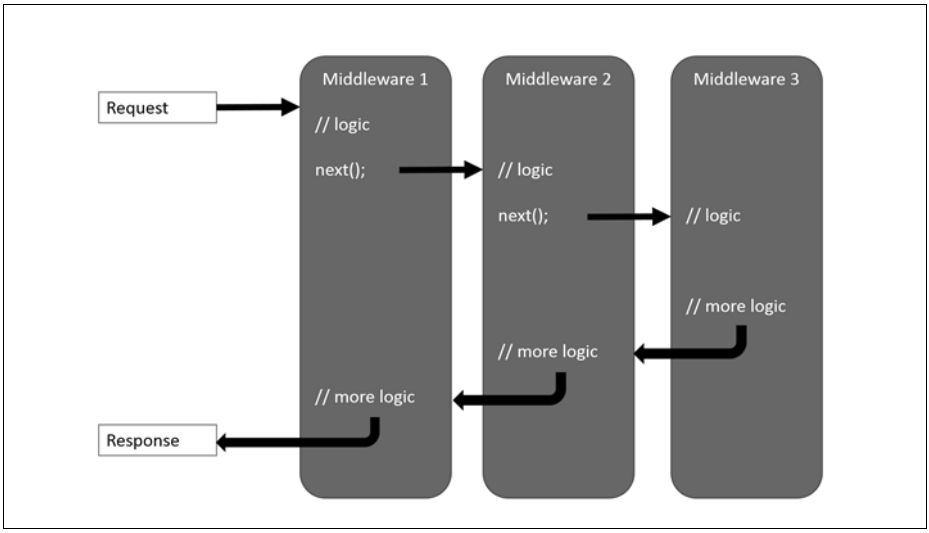


Abbildung 9-16: Middleware in ASP.NET Core (Urheber: Microsoft (docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware))

ASP.NET Core ist extrem modular aufgebaut. So sind auch Funktionen wie das Ausliefern von statischen Dateien in ASP.NET Core explizit per Middleware zu konfigurieren, siehe `app.UseStaticFiles()` in Listing 9-3. Auch andere Funktionen,

die im klassischen ASP.NET Standard waren, müssen nun explizit konfiguriert werden, zum Beispiel:

- Für das Auflisten eines Verzeichnisinhalts: `app.UseDirectoryBrowser()`.
- Für das Aufrufen von Standarddateien (*default.htm*, *default.html*, *index.htm* oder *index.html*) beim Zugriff auf einen Ordner: `app.UseDefaultFiles()`. Dabei ist die Reihenfolge wichtig: `app.UseDefaultFiles()` muss vor `app.UseStaticFiles()` kommen.
- Der Aufruf `app.UseFileServer()` kombiniert `app.UseDefaultFiles()`, `app.UseStaticFiles()` und `app.UseDirectoryBrowser()` in einem Aufruf.

Diese Methoden haben jeweils einen optionalen Parameter zur Festlegung von Optionen (zum Beispiel Ändern der Namen der Standarddateien bei `app.UseDefaultFiles()`).

Für die Entwicklungszeit wird auch Browser Link aktiviert, das heißt, wie bisher kann der Entwickler aus Visual Studio heraus per Debugging oder *View in Browser* gestartete Browserinstanzen zum Reload auffordern.

Bei den Fehlerseiten wird zwischen Entwicklungsumgebung (ausführliche Fehlerseite, `app.UseDeveloperExceptionPage()`) und Betriebsumgebung (spartanische Fehlerseite) unterschieden. Ob die aktuelle Umgebung die Entwicklungsumgebung ist, liefert `IHostingEnvironment` per Methode `IsDevelopment()`. `IsDevelopment()` basiert auf der Umgebungsvariablen *ASPNETCORE_ENVIRONMENT*. Wenn dort der Wert *Development* ist, liefert `IsDevelopment()` ein *true* zurück. Netterweise setzt Visual Studio automatisch diese Variable, und bei Bedarf lässt sie sich in den Projekteigenschaften anpassen.

Listing 9-3: Startup.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using ASPNETCoreHelloWorld.Data;
using ASPNETCoreHelloWorld.Models;
using ASPNETCoreHelloWorld.Services;

namespace ASPNETCoreHelloWorld
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
```

```
        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to
the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString(
            "DefaultConnection")));

            services.AddIdentity<ApplicationUser, IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>()
                .AddDefaultTokenProviders();

            // Add application services.
            services.AddTransient<IEmailSender, EmailSender>();

            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the
HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseBrowserLink();
                app.UseDeveloperExceptionPage();
                app.UseDatabaseErrorPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
            }

            app.UseStaticFiles();

            app.UseAuthentication();

            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

Referenzen/Pakete

Spannend bei den Referenzen ist das Nuget-Paket Microsoft.AspNetCore.All. Während Microsoft in ASP.NET Core 1.x nur die zwingend benötigten Pakete in den Projektvorlagen referenziert hat, findet man hier nun mit dem Paket Microsoft.AspNetCore.All (*www.nuget.org/packages/Microsoft.AspNetCore.All/2.0.0-pre*

view1-final) ein Meta-Paket, das alle verfügbaren ASP.NET Core und Entity Framework Core auf einmal referenziert. Alle Pakete dafür liegen im .NET Core Runtime Store:
C:\Program Files\dotnet\store\x64\netcoreapp2.0, der mit der Installation der .NET Core Runtime oder dem .NET Core SDK gefüllt wird.

In ASP.NET Core 1.x musste der Entwickler viele Paketreferenzen hinzufügen, sobald er etwas mehr machen wollte, als die Projektvorlage vorsah. Durch die Referenzierung des Meta-Pakets hat der Entwickler diese Aufgabe nicht mehr. Aber werden nun nicht zahlreiche Pakete referenziert, die später nicht gebraucht werden, was ja dem Gedanken von .NET Core widersprechen würde? Nein, weil die nicht benötigten Pakete beim Erstellen des Deployment-Pakets aussortiert werden.

Dennoch hat Microsoft eingesehen, dass eine Referenz zum Paket Microsoft.AspNetCore.All doch etwas viel für die meisten Fälle ist. In ASP.NET Core 2.1 gibt es nun ein weiteres, abgespecktes Meta-Paket. Während das bisherige »All«-Paket zahlreiche Nuget-Pakete umfasst, die man in einer Webanwendung selten braucht (zum Beispiel SQLite und den InMemory-Treiber), ist das neue in den Vorlagen verwendete Meta-Paket Microsoft.AspNetCore.App kompakter. Hier ist nur noch der Provider für Microsoft SQL Server enthalten (andere kann man selbstverständlich zusätzlich manuell einbinden).

