.NET CORE 2.0

ŠKOLENIE

Vladimír Gašpar

TÁTO PREZENTÁCIA

- http://leteckaposta.cz/258198545
- http://github.com/gasparv/netcore-class

KTO SOM?

- Vladimír Gašpar
- Učím na FEI TU v Košiciach v odbore Hospodárska informatika
  - Vývoj Android aplikácií
  - Senzorické siete – zber, spracovanie, analýza dát
- Developer
  - React.JS, JavaScript (napr. D3.js, Highcharts, PhantomJS) – Intersoft, a.s.
  - ASP.NET MVC – ICOS, a.s.
  - ASP.NET Core MVC – TUKE, Katedrový IS

ČO JE CIEĽOM?

- Podať dostatočné informácie na to, aby si vedel:
  1. Kedy migrovať na .NET Core a kedy ostať pri .NET Frameworku
  2. Odhadnúť effort pre migráciu .NET Framework kódu (ASP.NET MVC)
  3. Zorientovať sa v konvenciách v .NET Core a najmä v ASP.NET Core MVC
  4. Využívať silné stránky .NET Core a písať efektívny kód
  5. Vedel sa preniesť cez dokonalosti a nedokonalosti EntityFramework Core

AKO TIETO CIELE DOSIAHNEŠ?

- Školenie je len štartér – nestačí na working knowledge ...
- Počas školenia
  - Základná filozofia, teória a praktické demo ukážky vám dajú dobrý základ pre ďalšie štúdium
  - Vlastné programovanie vás uvedie do základných praktických zručností
- Po školení
  - Kniha **Pro ASP.NET Core MVC 2** je dobrou referenčnou príručkou pre prípad, že niečo časom zabudnete
  - Účasť na živom projekte v .NET Core je nutnosť pre ďalšie rozvíjanie
  - Komplexnejší vývoj webu vrátane .NET Core - Microsoft Exam 70-486

ČO BUDE OBSAHOM?

- Teória platná pre .NET Core, ASP.NET Core MVC. Demo ukážky kódov a kľúčových vlastností.
  - Čo je/nie je .NET Core, CoreFX, inštalácia a konfigurácia (5%)
  - **Visual Studio Code** - .NET Core CLI príkazy, Základy tvorby projektov vo VS Code (5%)
  - **.NET Core 2.0, .NET Standard** – špecifiká C# 7.1, kompatibilita .NET Standard knižníc, DI v .NET Core 2.0 (5%)
  - **ASP.NET Core MVC 2.0** – projekt, DI, middleware, IIS, Kestrel, konfigurácie, API, Views v Razor (nie Razor pages!!!) (50%)
  - **Entity Framework Core** – Prístupy k tvorbe DB modelu, prístupy k načítavaniu dát z kontextu DB (25%)
  - **Migrácia z .NET Framework** – kompatibilné knižnice, rozdiely v implementácii FE a BE (5%)
  - **.NET Core a Docker** – projekt ako docker kontajner, konfigurácia, docker-compose.yml (5%)

PRAVIDLÁ ... ?

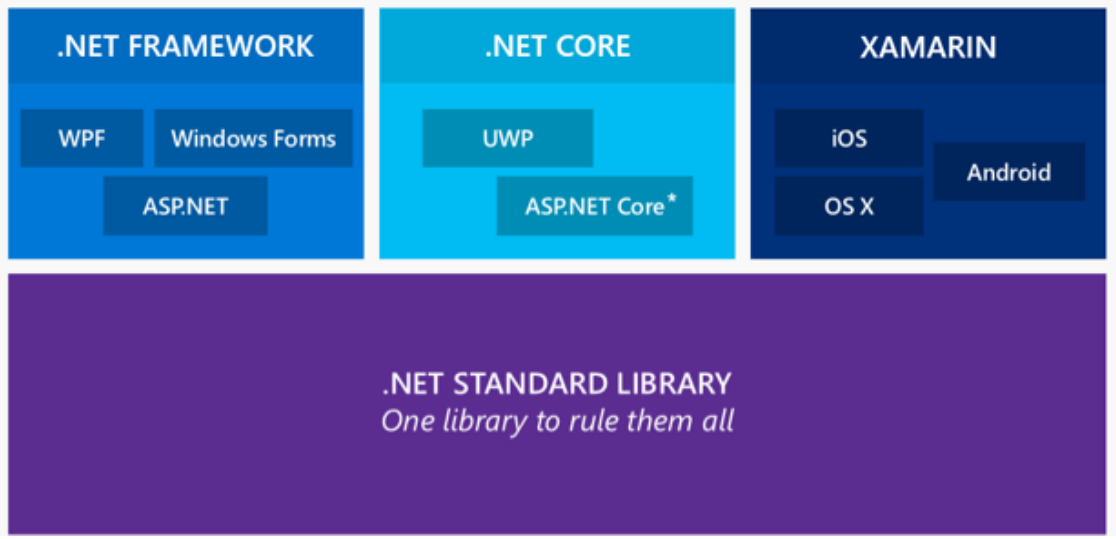
- Všetci mi môžete tykať – ako je vám lepšie
- Môžem tykať aj ja vám?
- Ak budete mať akékoľvek otázky/opravy k tomu čo hovorím/prezentujem, alebo akýkoľvek problém pokojne ma prerušte
- Určite neviem všetko – školenie by malo byť odrazovým mostíkom ak ste sa s .NET Core doposiaľ nestretli, resp. prechádzate .NET Framework-u
- Ak budete mať pocit, že potrebujete pauzu, daj mi vedieť – môj zvyk je 5-10 min pauza po cca. 2 hodinách

ČASŤ 1

O ekosystéme .NET Core

- Roadmapa produktov .NET
- Čo je a čo nie je .NET Core
- Typy projektov
- Kedy .NET Core a kedy .NET Framework
- Súčasný stav produktov .NET Core
- .NET Standard

ROAD MAP PRODUKTOV .NET



.NET Framework	1.0	2.0	3.0	3.5	4.0	4.5	4.6	4.7	4.7.1	4.7.2
.NET Core						2015(5) (1.0.0 beta)	1.0	1.1	2.0 (9/2017)	2.1 (5/2018)
.NET Standard						1.0	...	1.5	1.6	2.0

https://github.com/dotnet/core/blob/master/roadmap.md

ČO JE .NET CORE

- **Open source** projekt patriaci do skupiny .NET Foundation (dostupný na github-e https://github.com/dotnet/core)
- Pamäťovo a výpočtovo **menej náročný** „fork“ .NET Framework-u (.NET FX)
- Úplne **nová sada systémových knižníc** – CoreFX – nie všetky z .NET FX sú dostupné v .NET Core
- **Cross-platformový** framework (Runtime aj SDK) – Windows, Linux, Mac OS X
- Používa silne **modulárny prístup** – funkcionality sú často v samostatných NuGet balíčkoch
- Primárne je **určený pre tvorbu biznis logiky a back-endu** webových aplikácií
- Má dobrú škálovateľnosť a je ideálny na refaktoring projektov do podoby microservisov

ČO NIE JE .NET CORE

- Nie je náhrada za .NET FX – .NET Core sa vyvíja paralelne s .NET FX
- Nie je primárne určený pre Windowsové UI intenzívne aplikácie ako
  - WPF
  - Windows Forms
  - Universal Windows Platform
  - WCF služby
- Nie je všeliekom na zle napísaný projekt v .NET FX

AKÉ PROJEKTY MÔŽEM ROBIŤ S .NET CORE?

- Konzolové aplikácie
  - Workery a tasky
  - Viacvláknové aplikácie a pod.
- Webové aplikácie
  - SPA aplikácie – ľahký webserver s API a ľubovoľný FE (napr. Angular, React alebo Vue)
  - ASP.NET Core – alternatíva k ASP.NET WebForms
  - ASP.NET Core MVC – alternatíva k ASP.NET MVC
  - WebAPI cez *Microsoft.AspNetCore.Mvc* (ten istý controller pre MVC aj WebAPI) – alternatíva k WebAPI cez *System.Web.Http*
- Knižnice
  - ... .NET Standard ...

KEDY (NE)POUŽÍVAŤ .NET CORE

- Kedy áno
  - Ak je cieľom škálovateľná aplikácia (load balancing, mikroservis architektúra)
  - Ak je zmysluplné použitie viacvrstvovej architektúry
  - Ak je cieľom rýchla aplikácia
  - Ak plánujete nasadzovať aplikáciu na Linux web server (napr. apache, nginx)
  - Ak chcete používať kompilovaný FE s ľahkým REST API BE (napr. SPA)
- Kedy nie
  - Ak používate .NET Framework knižnice, ktoré nemajú implementáciu v .NET Standard alebo .NET Core
  - Ak používate 3rd party knižnicu / NuGet, ktorý nie je kompatibilný s .NET Core
  - Používanie komponentov, ktoré sú súčasťou OS Windows (napr. register, task scheduler, a pod.) a službu chcete hostovať aj mimo OS Windows
  - Ak je nevyhnutné vytvárať WCF služby
  - Ak používate distribúciu Linuxu, Windowsu alebo OS X, ktoré nie sú podporované
    - https://github.com/dotnet/core/blob/master/release-notes/2.0/2.0-supported-os.md

DÁ SA TO NEJAK OBABRAŤ? => CORE FX

- Obsahuje knižnice patriace do
  - základného balíka .NET Core
  - wrappery často používaných knižníc v .NET Frameworku
  - portnuté knižnice z .NET Frameworku do .NET Core
- Knižnice, ktoré CoreFX napriek tomu neponúka môžete ešte nájsť vo Windows Compatibility Pack
  - *System.Drawing.Common*
  - *System.IO.Packaging*
  - https://blogs.msdn.microsoft.com/dotnet/2017/11/16/announcing-the-windows-compatibility-pack-for-net-core/

Disclaimer:

Neodporúčam využívať tento pack, ak chcete cross-platform projekt. Nie všetky knižnice môžu bežať mimo OS Windows.