Gespeichert werden Referenzen in einem .NET-Core-Projekt nicht mehr in der packages.json, sondern direkt in der *.csproj*-Datei in XML-Form im Tag `<PackageReference>`. Zudem findet man dort im `<DotNetCliToolReference>` Erweiterungen für das .NET-Core-CLI-Werkzeug (*dotnet.exe*).

ASPNETCoreHelloWorld.csproj

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <UserSecretsId>aspnet-ASPNETCoreHelloWorld-B2585648-5F47-43BE-BECA-88ECC6610BD0</UserSecretsId>
  </PropertyGroup>


  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.1" PrivateAssets="All" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.2"
    PrivateAssets="All" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1" />
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.2" />
  </ItemGroup>

</Project>
```

Abbildung 9-17: Aufbau der .csproj-Datei in einem ASP.NET-Core-Webprojekt

Ursprünglich in der Beta-Phase und in .NET Core 1.0 hatte Microsoft die Referenzen in einer Datei project.json gespeichert, ist davon aber wieder abgekommen.

Eine *.csproj*-Datei eines .NET-Core-Projekts kann man in Visual Studio einfach über den Eintrag *Edit .csproj* im Kontextmenü des Projektes im Solution Explorer bearbeiten. Die manuelle Bearbeitung der Datei ist leider manchmal etwas mühsam, da man nicht alles im Solution Explorer sieht und einstellen kann, wie zum Beispiel die Tags `<DotNetCliToolReference>`.

Zum Hinzufügen eines Nuget-Pakets nutzt man wie bei klassischen .NET-Projekten den in Visual Studio eingebauten GUI für Nuget (Aufruf über *Manage Nuget Packages*) oder die Nuget Package Manager Console (NPMC) mit dem PowerShell-Commandlet Install-Package, zum Beispiel

Install-Package AutoMapper

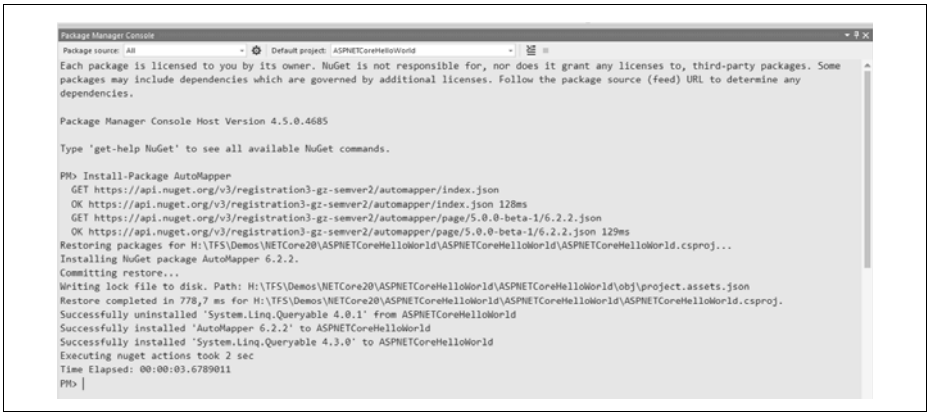


Abbildung 9-18: Paketinstallation mit der Nuget Package Manager Console (NPMC)

Wenn in den Einstellungen so gewählt, führt Visual Studio automatisch einen Package Restore beim Übersetzen eines Projekts aus. Alternativ können Sie mit *dotnet restore* einen Package Restore an der Kommandozeile starten, wenn man sich im aktuellen Projekt befindet.

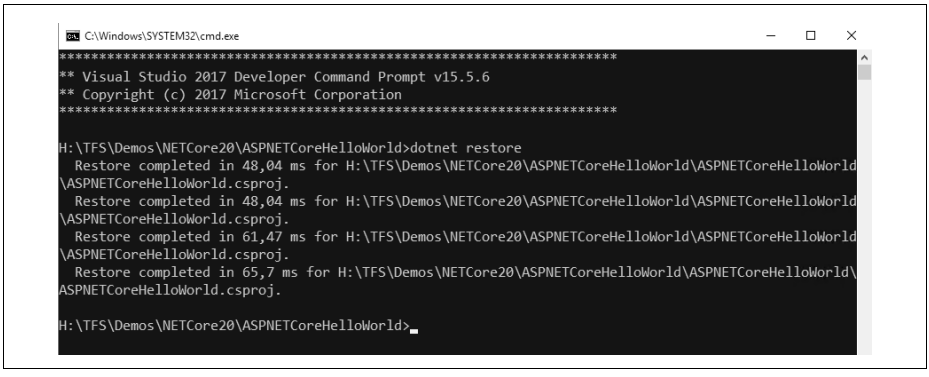


Abbildung 9-19: dotnet restore

Oft sind die beim Anlegen eines Projekts referenzierten Paketversionen bereits veraltet, denn die von Visual Studio verwendeten Versionen der einzelnen Pakete sind in den Projektvorlagen festgelegt, die Microsoft mit Visual Studio beziehungsweise den Updates dafür verbreitet. Sie sollten daher direkt nach dem Anlegen eines Projekts den Nuget Package Manager aufrufen und alle Pakete auf die aktuelle Version bringen.

Übersetzen und Debugging

Nach dem erfolgreichen Übersetzen finden Sie eine DLL und ergänzende JSON-Dateien im Unterordner *\bin\Debug\netcoreapp2.0*.

Leider hat Microsoft das in der Beta-Version von ASP.NET Core vorhandene Konzept der reinen Kompilierung im RAM verworfen (*github.com/aspnet/Home/issues/1471*). Auch die in der Beta-Phase enthaltene komplette Kompilierung der Views zur Entwicklungszeit ist wieder entfallen: Fehler zeigt Visual Studio zwar nach etwas Wartezeit durch rote Linien an, aber der Compiler meldet im Ausgabefenster auch bei Fehlern in Views »succeeded«.

Abbildung 9-20: Fehler im View werden von Visual Studio zwar markiert, aber beim Build kommt dennoch »succeeded« heraus.

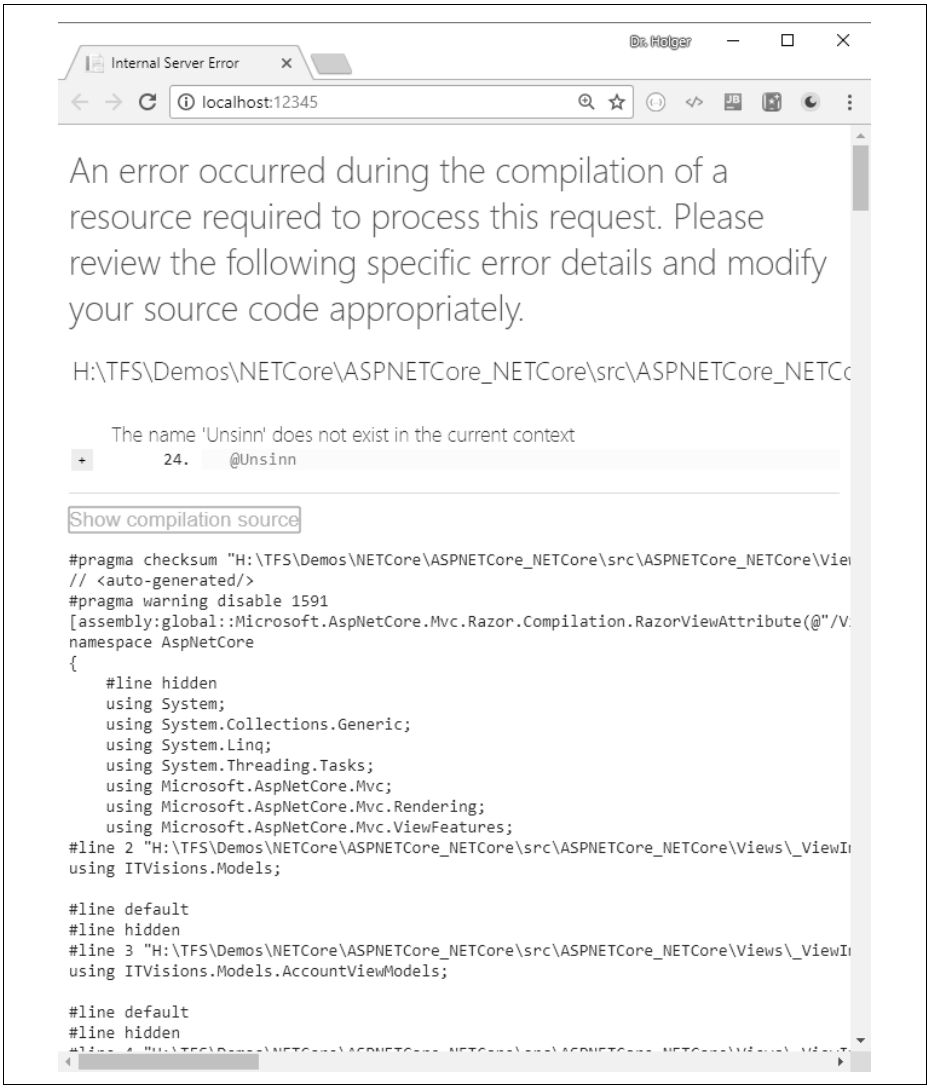


Abbildung 9-21: Der Fehler fliegt erst zur Laufzeit auf.

Wie bisher kann man auch ein ASP.NET-Core-Projekt direkt in Visual Studio im Debugger oder – etwas schneller ohne Debugger – per *View in Browser* (Tastenkürzel (Strg)+(Alt)+(W)) starten.

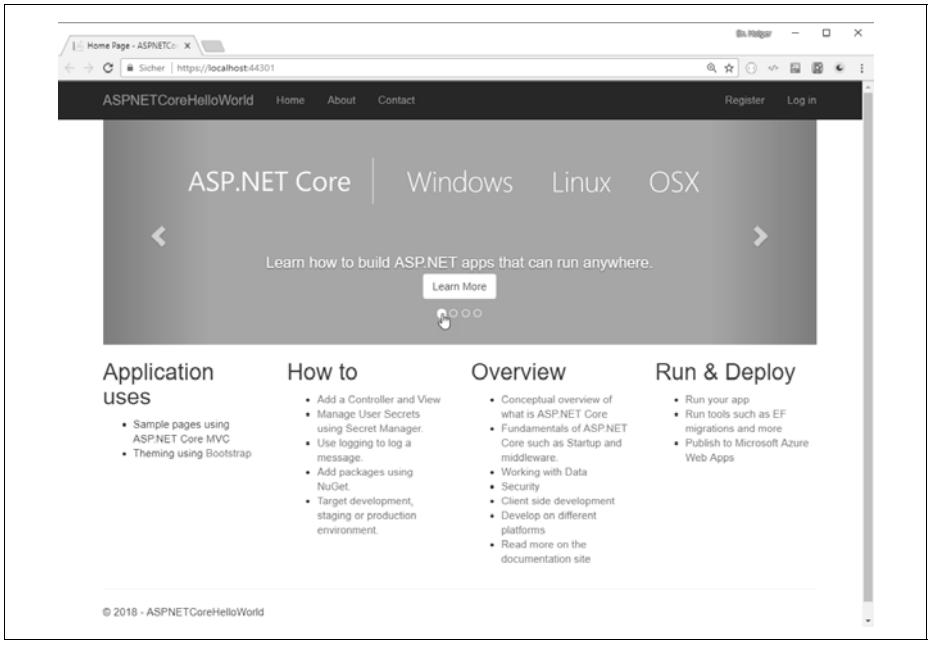


Abbildung 9-22: Die Webanwendung aus der gewählten Vorlage im Browser

Anders als das bisherige ASP.NET MVC kompiliert Visual Studio bei einem Browser-Reload eine Controller-Klasse automatisch neu, wenn sich an ihm etwas geändert hat. Allerdings funktioniert dies nur in Verbindung mit *View in Browser*, nicht mit dem Debugger.

Alternativ kann man die Webanwendung nun auch von der Kommandozeile aus starten, indem man mit *dotnet.exe* die erzeugte DLL aufruft (siehe Abbildung 9-23), das heißt, das Self-Hosting außerhalb des IIS beziehungsweise des IISExpress ist direkt ohne weitere Arbeitsschritte möglich.

Abbildung 9-23: Ausführung der Webanwendung mit dotnet.exe

Bitte beachten Sie, dass die Portnummer für den Start aus Visual Studio heraus in der Datei */Properties/LauchSettings.json* steht, während beim Start via *dotnet.exe* im Standardport 5000 verwendet wird. Um diese URL zu ändern, muss man in

program.cs nach `.UseStartup<Startup>()` ein `.UseUrls("http://localhost:nummer")` mit der gewünschten Portnummer einfügen.

Listing 9-4: Modifizierte program.cs*-Datei*

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace ASPNETCoreHelloWorld
{
 public class Program
 {
  public static void Main(string[] args)
  {
   BuildWebHost(args).Run();
  }

  public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
          .UseStartup<Startup>()
          .UseUrls("http://localhost:5050")
          .Build();
 }
}
```

Deployment

.NET Core bietet zwei Möglichkeiten zum Deployment einer .NET-Core-Anwendung:

- Framework-dependent Deployments (FDD): Hier entsteht eine Portable Applications (PA) alias.
- Self-contained Deployment (SCD): Hier entsteht eine Self-Contained Application (SCA).