ZRELOSŤ .NET CORE – ČO UŽ JE A ČO EŠTE „CHÝBA“

- .NET Core je už relatívne zrelý projekt, novej funkcionality pribúda už v súčasnosti pomenej
- Balíky naviazané na .NET Core nie sú ešte až tak zrelé
- napr. Entity Framework Core (ORM)
  - Chýba(l) lazy loading – Pridaný v release EFCore 2.1
  - Database-first prístup ku generovaniu DB kontextu – len pomocou scaffoldovania existujúcej DB.
  - ADO.NET vizuálny model chýba a neplánuje sa jeho implementácia
  - LINQ GroupBy agregoval dáta v pamäti aplikácie (procesu aplikácie) – od EFCore 2.1 už generuje SQL Select
  - DB Pohľady (views) – Pridané až v release EFCore 2.1
  - https://blogs.msmvps.com/ricardoperes/2016/09/05/missing-features-in-entity-framework-core/

Odkaz: EF Core

ČO JE .NET STANDARD

- Vytvára class library projekt
- Reusable
  - Použiteľná v projektoch targetujúcich .NET Framework (min 4.5)
  - Použiteľná v projektoch targetujúcich .NET Core (min. 1.0)
  - Použiteľná pre .NET aplikácie v Linuxe, MacOSX aj Windowse (len niektoré verzie – podobne ako .NET Core)
- Ideálne použiteľná na biznis logiku v samostatnom projekte

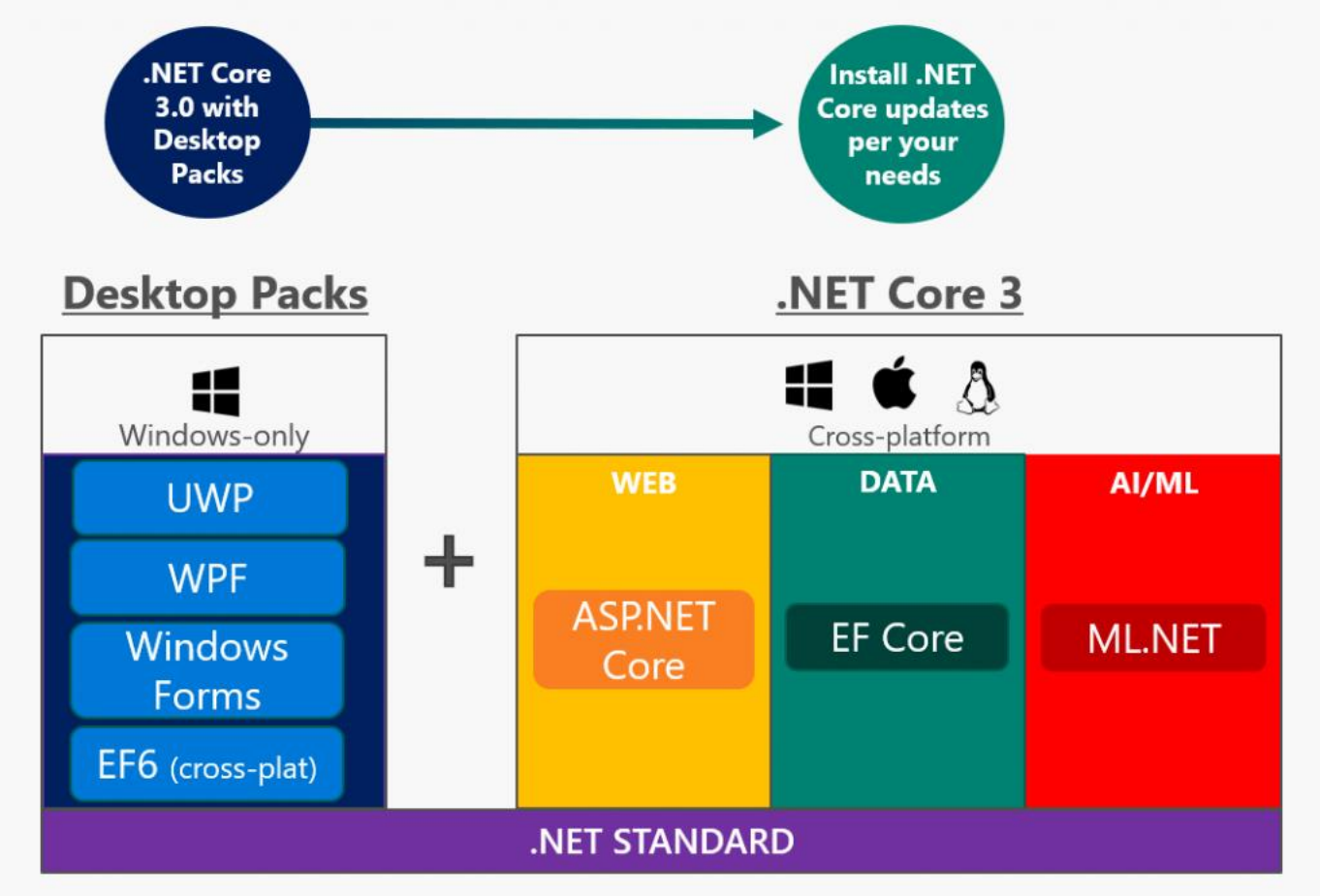
AKÚ VERZIU .NET STANDARD MÁM POUŽIŤ?

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework [1]	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299
Windows	8.0	8.0	8.1					
Windows Phone	8.1	8.1	8.1					
Windows Phone Silverlight	8.0							

http://immo.landwerth.net/netstandard-versions/#

ĎALŠIE VERZIE .NET CORE A .NET STANDARD

- .NET Core 3.0
- Q1 2019



https://blogs.msdn.microsoft.com/dotnet/2018/05/07/net-core-3-and-support-for-windows-desktop-applications/

ČASŤ II

Inštalácia, konfigurácia a prostredie

- Prerekvizity a inštalácia v OS Windows
- Prekrekvizity a inštalácia v OS Linux
- Editory Visual Studio 2017 a Visual Studio Code
- CLI príkazy pre .NET Core
- Práca s Visual Studio Code, nutné a vhodné rozšírenia
- Vytváranie jednoduchého projektu cez CLI
- Vytváranie solution-u cez CLI
- Štandardné súbory projektov .NET Core
- Verzia jazyka C#, podpora a nové funkcie v C# 7.1

INŠTALÁCIA A KONFIGURÁCIA - WINDOWS

- Čo je potrebné pre programovanie a spúšťanie programov?
  - .NET Core SDK https://www.microsoft.com/net/download/windows
  - .NET Core Runtime -//-
  - Obe sú súčasťou VS 2017 balíka „.NET Core Cross-platform development“
  - Použiteľná verzia .NET Core vo VS 2017 závisí od jeho verzie
    - .NET Core 2.1 => 15.7+
    - .NET Core 2.0 => 15.3+
    - .NET Core 1.x => 15.0+
  - Editor alebo IDE
    - Visual Studio 2017
    - Visual Studio Code
    - Iný editor
- https://docs.microsoft.com/en-us/dotnet/core/windows-prerequisites?tabs=netcore2x

KEDY KTORÝ EDITOR?

- Visual Studio 2017
  - Zložitejšie projekty a tímové projekty v TFS
  - Potreba profilovania a diagnostiky výkonnosti kódu
  - Viacprojektové aplikácie
  - Prioritou je BE
  - OS Windows
- Visual Studio Code
  - Jednoduchšie aplikácie
  - SPA aplikácie
  - REST API
  - Aplikácie s vlastným FE (React.JS, Angular, Vue, ...)
  - Prioritou je FE
  - OS Linux, OSX
- Iné ... zo zvyku

DISCLAIMER: Neodporúčam v jednom projekte kombinovať rôzne editory (napr. VS Code a Atom). Rôzne editory majú často rôzne formátovanie kódu

INŠTALÁCIA A KONFIGURÁCIA - WINDOWS

- Visual Studio 2017
  - Pre konzolové aplikácie a .NET Standard knižnice je nutný balík „.NET Core Cross-Platform development“
  - Pre webové aplikácie ASP.NET Core je nutný balík „ASP.NET and web development“
  - Pre podporu vývoja s DB serverom MS SQL je potrebná časť balíka „Data storage and processing “
- Overenie fungujúcej inštalácie
  - Command line alebo PowerShell
  - .NET Core CLI – príkaz dotnet
  - Visual Studio 2017 – Existujúci template na
    - Konzolovú aplikáciu .NET Core
    - .NET Standard class library

INŠTALÁCIA A KONFIGURÁCIA - LINUX

- Len vybrané distribúcie
  - Ubuntu & Mint
  - Debian
  - CentOS
  - Fedora
  - RHEL
  - SUSE & OpenSUSE
  - Alpine
- CLI príkazy a prerekvizity
  - https://www.microsoft.com/net/download/linux-package-manager/ubuntu18-04/sdk-current
  - https://docs.microsoft.com/en-us/dotnet/core/linux-prerequisites?tabs=netcore2x

.NET CORE CLI – ZÁKLADNÉ PRÍKAZY

- dotnet
- dotnet new
- dotnet run
- dotnet build
- dotnet test
- dotnet restore
- dotnet publish
- dotnet watch ... CLI HMR modul ... Je súčasťou .NET Core 2.1 SDK || Runtime

DEMO - .NET CORE CLI

- dotnet
- dotnet new <typ> [args | -n name ; -lang language ; -au authType]
- dotnet run
- dotnet build
- dotnet test
- dotnet restore – od 2.0 sa spustí automaticky ak je to potrebné (po builde, pri run atď.)
- dotnet publish
- dotnet watch ... CLI HMR modul ... Je súčasťou .NET Core 2.1 SDK || Runtime

DEMO - .NET CORE CLI

- Spravovanie referencií na iné projekty
  - *dotnet add reference – referencie na iné projekty*
  - *dotnet list reference <projekt>*
  - *dotnet remove reference*
- Spravovanie referencií na NuGet balíky
  - *dotnet add package*
  - *dotnet remove package*
- Možnosť pridávať vlastné a iné existujúce templates pomocou dotnet new –i <template>

DEMO – VYTVORENIE NOVÉHO PROJEKTU

1. Pridanie nového templatu do dotnet new, --install / -i napr. Vue.JS – použi google
2. Vytvorenie nového projektu podľa šablóny
    1. Argument pre názov projektu --name / -n
    2. Argument pre jazyk --language / -lang
    3. Argument pre autentifikáciu --auth