Eine Portable Application setzt voraus, dass auf dem Zielsystem .NET Core in der passenden (oder höheren) Version bereits vorab installiert wurde, so wie dies beim .NET »Full« Framework üblich ist. Die Portable Application ist ein plattformneutrales Paket, das deutlich kleiner ist als das SCA-Paket.

Bei einer Self-Contained Application umfasst das Deployment der Anwendung alles, was man braucht, um die Anwendung zu betreiben, also auch die passende .NET-Core-Runtime-Version. Man kann sie auf einen USB-Stick kopieren und direkt von dort starten, ohne sie vorher auf dem Zielsystem .NET Core installieren zu müssen. Dabei hat man die volle Kontrolle, welche Version von .NET Core man mitliefert. Eine Self-Contained Application braucht aber natürlich deutlich mehr Festplattenplatz. Viele solcher Anwendungen auf einem System brauchen dann eben viel Platz. Das Installationspaket ist zudem plattformspezifisch, das heißt, für jede Plattform muss ein solches Paket erstellt werden.

Während man eine Portable Application in Visual Studio per Menü *Build/Publish* generieren kann, ist eine SCA aktuell nur per Kommandozeile zu erstellen.

Das folgende PowerShell-Skript zeigt die Erzeugung und den Start einer Portable Application, die in Form einer DLL vorliegt.

```
$Projektname = "ASPNETCoreHelloWorld_NETCore"
$destination = "t:\$Projektname" + "_Deploy"
cd $psscriptroot\$Projektname
# Framework-Dependent Deployment (FDD) / Portable Application (PA)
dotnet publish -c release --framework netcoreapp2.0 -o $destination\PA
dotnet $destination\PA\$Projektname.dll
```

Das folgende PowerShell-Skript zeigt die Erzeugung und den Start einer SCA. Hier entsteht eine *.exe*-Datei.

```
$Projektname = "ASPNETCoreHelloWorld_NETCore"
$destination = "t:\$Projektname" + "_Deploy"
cd $psscriptroot\$Projektname
# Self-Contained Deployment (SCD) / Self-contained Application (SCA)
dotnet publish -c release --runtime win10-x64 --self-contained --framework
netcoreapp2.0 -o $destination\SCA
"$destination\SCA\$Projektname.exe"
```

Abbildung 9-24 zeigt das Deployment-Paket der Portable Application.

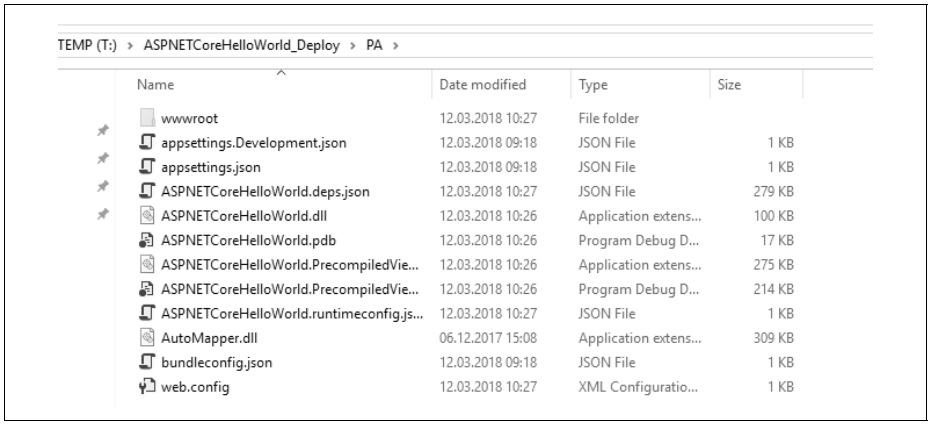


Abbildung 9-24: Eine ASP.NET-Core-Webanwendung als Portable Application

Das Deployment-Paket für die SCA ist sehr viel umfangreicher. Der Screenshot zeigt sogar nur einen Ausschnitt aus den entstandenen Dateien.

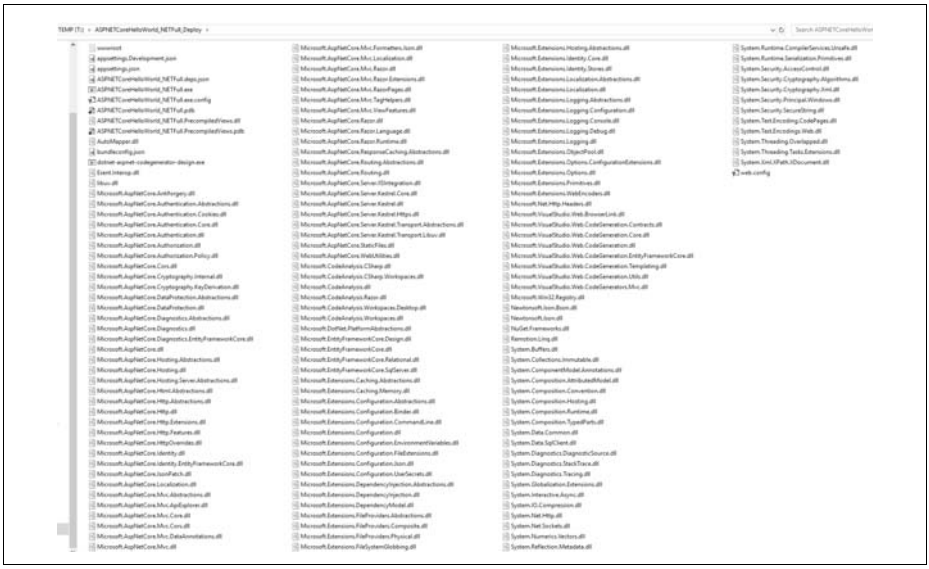


Abbildung 9-25: Eine ASP.NET-Core-Webanwendung als SCA (Ausschnitt)

Ein Satz in der Dokumentation macht nachdenklich: »It is possible for the .NET Core runtime and libraries to change without your knowledge in future releases. In rare cases, this may change the behavior of your app.« (*docs.microsoft.com/de-de/dotnet/articles/core/deploying/index*). Microsoft lässt also anklingen, dass Breaking Changes in der .NET-Core-Welt häufiger vorkommen als bei dem bisherigen .NET Framework. Tatsächlich gab es Breaking Changes zwischen ASP.NET Core 1.x und ASP.NET 2.0 (erlaubt bei Änderungen der Hauptversion nach dem Prinzip des Semantic Versioning, vgl. *semver.org*). Von solchen Änderungen ist man mit einer Self-Contained Application nur in der Entwicklung, aber nicht in der Produktion betroffen.

Der Vorteil der Portable Application ist jedoch, dass nur ein einziges Deployment-Paket für alle Plattformen entsteht, während es bei der Self-Contained Application für jede Zielplattform (zum Beispiel win7-x64, win8-x86, win10-x64, ubuntu.15.04-x64, debian.8.2-x64, osx.10.11-x64) ein Paket geben muss, weil ja .NET Core inklusive der Abstraktionsschicht für die jeweilige Zielplattform enthalten ist.

Leider verwendet Visual Studio beim Kompilieren nicht das Kommandozeilenwerkzeug dotnet.exe, und es gibt keine Unterschiede zwischen der Kompilierung in Visual Studio und der Kompilierung mit dotnet build. Daher kann es vorkommen, dass ein Projekt zwar in Visual Studio, aber nicht mit dotnet buildkompiliert. Eine solche Situation entsteht zum Beispiel, wenn man ein Projekt mit ASP.NET Core 1.x in Visual Studio 2015 (unter Verwendung von project.json) erstellt hat und es dann in Visual Studio 2017 zwangsweise auf das *.csproj*-basierte Format umstellt. Dabei baut Visual Studio ein Tag in die *.csproj*-Datei ein, die *dotnet build* nicht mag. Fehlermeldung von *dotnet build* ist dann:

error NU1003: PackageTargetFallback and AssetTargetFallback cannot be used together. Remove PackageTargetFallback(deprecated) references from the project environment.

Immerhin sagt die Fehlermeldung, was konkret zu tun ist: Man muss das Tag `<PackageTargetFallback>` aus der *.csproj*-Datei entfernen.

Ein weiteres Problem gibt es beim Deployment: Einige Punkte, die beim Kompilieren nur zu Warnungen führen, verhindern das erfolgreiche Deployment.

ASP.NET Core auf dem klassischen .NET Framework

ASP.NET Core kann nicht nur auf .NET Core betrieben werden, sondern auch auf dem klassischen .NET Framework ab Version 4.6.1.

Microsoft hatte ursprünglich im Mai 2017 angekündigt, dass die Unterstützung für das klassische .NET Framework als Unterbau für ASP.NET Core, die es in ASP.NET Core 1.x gab, in ASP.NET Core 2.x nicht mehr fortgeführt wird. Auf Protest der Entwicklergemeinde hat Microsoft diese Entscheidung aber revidiert (*www.heise.de/developer/meldung/Microsoft-will-ASP-NET-Core-nicht-mehr-auf-dem-klassischen-NET-anbieten-3708715.html*).

Einsatzszenarien

Dies ergibt in folgenden Fällen Sinn:

- Schrittweise Migration einer bestehen ASP.NET-MVC-Anwendung nach ASP.NET Core, weil ein Teil des Umstellungsaufwands entfällt, wenn nicht zusätzlich auch eine Migration von .NET Framework zu .NET Core erfolgen muss.
- Erstellen einer neuen ASP.NET-Core-Anwendung zunächst auf .NET Framework, wenn benötigte Klassen in .NET Core noch fehlen beziehungsweise benötigte Softwarekomponenten noch nicht auf .NET Core laufen.

Allerdings ist eine ASP.NET-Core-Anwendung auf dem klassischen .NET Framework nicht plattformunabhängig, das heißt, sie läuft nur unter Windows.

Wenn möglich, sollten Sie ASP.NET Core auf .NET Core betreiben. Diese Lösung hat die Vorteile der Plattformunabhängigkeit, die Option zum Betrieb auf Windows Nano Server, eine bessere Performance und die bessere Zukunftssicherheit.

Anlegen von ASP.NET-Core-Projekten mit dem klassischen .NET Framework

Neue ASP.NET-Core-Projekte auf Basis von .NET »Full« Framework können Sie nicht mit *dotnet new* anlegen, sondern nur mit Visual Studio. Wie bei .NET Core wählt man zunächst:

- *Add New Project/Visual C#/Web/ASP.NET Core Web Application*

  oder
- *Add New Project/Visual C#/.NET Core/ASP.NET Core Web Application*

Im zweiten Dialogschritt wählt man dann oben links ».NET Framework« statt ».NET Core«.

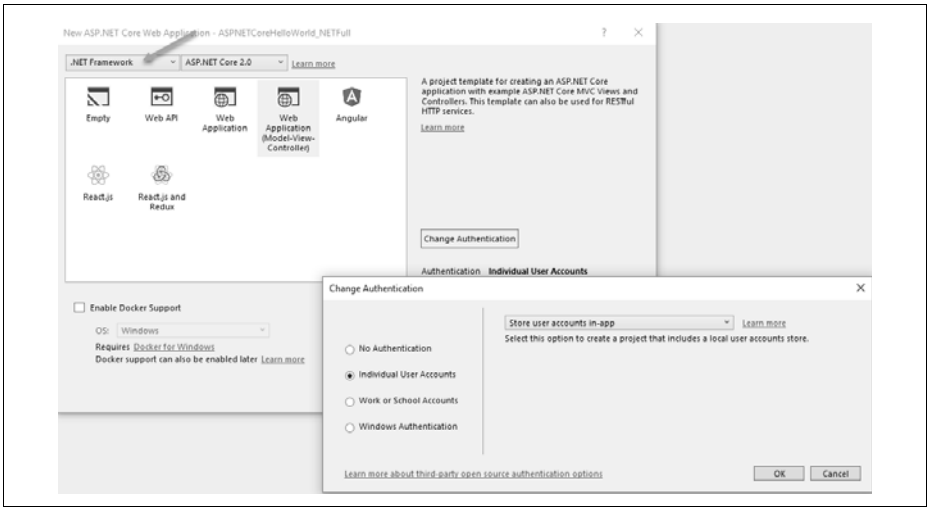


Abbildung 9-26: Im zweiten Dialogschritt oben links .NET Framework *statt* .NET Core *auswählen*

Projektaufbau

In dem daraus entstehenden Projekt gibt es folgende Unterschiede:

- Im Ast */Dependencies* gibt es */Assemblies* für Verweise zu DLL des .NET Frameworks. Den Unterast */SDK* gibt es nicht.
- Unter */Dependencies/Nuget* wird nicht das Paket Microsoft.AspNetCore.All referenziert, sondern mehrere Einzelpakete von ASP.NET Core (siehe nächste Abbildung).