DEMO - VYTVORENIE SOLUTIONU S VIACERÝMI PROJEKTMI

1. Vytvorenie sln súboru [dotnet new sln –o <názov>]
2. Pridanie prvého projektu [dotnet new <typ> -n <názov> -o <output_adresár>]
3. Pridanie druhého projektu [dotnet new <typ> -n <názov> -o <output_adresár>]
4. Pridanie referencie medzi projektmi [dotnet add <kam> reference <čo>]
5. Pridanie projektov do solution [dotnet sln add <sln_súbor> <názov_projektu>]
6. Build / run projektu [dotnet watch --verbose run]

VISUAL STUDIO CODE

- Otvoriť adresár ako projekt
- Terminálové okno – alternatíva k otvorenému terminálu
- Možnosť mať otvorených paralelne viac terminálov
- Upraviteľný editor s množstvom rozšírení
- CTRL+SHIFT+P / F1 – vyhľadávanie príkazov
  - Terminal: Select Default Shell – Git Bash, Powershell, Command prompt
  - Extensions: Install Extensions
- Debugging
- "Solution explorer"
- Source control
- Ľahká integrácia projektov .NET Core a front-endu

VS CODE – MUST HAVE PACKAGES

- C#
- C# Extensions
- NuGet package manager – náhrada za Nuget vo VS 2017
- MS Build Project tools – Intellisense pre csproj súbor a odkazy na balíky
- ASP.NET Helper – atribúty pre taghelper <a> asp-controller/action/route-param
- vscode-icons – zlepší prehľadnosť súborov v projekte/solutione pridaním ikon k súborom
- Auto import (pre typescript FE)
- Docker – zvýrazňovanie syntaxe v docker a docker-compose súboroch
- Mssql – VS Code konektor na MSSQL databázy
- REST Client – volanie REST API z VS Code
- SQLite – explorer pre SQLite databázy

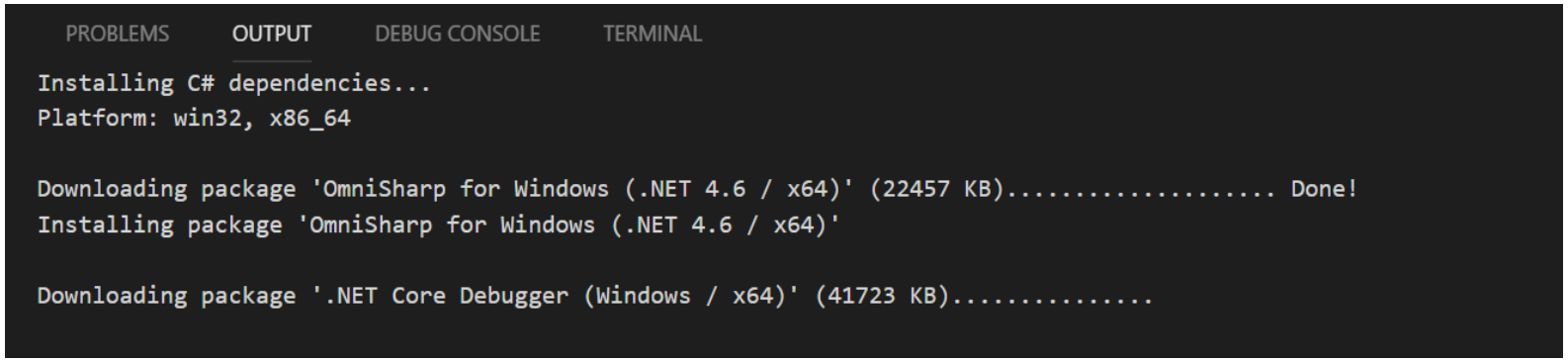
DEMO – SPUSTENIE PROJEKTU – VS CODE V DEBUG REŽIME

1. VS code po zistení, že sa jedná o .NET Core projekt nainštaluje nutné balíky
2. Súčasťou by mal byť aj .NET Core Debugger, ktorý dovolí spustiť aplikáciu v debug móde
3. CTRL+SHIFT+D – otvorí debug okno vo VS Code
4. Tlačidlo Debug spustí aktuálne otvorený projekt

Môžete pridať breakpointy podobne ako vo VS 2017 a krokovať svoj kód

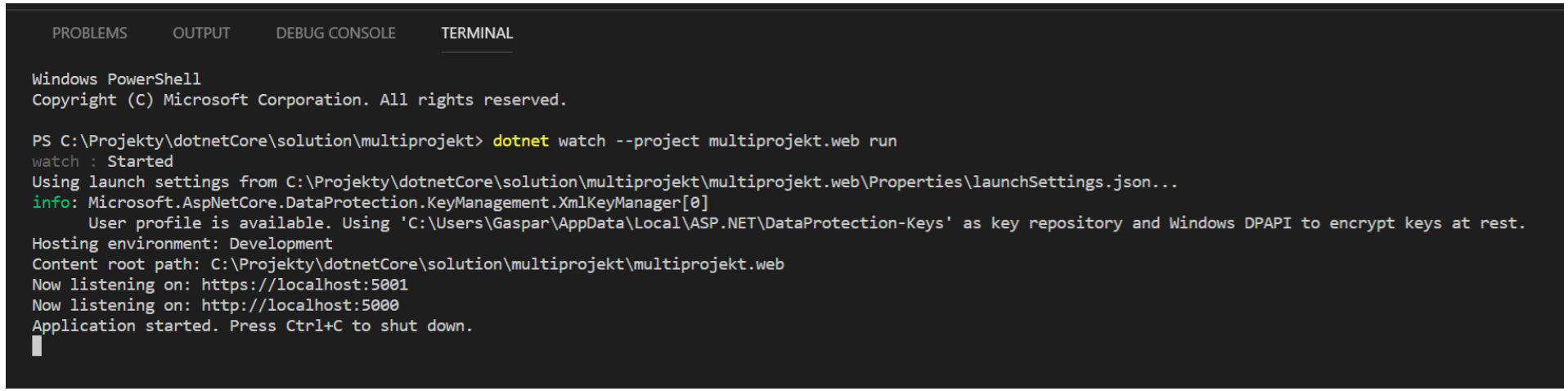
Watches fungujú

- ako príkazy v debug konzole VS Code (štandardne v spodnej časti)
- tradične pridané v okne debuggera (štandardne vľavo)



DEMO – SPUSTENIE PROJEKTU – VS CODE BEZ DEBUGU

- Z terminálu (bash, powershell, command prompt) vo VS Code
- dotnet watch --project <nazov> run
- Logy sa budu zaznamenávať do okna terminal
- Takto spustený kód nie je možné debugovať



ZÁKLADNÉ SÚBORY PROJEKTOV V .NET CORE

- Program.cs [Web, Konzola]
  - Obsahuje main funkciu /od v. C# 7.1 môže byť async/
  - V prípade webu obsahuje vytvorenie a nastavenie web servera
  - Odkazuje sa na konfiguráciu webu
- *.csproj [každý projekt]
  - Obsahuje základné info o projekte a verziu frameworku
  - Všetky referencie na NuGet balíky a iné projekty
  - Referencie na balíky, ktoré obsahujú doplnkové CLI príkazy
- Ostatné základné súbory už súvisia s ASP.NET Core (dostanem sa k tomu neskôr)

.NET CORE A VERZIE JAZYKA C#

- Nové verzie .NET Core podporujú nové minor verzie jazyka C#

Verzia C#	6	7.0	7.1	7.2	7.3
Visual Studio	2015	2017	2017 15.3	2017 15.5	2017 15.7
.NET Core SDK	1.0	1.0	2.0	2.0	2.1 RC1

.NET CORE 2.0 A C# 7.1

- Main metóda môže byť implementovaná ako **async**
- *public static async Task Main(string[] args)* vs *pubilc static void Main(string[] args)*
- Kompilátor s tým môže mať štandardne problém

```
Program.cs(15,29): error CS8107: Feature 'async main' is not available in C# 7.0. Please use languag
e version 7.1 or greater. [C:\Projekty\dotnetCore\solution\multiprojekt\multiprojekt.web\multiprojek
t.web.csproj]
CSC : error CS5001: Program does not contain a static 'Main' method suitable for an entry point [C:\
Projekty\dotnetCore\solution\multiprojekt\multiprojekt.web\multiprojekt.web.csproj]

The build failed. Please fix the build errors and run again.
```

- V *.csproj je možné zmeniť štandardnú verziu jazyka v tagu <*LangVersion*>

<*PropertyGroup*>

<*LangVersion*>*latest*</*LangVersion*>

</*PropertyGroup*>

- Ak používate async metódy, mali by obsahovať buď *Continuation pattern* alebo *await* asynchrónnych volaní

.NET CORE 2.0 A C# 7.1

- Používanie natívnych tuples funguje od C# 7.0 (predtým Tuple<>)
- V C# 7.1 sa názvy tuplov môžu automaticky odvodiť podľa názvu objektu(premennej) v tuple
- Pridané ďalšie možnosti inicializácie premenných pomocou kľúčového slova *default*

```
 int a= 1;
 int b= default;

 (int,int) tupleBezNazvov = (1,default);
 (int,int) tupleBezNazvovSilneTypovany = (a,b);
 (int a,int b) tupleSNazvamiSilneTypovany = (a,b);
 var tupleBezNazvovATypov = (1,0);
 var tupleSNazvami = (a,b);

Console.WriteLine($"Tuple bez odvodenia - prvok b: {tupleBezNazvov.Item2}");
Console.WriteLine($"Tuple bez odvodenia - silne typovany - prvok b: {tupleBezNazvovSilneTypovany.Item2}");
Console.WriteLine($"Tuple bez odvodenia - silne typovany - s nazvami - prvok b: {tupleSNazvamiSilneTypovany.Item2}");
Console.WriteLine($"Tuple s odvodenim - inferred types - prvok b: {tupleBezNazvovATypov.Item2}");
Console.WriteLine($"Tuple s odvodenim - prvok b: {tupleSNazvami.b}");
```

- Výsledok

```
Tuple bez odvodenia - prvok b: 0
Tuple bez odvodenia - silne typovany - prvok b: 0
Tuple bez odvodenia - silne typovany - s nazvami - prvok b: 0
Tuple s odvodenim - inferred types - prvok b: 0
Tuple s odvodenim - prvok b: 0
```

ČASŤ III

Projekty typu .NET Core .NET Standard

- Ukážka kompatibility v .NET Standard
- Služby - Dependency Injection v .NET Core
- Životný cyklus služieb

DEMO EX_1 – KOMPATIBILITA V .NET STANDARD

Kroky:

1. vytvorím si solution, ktorý bude obsahovať .NET Standard 1.6 class library projekt
2. napíšem jednoduchú knižnicu v .NET Standard 1.6
3. Vytvorím si .NET Framework 4.7.1 konzolový projekt, importnem knižnicu (projekt) a vyskúšam funkčnosť
4. Vytvorím si .NET Core 2.0 konzolový projekt, importnem knižnicu (projekt) a vyskúšam funkčnosť
5. Vytvorím si .NET Framework 3.5 konzolový projekt, importnem knižnicu (projekt) a vyskúšam funkčnosť

Predpoklady:

- Projekty v bodoch 3,4 pôjdu bez problémov skompilovať a aplikácia bude bežať bez problémov.
- Pri projekte v bode 5 importovanie knižnice bude fungovať ,ale kompilátor pri builde hodí chybu o nekompatibilnej verzii .NET FX.