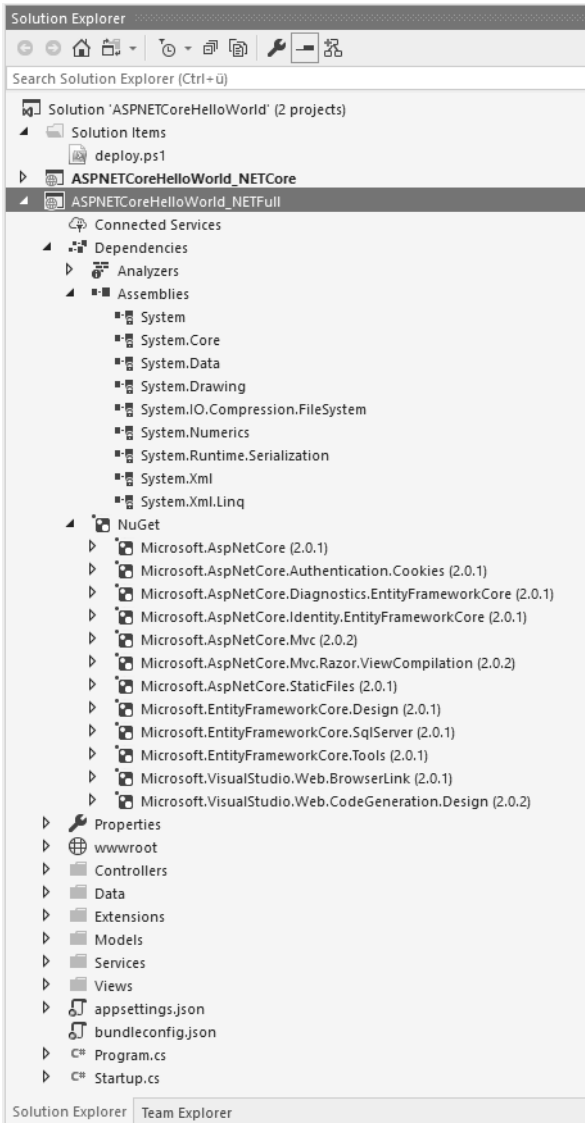


Abbildung 9-27: Aufbau eines MVC-Projekts mit ASP.NET Core auf .NET »Full« Framework

Abweichend von den Beta-Versionen verzichtet Microsoft darauf, in den ASP.NET-Core-Vorlagen gleichzeitig für .NET Core und das klassische .NET Framework zu kompilieren. Die Projektstruktur ist dadurch einfacher. Ebenfalls nicht mehr im Standard vorhanden ist die Verwendung der JavaScript-basierten Build-Werkzeuge Gulp oder Grunt. Microsoft setzt nun für die Minifizierung und das Bündeln wieder auf eine Visual-Studio-Erweiterung (*visualstudiogallery.msdn.microsoft.com/9ec27da7-e24b-4d56-8064-fd7e88ac1c40*). Der Bundler & Minifier wird über eine Datei *bundleconfig.json* im Projektwurzelverzeichnis konfiguriert. Minifizierung und Bündeln kann der Entwickler dann über Task Runner Explorer in den Übersetzungsprozess einbinden oder auch manuell über Kontextmenübefehle in Solution Explorer auslösen.

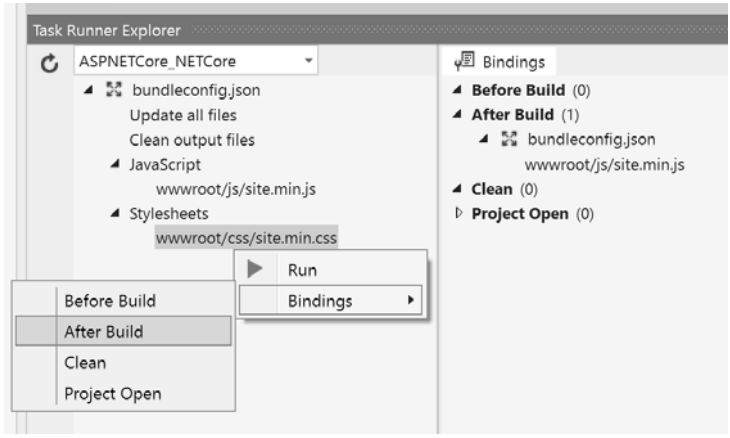
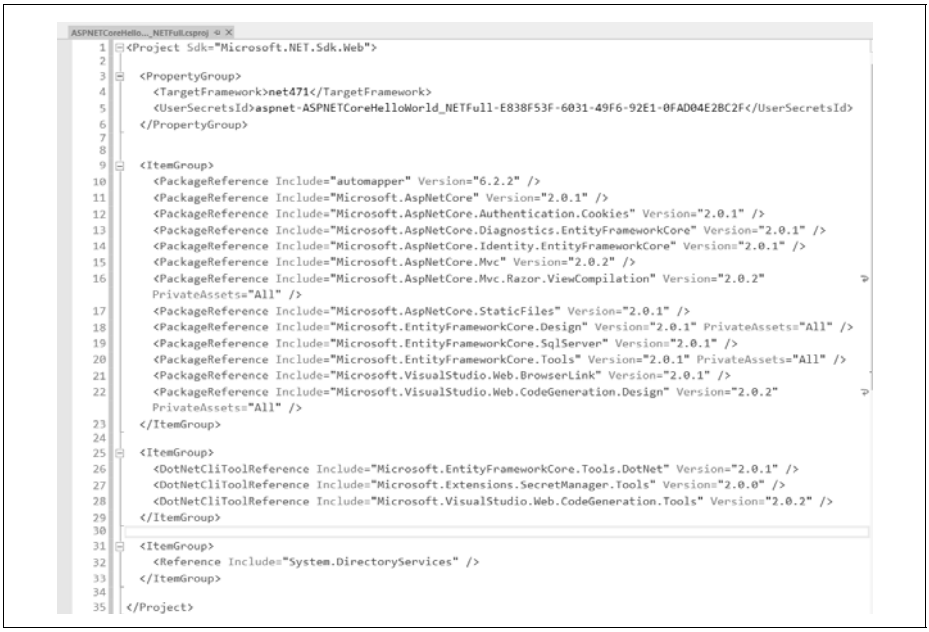


Abbildung 9-28: Einbinden von Tasks des Bundler & Minifier in den Übersetzungsprozess

Referenzen/Pakete

Nuget-Pakete verwaltet man wie bei .NET-Core-basierten Projekten. Auch hier landen die Referenzen als Tag `<PackageReference>` in der *.csproj*-Datei. Zusätzlich kann man wie in klassischen .NET-Projekten Referenzen auf Assemlies aus dem .NET Framework mit *Add Reference* erstellen. Diese landen in der *.csproj*-Datei als Tag `<Reference>`.



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

  <PropertyGroup>
    <TargetFramework>net471</TargetFramework>
    <UserSecretsId>aspnet-ASPNETCoreHelloWorld_NETFull-E838F53F-6031-49F6-92E1-0FAD04E2BC2F</UserSecretsId>
  </PropertyGroup>


  <ItemGroup>
    <PackageReference Include="automapper" Version="6.2.2" />
    <PackageReference Include="Microsoft.AspNetCore" Version="2.0.1" />
    <PackageReference Include="Microsoft.AspNetCore.Authentication.Cookies" Version="2.0.1" />
    <PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="2.0.1" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="2.0.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.0.2" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.2"
    PrivateAssets="All" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.1" PrivateAssets="All" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.1" PrivateAssets="All" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.1" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.2"
    PrivateAssets="All" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1" />
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.2" />
  </ItemGroup>

  <ItemGroup>
    <Reference Include="System.DirectoryServices" />
  </ItemGroup>

</Project>
```

Abbildung 9-29: Beispiel für eine .csproj*-Datei für ein ASP.NET-Core-Projekt auf Basis .NET »Full« Framework*

Deployment

Ein Self-Contained Deployment ist nicht möglich, weil es diese Möglichkeit in .NET »Full« Framework nicht gibt. Auch ein Portable App-Deployment mit *dotnet build* ist nicht möglich. Man kann in Visual Studio nur *Build/Publish* wählen.

Bei *Target Runtime* steht in diesem Fall immer »Win7-x86«, auch wenn Sie auf einem anderen Betriebssystem arbeiten. Microsoft will damit sagen: Das Paket läuft auf Windows 7 (x86) und höher (inklusive x64).

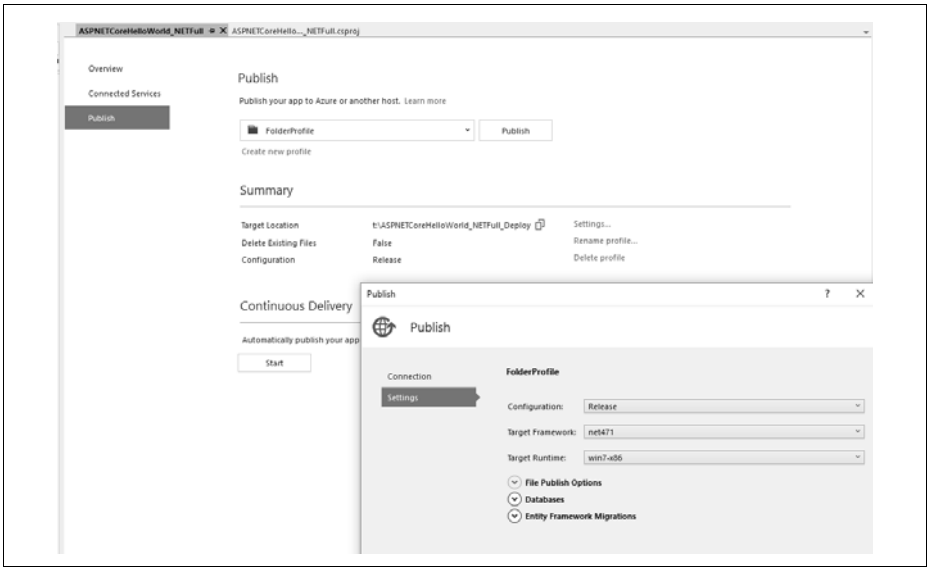


Abbildung 9-30: Publish in Visual Studio

Es entstehen neben einer *.exe*-Datei noch viele andere Dateien. Dennoch wird zusätzlich auf dem Zielsystem die vorherige Installation des passenden .NET Frameworks erwartet.

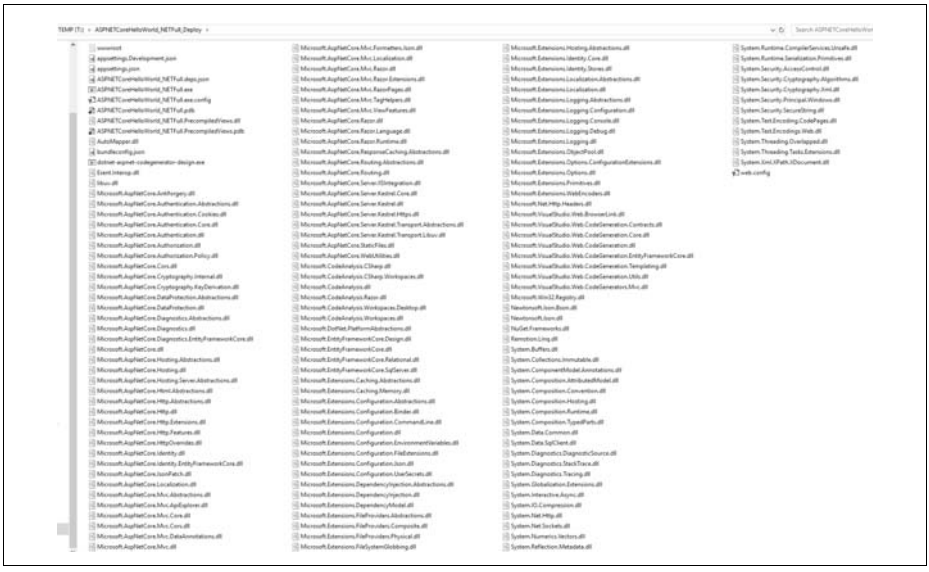


Abbildung 9-31: Ausgabe von Publish

In ASP.NET Core integrierte Webserver: Kestrel versus HTTP.sys (WebListener)

Das klassische ASP.NET war eine Erweiterung für die Internet Information Services (IIS) und integrierte sich über ein IIS-Modul. ASP.NET Core ist aber plattformneutral und zudem nicht an einen bestimmten Webserver gebunden. Zudem war bei ASP.NET Core das Ziel, das Self-Hosting (das heißt das Hosting in einem beliebigen Prozess, der nicht ein ausgewachsener Webserver ist, zum Beispiel in einer Konsolenanwendung) leicht zu ermöglichen.

ASP.NET Core liefert daher selbst zwei Webserver mit: Kestrel und HTTP.sys (in ASP.NET Core 1.x noch »WebListener« genannt).

- *Kestrel*: läuft sowohl unter Windows als auch unter Linux und macOS. Er kann als eigener Webserver laufen oder sich als Reverse Proxy Server in einen Webserver wie IIS, Apache oder Nginx integieren.
- *HTTP.sys*: läuft nur auf Windows auf Basis des HTTP.sys Kernel-Treibers von Windows. Http.sys integriert sich nicht in einen anderen Webserver, sondern dient nur dem Self-Hosting. Http.sys bietet einige zusätzliche Features für das Self-Hosting, die bei Kestrel nicht zur Verfügung stehen, zum Beispiel HTTP/2 mit TLS, Web-Sockets, Port Sharing und Windows-Authentifizierung.

Kestrel

Kestrel ist der Standard in allen Projektvorlagen, die Microsoft in Visual Studio und bei *dotnet new* mitliefert. Kestrel wird aktiviert in der Program-Klasse durch `WebHost.UseKestrel()` aus dem Nuget-Paket Microsoft.AspNetCore.Server.Kestrel. Der Aufruf `WebHost.CreateDefaultBuilder(args)` beinhaltet auch `UseKestrel()`.