- https://github.com/gasparv/netcore-class/tree/master/Ex_1_NetStandard

Poznámka: Je možné targetnúť aj viac frameworkov naraz

https://docs.microsoft.com/en-us/dotnet/standard/frameworks#supported-target-framework-versions

DEPENDENCY INJECTION ... KEDY, PREČO?

- DI by sa mala používať v prípade služieb, t.j. ak je väzba medzi triedami A,B typu A používa B - asociácia.
- DI vytvára slabú väzbu, t.j. ak aj objekt triedy A zanikne, objekt triedy B stále existuje. Podobne to platí aj naopak.
- DI sa používa z dôvodu lepšej správy pamäte Garbage collectorom a lepšej kontroly nad vznikom a zánikom objektov.
- Pri použití DI nemusia triedy služieb implementovať IDisposable.
- V ASP.NET Core MVC sa DI používa so vzorom typu Repository, napriek tomu, že medzi triedami nemusí nutne existovať asociácia.

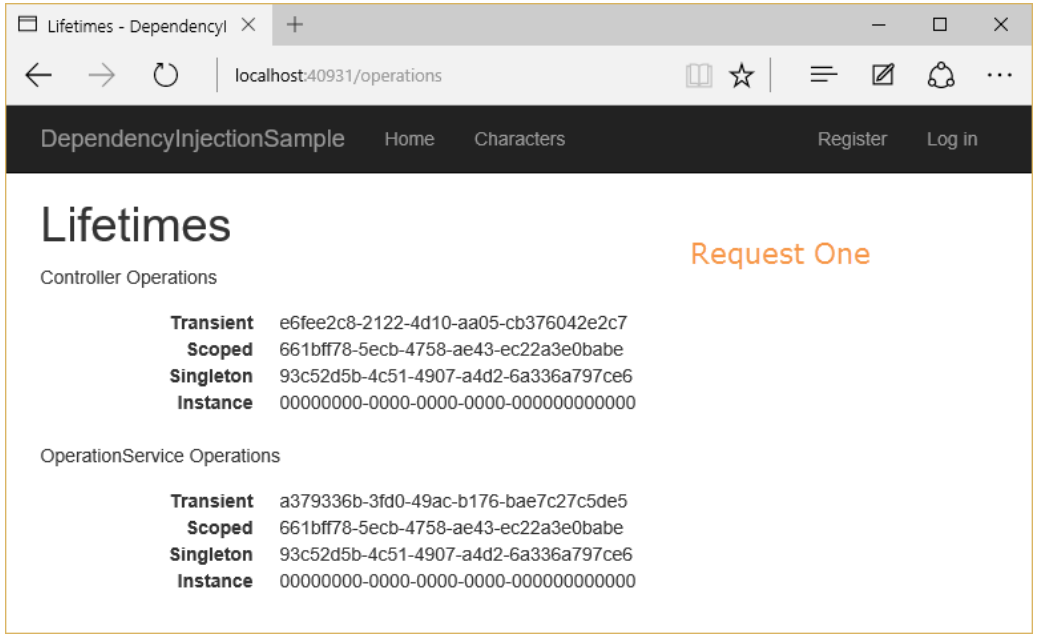
DEPENDENCY INJECTION V .NET CORE

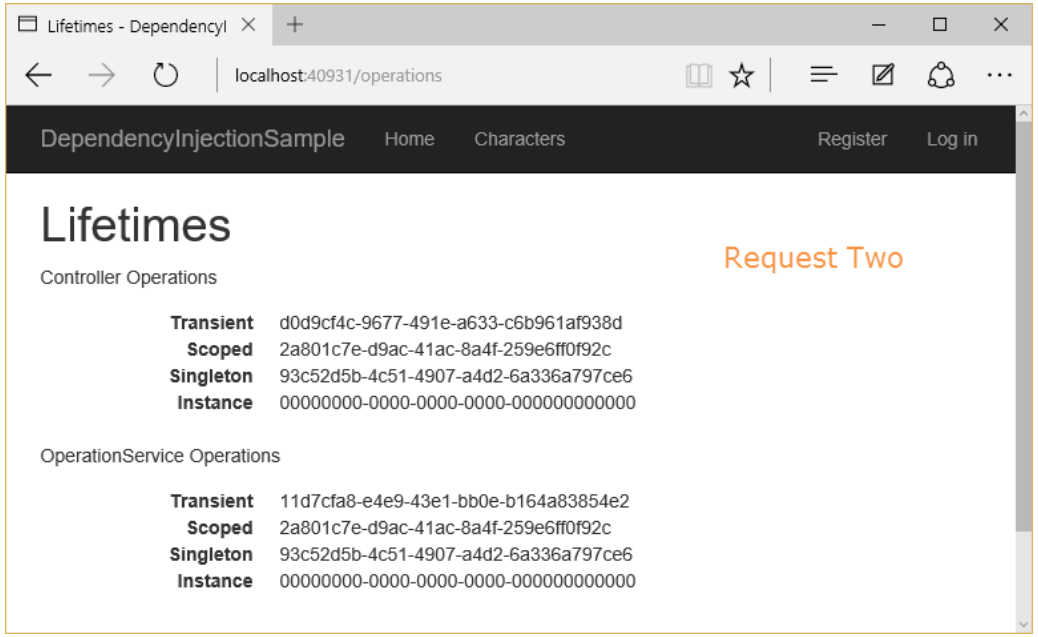
- Klasický projekt (Konzola alebo .NET Standard) používa trochu iný prístup ako ASP.NET Core
- Nutnosť nainštalovať do projektu NuGet balíček *Microsoft.Extensions.DependencyInjection*
- Balíček obsahuje aj *Microsoft.Extensions.DependencyInjection.Abstractions*
- Služby sa registrujú cez objekt typu *IServiceCollection* podľa *ServiceCollection*
- Služby sa volajú pomocou objektu, kt. je vytvorený metódou *BuildServiceProvider()* nad objektom typu *ServiceCollection*

- *ServiceCollection* neumožňuje registrovať viac implementácií jedného rozhrania plain-text kľúčom
  a) https://github.com/yuriy-nelipovich/DependencyInjection.Extensions NuGet Neleus.DependencyInjection.Extensions
  b) Factory pattern https://stackoverflow.com/a/44177920/3970830
  c) Použitie iného IoC kontajnéra

ŽIVOTNÝ ČAS SLUŽIEB

- Každá služba môže byť pridaná s rôznym životným časom
  - *Singleton* – jediný objekt (inštancia) počas celého behu programu – podľa návrhového vzoru Singleton
  - *Scoped* – pri jej volaní služba vygeneruje nový objekt.V rámci jedného requestu je aj pre rôzne kontexty použitý ten istý objekt
  - *Transient* – pri jej volaní služba vygeneruje nový objekt.V rámci jedného requestu je pre rôzne kontexty generovaný vždy nový objekt





DEMO EX_2_1 – DEPENDENCY INJECTION V .NET CORE(KONZOLA)

Kroky:

1. Vytvorím si konzolový projekt .NET Core 2.0 (A) a .NET Standard Class library (B) – v A pridám referenciu na B
2. V B do adresáru Services vytvorím implementáciu služby **PressoMachine** s metódou **MakeCoffee**()
3. V B do adresáru Models vytvorím implementáciu Zamestnanca (Employee) s metódou **MakeCoffee**()
4. V projekte A nainštalujem NuGet balíček „*Microsoft.Extensions.DependencyInjection*“ (+Abstractions)
5. V triede *Program.cs* – metóde *main* vytvorím objekt typu *IServiceCollection* konštruktorom triedy *ServiceCollection*
6. Vytvorím si nového zamestnanca použitím jeho štandardného konštruktora
7. V B v triede **Employee** použijem *constructor injection* pre použitie služby presostroja
8. Pomocou metódy .AddScoped<T>() zaregistrujem službu **PressoMachine**
9. Pomocou metódy .BuildServiceProvider() vytvorím poskytovateľa registrovaných služieb
10. V objekte zamestnanec zavolám metódu **MakeCoffee**().

DEMO EX_2_1 – DEPENDENCY INJECTION V .NET CORE

Predpoklady

- Zamestnanec si môže spraviť kávu pomocou presostroja
- Presostroj nie je vytvorený pomocou explicitného volania jeho konštruktora
- Presostroj je silne typovaná služba
- Ak sa presostroj nenájde, zamestnanec si kávu nespraví ...

Možný problém

- Firma má viac typov kávovarov, napr. Presostroj, Kávomat, DolceGusto (kapsulovač)
- Každý z nich vie uvariť kávu, akurát má iný postup, iný zdroj kávy a podobne (iná implementácia pre varenie kávy)

DEMO EX_2_2 – DI V .NET CORE - INTERFACE

Kroky (kroky 1-6 sú identické s EX_2_1):

7. V B vytvorím v adresári Definitions rozhranie **ICoffeeMaker**, ktoré obsahuje metódu void MakeCoffee();
8. Trieda **PressoMachine** nech implementuje rozhranie **ICoffeeMaker**.
9. Na obraz triedy **PressoMachine** vytvorím v adresári Services implementáciu triedy CapsuleCoffeemaker(), kt. tiež implementuje rozhranie **ICoffeeMaker**.
10. V triede **Employee** vymením **PressoMachine** za **ICoffeeMaker** pre *constructor injection*
11. V Program.cs zaregistrujem oba kávovary spolu s rozhraním, ktoré implementujú
12. V konštruktore zamestnanca vymením službu typu PressoMachine za vybranú službu, napr. podľa typu.