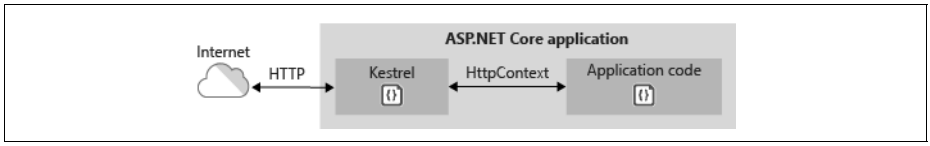


Abbildung 9-32: Kestrel als eigenständiger Webserver (Urheber: Microsoft (docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?tabs=aspnetcore2x))

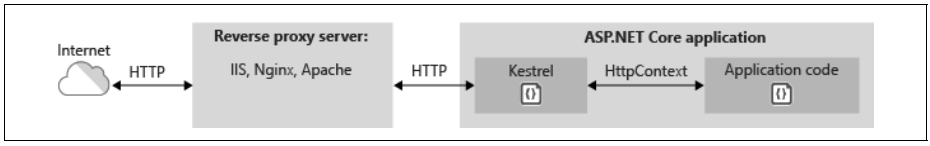


Abbildung 9-33: Kestrel als Reverse Proxy Server (Urheber: Microsoft (docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?tabs=aspnetcore2x)

Für die Integration in die Internet Information Services (IIS) von Windows wird das ASP.NET Core IIS-Module (*C:\Windows\System32\inetsrv\aspnetcore.dll*) benötigt. Dieses Modul erhält man durch die Installation von .NET Core Windows Server Hosting Bundle (*aka.ms/dotnetcore-2-windowshosting*) auf dem System mit den IIS. Zudem ist zu beachten, dass man in den IIS im Application Pool für die Website, die ASP.NET Core verwendet, bei *.NET CLR Version* Folgendes einstellt: »No Managed Code«. Das erscheint im ersten Moment paradox, weil ja .NET Core auch Managed Code ist. Dieser Managed Code läuft aber eben nicht in den IIS, sondern in einem getrennten Prozess (siehe Abbildung 9-34). Eine Schritt-für-Schritt-Klick-Anleitung dafür finden Sie unter (*docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/index?tabs=aspnetcore2x*).

Ebenso gibt es Anleitungen für die Integration in Nginx (*docs.microsoft.com/en-us/aspnet/core/host-and-deploy/linux-nginx?tabs=aspnetcore2x*) und Apache (*docs.microsoft.com/en-us/aspnet/core/host-and-deploy/linux-apache?tabs=aspnetcore2x*).

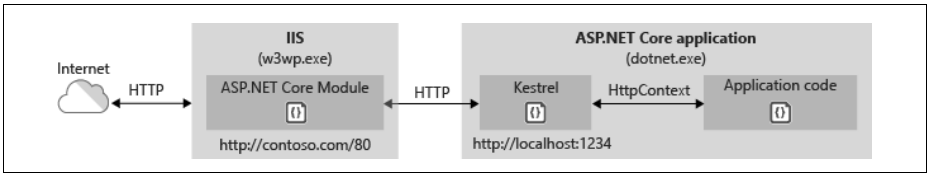


Abbildung 9-34: Integration von Kestrel in die IIS (Urheber: Microsoft (docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/aspnet-core-module))

Während das klassische ASP.NET im Worker Process der IIS lief, ist in ASP.NET Core 1.x und 2.0 nicht vorgesehen, dass ASP.NET Core direkt im Prozess eines Webservers läuft. Es soll hingegen als eigenständiger Prozess arbeiten, damit ASP.NET Core in vielen Webservern funktioniert und dort konsistent arbeitet. Allerdings gibt es hierbei auch einen Leistungsverlust. In ASP.NET Core 2.1 soll ASP.NET Core daher zumindest wieder im Worker Process der IIS unter Windows laufen können.

HTTP.sys

Abbildung 9-35 zeigt die Arbeitsweise von HTTP.sys.

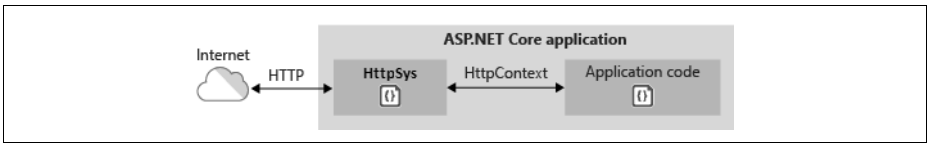


Abbildung 9-35: HTTP.Sys als eigenständiger Webserver (Urheber: Microsoft (docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?tabs=aspnetcore2x))

Für diesen Treiber wird das Paket Microsoft.AspNetCore.Server.HttpSys benötigt. Die Aktivierung erfolgt in der Program-Klasse auf dem WebHostBuilder durch den Aufruf von `UseHttpSys()` anstelle von `UseKestrel()` und `UseIISIntegration()`.

Listing 9-5: Verwendung von HTTP.sys anstelle von Kestrel

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Server.HttpSys;

namespace ITVisions
{
 public class Program
 {
  public static void Main(string[] args)
  {
   var host = new WebHostBuilder()
       //.UseKestrel()
       //.UseContentRoot(System.IO.Directory.GetCurrentDirectory())
       //.UseIISIntegration()
       .UseHttpSys(options =>
               {
                         // The following options are set to default values.
                         options.Authentication.Schemes = AuthenticationSchemes.None;
                options.Authentication.AllowAnonymous = true;
                options.MaxConnections = null;
                options.MaxRequestBodySize = 30000000;
                options.UrlPrefixes.Add("http://localhost:5000");
               })

       .UseStartup<Startup>()
       .Build();

   host.Run();
  }
 }
}
```

Wenn Sie sowohl `UseHttpSys()` als auch `UseKestrel()` aufrufen, kommt es nicht zum Fehler. Es gewinnt der letzte Aufruf. Ein Hosting in zwei Webservern parallel ist in einem Prozess nicht möglich.