DEMO EX_2_2 – DI V .NET CORE - INTERFACE

Predpoklady:

- Vytvoríme lepšie možnosti abstrakcie pre implementáciu objektov s rovnakými verejnými funkciami
- Registrujeme rôzne služby a v rôznom kontexte ich môžu používať aj tie isté typy objektov
- Zlepší sa štruktúrovanie kódu
- Vieme použiť ktorýkoľvek kávovar vo firme jednoduchou zmenou pri volaní služby

ČASŤ IV

Projekty typu ASP.NET Core MVC

KONFIGURÁCIA SLUŽBY DI

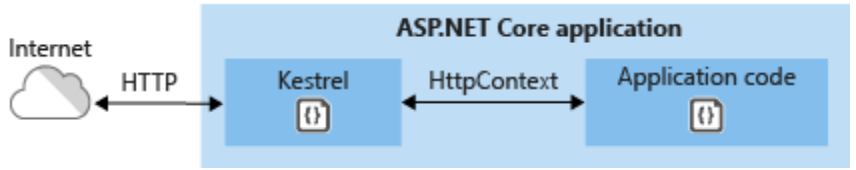
- Kestrel server a minimalistická konfigurácia
- Štruktúra projektu podľa templatu MVC
- Základná konfigurácia projektu
- Dependency Injection v ASP.NET Core
- Základné súbory a ich úloha
  - Konfiguračné položky
  - Nastavenie prostredí a ich vlastností
  - Startup trieda servera
- Štandardné služby v ASP.NET Core 2.0
- Prostredie nasadenia (HostingEnvironment)
- HTTP Request handling pomocou Middlewaru
- Bundling
- LEN PRE INFO – Prerekvizity na nasadenie na IIS

ĽAHKÝ WEBHOST SERVER - KESTREL

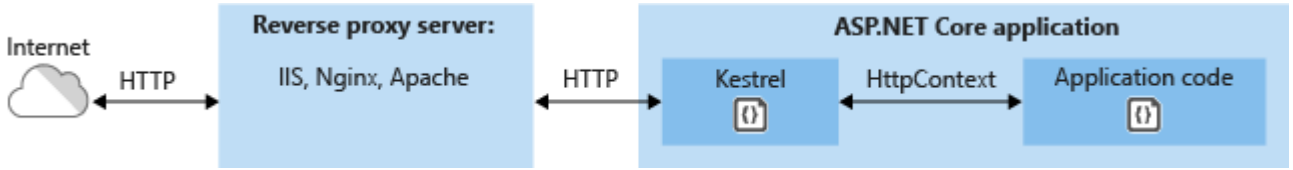
- .NET Core umožňuje použiť rovnako ľahký webserver ako napr. Node.Js
- Kestrel ako aj Node.JS vychádza z knižnice **libuv**
- Štart a konfigurácia je možná v Main funkcii
- Niektoré konvencie musia byť zachované ako v ASP.NET Core
- Nutné balíky
  - **Microsoft.AspNetCore**
  - **Microsoft.AspNetCore.Hosting**
  - **Microsoft.AspNetCore.Mvc** (len pre spustenie s middlewarom MVC)

ĽAHKÝ WEBHOST SERVER - KESTREL

- Typické scenáre použitia serveru Kestrel
- Kestrel server ako jednoduchý server pre odľahčené aplikácie



- Kestrel ako aplikačný server za komplexnejším web serverom (reverse proxy prístup)



DEMO EX_3 – KESTREL WEBHOST

Kroky:

1. Nainštalujem všetky potrebné NuGet balíky
   - **Microsoft.AspNetCore**
   - **Microsoft.AspNetCore.Hosting**
   - **Microsoft.AspNetCore.Mvc**
2. Vo funkcii main vytvorím webhost pomocou metódy **WebHost.Build()** a spustím metódou **Run()**
3. Vytvorím triedu Startup.cs a implementujem v nej metódy **Configure(IApplicationBuilder)** a **ConfigureServices(IServiceCollection)**
4. V metóde **ConfigureServices** zaregistrujem službu MVC pomocou metódu **.AddMvc**();
5. V metóde **Configure** pridám do pipelinu middleware MVC pomocou metódy .**UseMvc**();

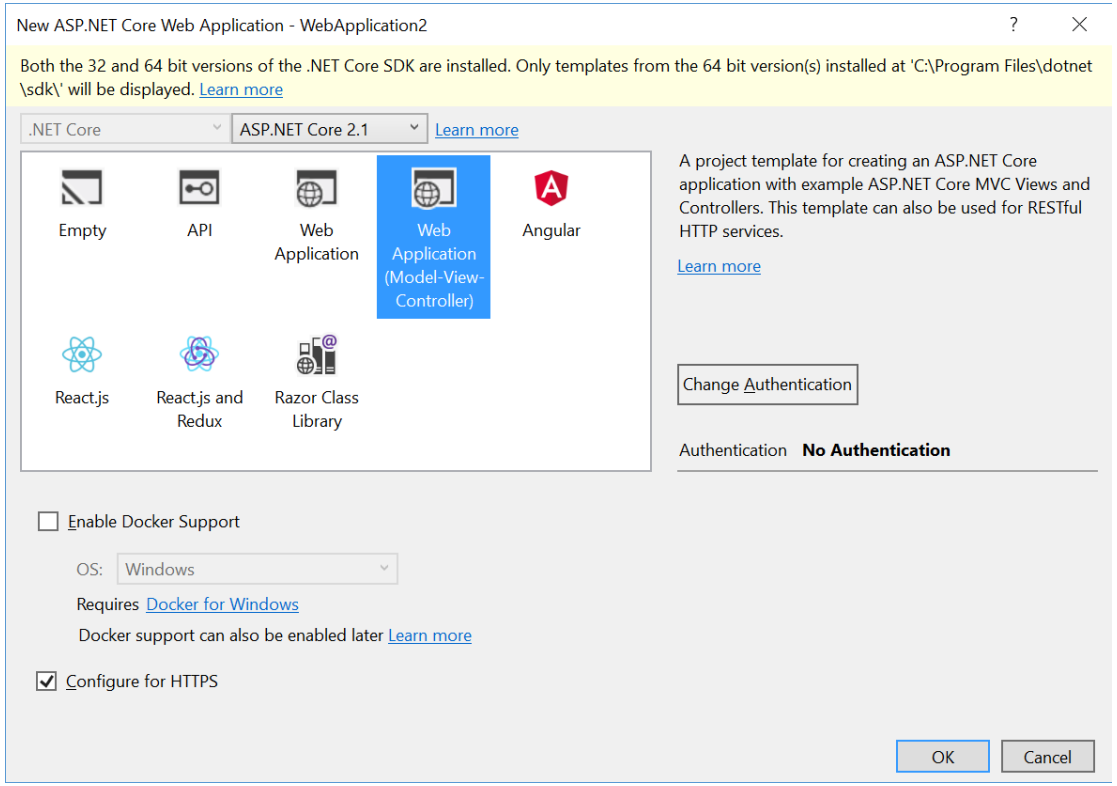
DEMO EX_3 – KESTREL WEBHOST

Predpoklady:

- Po spustení aplikácie sa otvorí konzolové okno
- Spustí sa proces-listener na HTTP requesty v konzolovom okne
- Listener počúva na porte 5000 a 5001 na localhost
- Logovanie servera je nastavené do konzoly

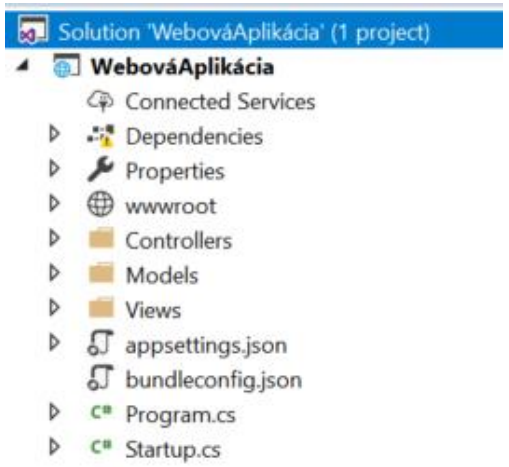
NOVÝ PROJEKT ASP.NET CORE

- Web / ASP.NET Core Web Application
- Alternatívy
  - Prázdny projekt – vlastná konfigurácia a architektúra
  - API – príprava pre API Routy a CRUD operácie
  - Web application – Razor Pages (alt. k webforms)
  - Web application MVC – MVC model aplikácie
  - Angular – API backend, Angular frontend
  - React.JS – API backend, React.JS frontend
  - React.js & redux – API backend, Redux store, React FE
  - Razor class library (.NET Core 2.1) – views a razorpages ako class lib
- Ďalšie možnosti
  - Verzia frameworku
  - Spôsob autentifikácie
  - Podpora nasadenia projektu ako Docker imagu
  - Defaultne nakonfigurovať server pre prístup cez HTTPS (.NET Core 2.1)



ŠTRUKTÚRA PROJEKTU PODĽA TEMPLATU MVC

- V projekte by mali byť úlohy kódu separované v samostatných adresároch
  - FE – hostuje ho adresár **wwwroot**
  - Services (BL) – adresár **Services** – v ňom sa zvyknú deliť definície od implementácií (Rozhrania služieb od tried)
  - ViewModels – adresár **Models -** modely určené ako zdroj dát pre Views
  - Data access – adresár **Data** – obsahuje DB kontext a dátové modely
  - Controller – adresár **Controllers** – obsahuje endpointy všetkých funkcií systému
- Každá časť je separovateľná ako samostatný projekt (jednoduchšie horizontálne škálovanie)
- Služby, Modely a Data access môžeš tvoriť ako .NET Standard class library projekt.

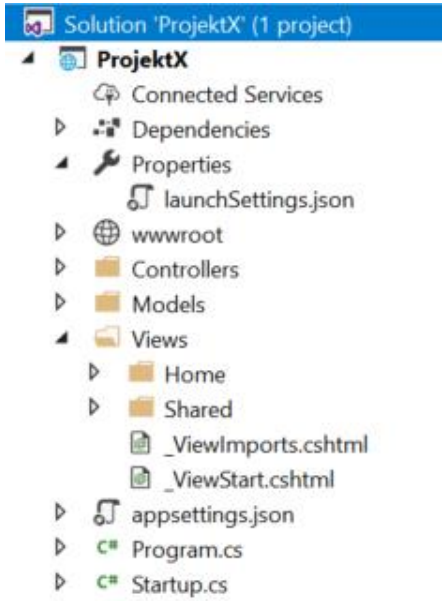


ŠTRUKTÚRA PROJEKTU PODĽA TEMPLATU MVC – POPIS SÚBOROV

- **appsettings.json** – vlastné konfiguračné nastavenia, spojenie s POCO triedami alebo XML s nastaveniami štýlom kľúč – hodnta. Odporúčané miesto pre pridávanie ConnectionStrings
- **Program.cs** – ako aj v konzolovom projekte obsahuje funkciu Main. V ASP.NET Core aj vytvorenie webhostu
- **Startup.cs** – obsahuje konfiguráciu prostredia, služieb, servera a middlewaru – nastavuje r-r pipeline
- **launchsettings.json** – v adresári properties. Obsahuje nastavenia pre spustenie webu v rôznych prostrediach.
- **_ViewImports.cshtml** – zdieľaný view pre globálny import knižníc a injectovanie služieb, kt. využívajú všetky views
- **bundleconfig.json (nepovinný)** – nastavenia pre bundling a minifying systém front-endu (wwwroot). Odporúčam používať komunitný NuGet „Bundler & Minifier“.

https://marketplace.visualstudio.com/items?itemName=MadsKristensen.BundlerMinifier

- **bower.json** (viacmenej deprecated/alternatíva je **package.json** z yarn) – Bower ešte stále má podporu vo VS 2017. do v.15.5. Od 15.8 (aktuálne preview) pribudne **Client-side library manager**.



KONFIGURÁCIA APLIKÁCIE

- **appsettings.json –** vlastné konfiguračné položky – napr. API kľúče, ConnectionStrings alebo čokoľvek iné
- Je unikátny pre každé prostredie – appsettings.{environment}.json
- Spôsoby správy konfigurácie
  - Objekt Configuration pridaný cez DI rozhranie IConfiguration
    - Problém, konzistentnosti názvov sekcií a kľúčov – plain text zápis aj vyhľadávanie bez literalu.
  - POCO triedy reflektujúce appsettings.json sekcie – pridaný do objektu Configuration pomocou DI – použiteľný ako IOption<T>
    - Problém, ak konfiguračná POCO trieda nie je zhodná s appsettings.json súborom. Výnimka vznikne až počas runtimu (keď sa nastavenie má použiť), nie pri spustení aplikácie.
    - Niektorí autori tento problém riešia vytvorením extension metódy nad IServiceCollection, ktorá triedu nastavení pridá ako Singleton cez DI.

    https://www.strathweb.com/2016/09/strongly-typed-configuration-in-asp-net-core-without-ioptionst/
  - ConfigurationProvider – JsonConfigurationProvider – **neodporúčaný spôsob**
- **Citlivé konfiguračné položky by nemali byť súčasťou appsettings.json**

a) Počas developmentu je možné použiť Secrets manager tool (služba AddUserSecrets)

b) V produkčnom prostredí sa odporúča buď**: Azure Key Vault** ; vlastný **ConfigurationProivder**, ktorý dáta šifruje ; **premenné prostredia**

DEMO EX_4_1 – NASTAVENIA APLIKÁCIE - IConfiguration

Kroky:

1. Vytvorím nový ASP.NET Core MVC projekt
2. Do súboru appsettings.json doplním sekciu „**WeatherApiSettings**“ do ktorej pridám dva parametre:
    1. „WeatherApiVersion“:1
    2. „WeatherApiKey“:“$+r@$’n3_+4jnY_k!’u[“
3. (Nepovinné) V Startup.cs môžem pozrieť ako vyzerá injectnutie konfigurácie
4. V HomeController.cs injectnem IConfiguration metódou *constructor injection*
5. Pomocou metód **GetSection**() a **GetValue**() objektu **Configuration** získam napr. **WeatherApiKey**
6. Pridám si breakpoint na miesto, kde som žiadal o konfiguračnú položku

DEMO EX_4_1 – NASTAVENIA APLIKÁCIE - IConfiguration

Predpoklady:

- Konfiguračnú položku mám načítanú v premennej po jej vytiahnutí cez **IConfiguration**
- Pre identifikáciu sekcie a položky musím v metódach **GetSection**() a **GetValue**() použiť plain text ako vstup
- Typ položky nie je automaticky odvodený – metóda **GetValue**() vráti typ **object**
- K hodnote konfiguračnej položky sa nedostanem bez použitia metódy **GetSection()**
- Konfiguračné položky sú ukladané mimo kód aplikácie, takže nemusím rekompilovať kód
- Alternatíva je vyhľadanie podľa kľúča [nazovSekcie:nazovPolozky]

DEMO EX_4_2 – NASTAVENIA APLIKÁCIE – IOption<>

Kroky:

1. Vytvorím ASP.NET Core MVC projekt
2. Vytvorím POCO triedu „**WeatherApiSettings**“, ktorá bude obsahovať prázdny public konštruktor
3. Vytvorím dve public properties s getterom a setterom. Môžem im prednastaviť hodnotu.
   - WeatherApiVersion
   - WeatherApiKey -> $+r@$’n3_+4jnY_k!’u[
4. V súbore Startup.cs, metóde **ConfigureServices** zaregistrujem použitie **IOption** konfigurácie pomocou metódy **AddOptions**() nad objektom **services**
5. V súbore Startup.cs, metóde **ConfigureServices** zaregistrujem konfiguračnú triedu pomocou metódy **Configure<>** nad objektom **services**
6. V HomeController.cs injectnem **IOption<WeatherApiSettings>** a hodnotu položky získam pomocou: **_option.Value.WeatherApiKey**

DEMO EX_4_2 – NASTAVENIA APLIKÁCIE – IOption<>

Predpoklady:

- Pre získanie konfiguračnej položky použijem literál namiesto jej plain text názvu
- Pre získanie konfiguračnej položky zo sekcie nepotrebujem názov sekcie – premietne sa do názvu triedy
- Typ konfiguračnej položky je odvodený podľa typu, kt. je definovaný v príslušnej property v POCO triede
- Preklep v názve sekcie alebo konfiguračnej položky nespôsobí pád aplikácie počas runtimu ale pri kompilácii
- IOption prístup mi dovolí separovať veľký .json konfiguračný súbor na viac kontextovo špecifických POCO tried
- Pre nastavenie hodnoty konfiguračnej položky pri spustení aplikácie môžem použiť delegáta pri registrácii konfigurácie v metóde **ConfigureServices**().

Nevýhody:

- Zmeny konfiguračnej položky nie sú permanentné (ukladané)
- Konfiguračné položky sú ukladané v kóde aplikácie (nemusia byť) preto pri zmene nastavení je nutná rekompilácia

KONFIGURÁCIA – ZDROJE A PRIORITY

- Každý zdroj konfiguračných položiek ma inú prioritu
  - Najnižšia priorita – appsettings.json
  - Stredná priorita – položka definovaná v POCO triede s priradenou hodnotou
  - Najvyššia priorita – položka priradená pomocou delegáta
- Rôzne zdroje/poskytovatelia konfiguračných položiek
  - *.ini, *.json, *.xml súbory
  - In-memory .NET Objekty – využíva binding
  - Premenné prostredia (Env variables)
  - Argumenty v príkazovom riadku – vstupujú ako args parameter do metódy Main
  - EFCore ako zdroj konfiguračných dát
  - Iné – napr. Secrets store, key vault a podobne

Disclaimer: Odporúča sa použiť kombinácia appsettings.json a POCO tried injectnutých ako IOption<>. V POCO triedach sa neodporúča nastavovať defaultné hodnoty pre konfiguračné položky, aby sa zabránilo nutnosti rekompilácie v prípade zmien konfigurácie. V prípade, že položka neexistuje má byť vyvolanie konfiguračnej položky ošetrené buď pomocou try/catch bloku alebo null check.

DEPENDENCY INJECTION V ASP.NET CORE - SLUŽBY

- DI používa iný vzor pre registráciu služieb
- Všetky služby (systémové aj vlastné) by mali byť registrované v metóde **ConfigureServices** v triede **Startup**
- Služby sa pridávajú štandardne v objekte **services** injectnutého rozhrania **IServiceCollection** metódami
  - **.AddTransient**
  - **.AddScoped**
  - **.AddSingleton**
- Pre získanie služby sa nepoužíva ServiceProvider ale priamo injectovanie pomocou
  - Konštruktora – služba je injectnutá ako parameter konštruktora
  - Akcie – služba je injectnutá ako parameter metódy kontrolera s atribútom **[FromServices]**
  - Property – služba je injectnutá ako property cieľového objektu
  - Metódy – služba je injectnutá ako vstup metódy pri jej volaní nad objektom
- ServiceProvider sa dá použiť, ak mám viac implementácií (class) pre jednu definíciu (interface) a potrebujem nájsť konkrétnu implementáciu

EXISTUJÚCE SLUŽBY V ASP.NET CORE

- Je ich veľa – uvediem len tie základné
- AddAntiforgery – pri requestoch s Antiforgery tokenom (AFT) slúži ako override pre správu AFT atribútov v hlavičke alebo cookie.
- AddAuthentication – pridá služby pre autentifikáciu s možnosťou zvoliť providera (cookie, JWT Bearer token, facebook, google, microsoft, twitter, windows, atd.).
- AddIdentity – pridá služby pre identity framework (**IdentityUser**, **IdentityRole**). Nutný middleware UseAuthentication
- AddCors – pridá služby pre správu cross-origin requestov (nutný middleware UseCors)
- AddDbContext – pridá službu pre správu databázového kontextu. Dá sa špecifikovať aj trieda s kontextom ako generický parameter a DB provider cez delegáta.
- AddMvc – pridá služby nutné pre fungovanie MVC frameworku a zaregistruje konvencie v MVC.
- AddLocalization – pridá lokalizačné služby
- AddLogging – pridá logovacie služby. Logovanie sa dá spustiť aj v Program.cs cez fluent api príkaz **ConfigureLogging** pri buildovaní webhosta a následné nastavenie pomocou delegáta

LOGOVANIE

Krátka ukážka ...

- Vstavaní provideri pre logovanie
- Doplnkové možnosti pre logovanie

DEMO EX_5 – DEPENDENCY INJECTION – SPÔSOBY

Kroky:

1. Vytvorím si ASP.NET Core MVC projekt
2. Do adresáru Services/Definitions pridám public rozhranie **ICoffeeMachine** s metódou void **MakeCoffee**()
3. Do adresáru Services/Implementations pridám public triedy **CapsuleCoffemaker**, **PressoMachine**, **CoffeeKettle**, kt. implementujú **ICoffeeMachine**. Metódam **MakeCoffee**() pridám funkcionalitu.
4. Všetky 3 služby zaregistrujem v Startup.cs ako Scoped
5. V adresári Models vytvorím triedu **Employee** a public metódu **MakeCoffee**()
6. V adresári Models vytvorím triedu **Guest** a public metódu **MakeCoffee**()
7. V adresári Models vytvorím triedu **Owner** a public metódu **MakeCoffee**()
8. Pre bod 5 injectnem **ICoffeeMachine** metódou ***constructor injection***
9. Pre bod 6 injectnem **ICoffeeMachine** metódou ***property injection***
10. Pre bod 7 injectnem **ICoffeeMachine** metódou ***method injection***
11. V **HomeController** injectnem **IServiceCollection**, vytvorím objekty osôb a priradím im jeden typ kávovaru

Pozn.: Pre výpis do Output okna použijem príkaz Debug.WriteLine()

PROSTREDIE NASADENIA – CARE FOR THE ENVIRONMENT ☺

- 3 základné prostredia nasadenia
  - Development
  - Staging
  - Production
- Rôzne konfigurácie pre rôzne prostredia nasadenia – rôzny handling pre http requesty, rôzne FE súbory, a.i.
- Štandardne je to injectnuté rozhranie **IHostingEnvironment** ako parameter metódy Configure (*method injection*)
- Nastavuje ho premenná prostredia ASPNETCORE_ENVIRONMENT
- Profily spustenia aplikácie sú definované v súbore launchSettings.json v časti Properties

PROSTREDIE NASADENIA – ŠTRUKTÚRA LAUNCHSETTINGS.JSON

- Sekcia profiles obsahuje rôzne profily spustenia aplikácie v rôznych prostrediach nasadenia
- Atribút commandName môže obsahovať 3 rôzne príkazy a špecifikuje aký server má byť aplikáciu spustiť
  - Project – profil je platný pri spustení pomocou príkazu dotnet – spustí sa Kestrel server
  - IIS Express – profil je platný vo Visual Studio 2017 – spustí sa IIS Express, ktorý na pozadí spustí Kestrel (ak je nastavený)
  - IIS – profil je platný v aplikácii nasadenej na IIS – aplikácia spustená v v Application poole na IIS (Env var je možné zadať v app poole v iis)
- Atribút launchBrowser môže obsahovať true/false a určuje či sa má otvoriť prehliadač po spustení aplikácie
- Atribút applicationUrl môže definovať viacero url adries, na ktorých bude aplikácia prístupná v danom profile
- Atribút environmentVariables obsahuje parametre prostredia, vrátane ASPNETCORE_ENVIRONMENT

PROSTREDIE NASADENIA – POUŽITIE

- V startup.cs je možné použiť metódy pre zistenie aktuálne nastaveného prostredia
  - IsDevelopment()
  - IsStaging()
  - IsProduction()
  - IsEnvironment(string nazov)
- Startup.cs povoľuje aj špecifikovať konfiguračné metódy s názvom prostredia
  - Configure{environment}
  - Configure{environment}Services

Tento prístup je konvencia, t.j. metódy s názvom prostredia sa spustia len pri špecifikovaní príslušnej premennej prostredia

Vlastné prostredie je možné dodefinovať kdekoľvek jeho použitím (napr. metódou Configure{env})

EX_6 – HOSTING ENVIRONMENT

Kroky:

1. Vytvorím nový ASP.NET Core MVC projekt
2. V **launchSettings.json** si vytvorím nový profil s názvom *„ProductionProfile“, commandName: Project, launchBrowser: true*. Do **environmentVariables** pridám *ASPNETCORE_ENVIRONMENT* s hodnotou *„Production“*. Posledný parameter je **applicationUrl**, pre ktorú špecifikujem iný port.
3. V Startup.cs vytvorím metódu **ConfigureProduction**, v ktorej vynechám middleware, ktorý sa nehodí pre produkčné prostredie nasadenia aplikácie (Developer exceptions a browserlink)
4. V metódach **Configure** a **ConfigureProduction** pridám overenie prostredia pomocou metód **env.IsDevelopment**, resp. **env.IsProduction**.
5. Zároveň si v týchto metódach pridám výpis do konzoly (ak spúšťam cez CLI).
6. V **HomeController**, v metóde Index hodím exception aby bolo jasné či middleware **UseDeveloperExceptions** bol použitý alebo nie.
7. Pre lepšie odsledovanie môžem pridať breakpointy do metód **Configure** a **ConfigureProduction** v *Startup.cs*

EX_6 – HOSTING ENVIRONMENT

Predpoklady:

- Po spustení s profilom **ProductionProfile** cez CLI je vo výpise zobrazené prostredie Production
- Po načítaní stránky s profilom **ProductionProfile** sa zobrazí štandardná chybová stránka /Home/Error bez interných popisov chýb (stack trace stránka)
- Po spustení s profilom **Ex_6_HostingEnvironment** cez CLI je vo výpise zobrazené prostredie Development
- Po načítaní stránky s profilom **Ex_6_HostingEnvironment** sa zobrazí chybová stránka s interným popisom chyby (stack trace stránka)

EX_6 – HOSTING ENVIRONMENT – NA STRANE VIEWS

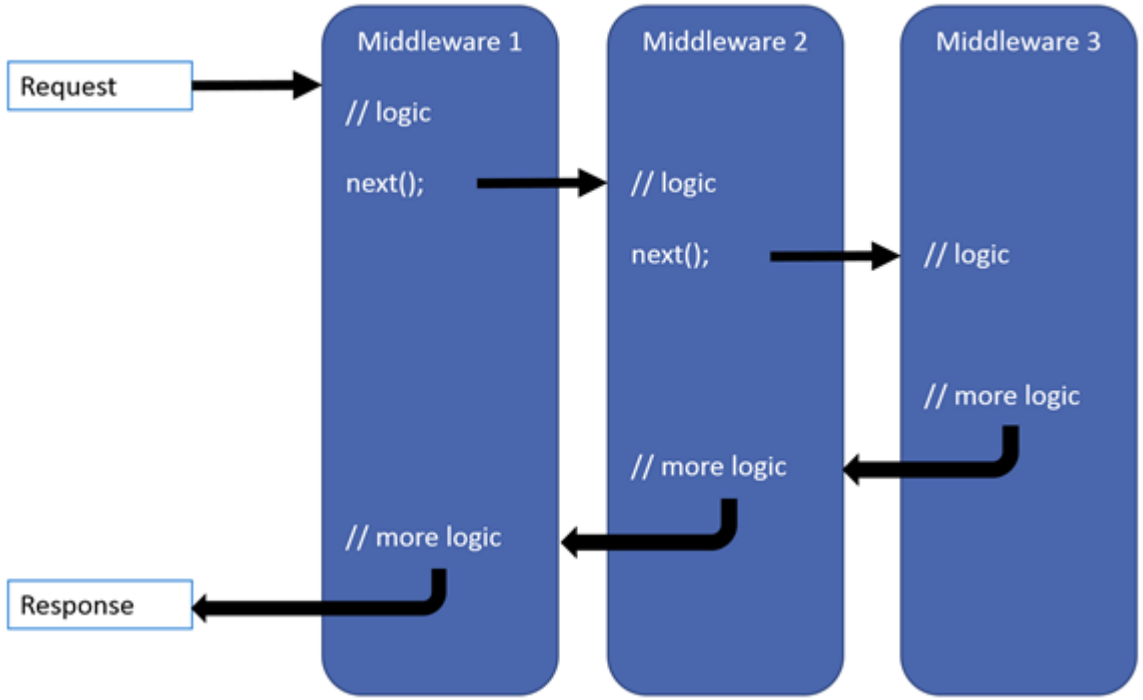
Poznámka:

- Všimnite si napr. v Views/Shared/_Layout.cshtml tagy <environment>, ktoré sú na začiatku súboru
- Definujú čo sa má (include) alebo nemá (exclude) zobraziť ak je nastavené dané prostredie
- Štandardné je použitie pre
  - Import minifikovaných a bundlovaných štýlov a skriptov alebo CDN zdrojov – Production
  - Zobrazovanie informácií vo view len pre potreby overenia fungovania počas developmentu
- Úloha – vráťte **HomeController** do pôvodného stavu a vyskúšajte si v súbore **_Layout** ako funguje environment na titulke stránky

SPRACOVANIE POŽIADAVIEK (HTTP REQUEST HANDLING)

- Spracovanie requestov má na starosti **IApplicationBuilder**, kt. definuje pipeline, cez ktorý musí každý request prejsť
- Pre HTTP requesty sa v súčastnosti používa **WebHost** (viď. Program.cs). Aktuálne sa ale pracuje na jeho náhrade všeobecnejším generickým hostom (preview v ASP.NET Core 2.1)
- V triede Startup.cs, v metóde **Configure** sa nastavuje postupnosť metód, cez ktoré request prechádza
- **IApplicationBuilder** je injectnutý do metódy **Configure** pomocou *method injection*.
- Volané metódy, cez ktoré každý request prechádza sa nazývajú HTTP Middleware
- Poradie v akom request prechádza cez pipeline je dôležitý (poradie metód v **Configure**)
- Request končí v middlewari vtedy, ak už nie je postúpený v pipeline ďalšiemu middlewaru (terminating middleware)
- Ukončovacie podmienky môžu byť rôzne (v závislosti od middlewaru), napr. problém s autentifikáciou, autorizáciou, dosiahnutie správnej cesty, získanie statického súboru, a.i.

HTTP MIDDLEWARE - PRINCÍP



https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.1&tabs=aspnetcore2x

HTTP MIDDLEWARE – POUŽITIE

- Middleware je možné pridať do pipelinu
  - Priamo (in-line metóda) použitím inline funkcie a delegáta s parametrami **(context, next)**
  - Ak je definovaný ako samostatná trieda, pomocou extension metódy **UseMiddleware<>**
  - Ak je definovaný ako samostatná trieda, pomocou vlastnej extension metódu
- 3 typy metód pre in-line middleware (priamo definovaný pomocou delegátov v metóde **Configure**)
  - Use – middleware, ktorý môže ukončiť pipeline alebo zavolať ďalší middleware pomocou metódy **next.Invoke();**
  - Run – ukončujúci middware (nesmie volať next)
  - Map –zavolá middleware, ak request príde na mapovanú cestu (URL). Ukončuje pipeline (nesmie volať **next.Invoke()**)
  - MapWhen – zavolá middleware, v prípade, že platí predikát (podmienka) na zavolanie middlewaru. Ukončuje pipeline (nesmie volať **next.Invoke()**)
  - UseWhen – middleware, ktorý môže ukončiť pipeline alebo zavolať ďalší middleware, ak je splnený predikát.

HTTP MIDDLEWARE – MIDDLEWARE AKO TRIEDA

- Pravidlá:
  - Trieda Middlewaru musí byť public
  - Služba RequestDelegate musí byť injectnutá metódou *constructor injection*
  - Trieda Middlewaru musí obsahovať metódu Invoke (resp. **InvokeAsync**) so vstupom typu **HttpContext**
  - Metóda Invoke (resp. **InvokeAsync**) musí obsahovať ukončenie pipelinu alebo zavolať ďalší middleware v rade pomocou návratu objektu **RequestDelegate** so vstupom typu **HttpContext**
  - Po zavolaní ďalšieho middlewaru môžu byť vykonávané funkcie, ale už sa nesmie zasahovať do responsu (vyvolá výnimku).
  - Middleware definovaný v triede je pridaný v metóde **Configure** metódou .**UseMiddleware<T>**()
  - Alternatívne je možné pre nový middleware vytvoriť rozširujúcu metódu (extension method) nad **IApplicationBuilder** a použiť ju v metóde **Configure**

EX_7_1 – ZÁKLADNÁ KONFIGURÁCIA MIDDLEWARU

Kroky:

1. Vytvorím nový ASP.NET Core MVC projekt
2. V **launchSettings.json** nakonfigurujem nový profil **ProductionProfile** s *commandName: Project, launchBrowser:true,* a **ASPNETCORE_ENVIRONMENT**: „*Production*“. URL nastavím na port 12556
3. V Startup.cs prepíšem názov metódy **Configure** na **ConfigureDevelopment**, aby sa middleware navzájom neovplyvňoval v rôznych prostrediach
4. Vytvorím 3 privátne statické void metódy, ktorý vstupom je **IApplicationBuilder**
    1. **FirstMiddlewareDelegate** – zapíše cez .Use text do responsu aj do konzoly. Nevolá next.Invoke().
    2. **SecondMiddlewareDelegate** – zapíše cez .Use text do responsu aj do konzoly. Volá next.Invoke().
    3. **ThirdMiddlewareDelegate** – zapíše cez .Use text do responsu aj do konzoly. Nevolá next.Invoke().
5. Vytvorím metódu **ConfigureProduction** so vstupmi **IApplicationBuilder** a **IHostingEnvironment**

EX_7_1 – ZÁKLADNÁ KONFIGURÁCIA MIDDLEWARU - POKRAČOVANIE

6. Zaregistujem middleware v tomto poradí:
    1. FirstMiddleware – metódou .Map na path „/alpha“
    2. SecondMiddleware – metódou UseWhen s podmienkou, ak query obsahuje parameter „beta“
    3. ThirdMiddleware – metódou MapWhen s podmienkou, ak query obsahuje parameter „gamma“
    4. Inline middleware – metódou .Run, ktorá zapíše do responsu text „Production environment has not yet been set.“

EX_7_1 – ZÁKLADNÁ KONFIGURÁCIA MIDDLEWARU

Výsledky kombinácií:

- localhost:12556 = Production environment has not yet been set.
- localhost:12556/alpha = First middleware is writing the response
- localhost:12556/alpha?beta = First middleware is writing the response
- localhost:12556/alpha?gamma = First middleware is writing the response
- localhost:12556/alpha?beta&gamma = First middleware is writing the response
- localhost:12556?beta = Second middleware is writing the response. Production Environment has not yet been set.
- localhost:12556?gamma = Third middleware is writing the response
- localhost:12556?beta&gamma = Second middleware is writing the response. Third middleware is writing the response.

EX_7_1 – ZÁKLADNÁ KONFIGURÁCIA MIDDLEWARU

Predpoklady:

- Vždy, ak sa použije v URL /alpha, tak je pipeline ukončený FIRST MIDDLEWARE lebo používa metódu MAP a je prvý v pipeline
- Ak sa použijú query parametre beta a gamma súčasne, pipeline začína SECOND MIDDLEWARE lebo je prvý v poradí a jeho predikát je splnený. Pipeline ukončuje THIRD MIDDLEWARE lebo jeho predikát je splnený a používa metódu MAP.
- Ak je použitý len query parameter beta, pipeline začne SECOND MIDDLEWARE lebo je splnený jeho predikát. Pipeline ukončuje anonymný middleware app.Run pretože SECOND MIDDLEWARE obsahuje vyvolanie ďalšieho middlewaru pomocou delegáta. Keďže predikát THIRD MIDDLEWARE nie je splnený, zavolá sa až **app.Run**, ktorý je terminálny.
- Ak je použitý len query parameter gamma, pipeline začne THIRD MIDDLEWARE lebo je splnený jeho predikát. Keďže je volaný metódou **MapWhen**, tento middleware je terminálny a nemôže volať ďalší middleware.

EX_7_2 – MIDDLEWARE AKO VLASTNÁ TRIEDA

Kroky:

1. Vytvorím ASP.NET Core MVC projekt
2. Do adresáru Middleware pridám public triedu s názvom „**IPFilteringMiddleware**“
3. Metódou *constructor injection* injectnem RequestDelegate (delegát na middleware)
4. Vytvorím metódu **Invoke** (od .NET Core 2.1 funguje aj **InvokeAsync**) so vstupom **HttpContext** a výstupom Task
5. Do metódy **Invoke** môžem injectnuť služby podľa potreby spôsobom *method injection*
6. V metóde **Invoke** vytvorím podmienku, ktorej splnenie buď vráti zápis do responsu alebo delegáta na ďalší middleware
7. ALT1: V triede Startup.cs v metóde **Configure** pridám do pipelinu middleware „**app.UseMiddleware<IPFilteringMiddleware>();**“
8. ALT2: Vytvorím si extension metódu, ktorá bude slúžiť pre pridanie tohto konkrétneho middlewaru (napr. **UseIPFiltering**) – public static metóda vracajúca **IApplicationBuilder**. V Startup.cs ju použijem ako „**app.UseIPFiltering()**“

EX_7_2 – MIDDLEWARE AKO VLASTNÁ TRIEDA

Predpoklady:

- Na stránku sa dostanú len klienti s povolenými segmentmi
- Správanie pre development a production prostredie bude rôzne
- Extension metóda zabalí štandardnú funkciu .**UseMiddleware**
- Pipeline je ukončený buď hláškou, kt. vygeneruje **IPFilteringMiddleware** alebo Mvc middlewarom

Doplnenie:

- Middleware môže implementovať rozhranie **IMiddleware** (tzv. Factory middleware pattern).
- Výhoda: Takto vytvorený middleware je možné registrovať ako službu – middleware sa generuje pre každý request napr.
- Nevýhoda: Nie je povolené pridávať parametre v extension metóde do **UseMiddleware** – NotSupportedException
- MiddlewareFactory je len proxy trieda na IServiceProvider
  https://github.com/aspnet/HttpAbstractions/blob/release/2.0/src/Microsoft.AspNetCore.Http/MiddlewareFactory.cs
- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/extensibility-third-party-container?view=aspnetcore-2.1

ASP.NET CORE MIDDLEWARE

- Existuje pomerne veľa middleware na strane ASP.NET Core – spomeniem tie najčastejšie
  - UseStaticFiles – štandardne nalinkuje ako statický obsah z wwwroot. Dá sa použiť viackrát pre viac adresárov.
  - UseDefaultFiles – povoľuje volať inicializačné súbory mimo routing systému (napr. index.html)
  - UseDirectoryBrowser – povoľuje prehliadanie obsahu adresárov
  - UseDeveloperExceptionPage – zobrazuje rozšírené info o výnimkách
  - UseExceptionHandler – middleware pre správu výnimiek v produkčnom prostredí – väčšinou redirect
- Veľa middlewarov závisí od použitia určitej služby
  - UseAuthentication (AddAuthentication)
  - UseCors (AddCors)
  - ...

BUNDLING A MINIFIKÁCIA

- Bundling – spojenie viacerých zdrojových súborov (skriptov a štýlov) do jedného súboru
- Minifikácia – techniky predspracovania zdrojových súborov (skriptov a štýlov), ktorá minimalizuje ich veľkosť
- MVC štandardne realizuje minifikáciu a bundling ako task pri builde podľa konfigurácie v súbore **bundleconfig.json**
- Konvencia je používať jeden bundle (výsledný súbor) na skripty, jeden na štýly
- Vstupné súbory je možné uvádzať samostatne alebo pomocou globbing vzoru https://gruntjs.com/configuring-tasks#globbing-patterns
- Bundling do projektu je možné pridať ako NuGet komunitný balíček **BuildBundlerMinifier** a spúšťať ho
  - Pri builde ako task – ak sa používa NuGet balík **BuildBundlerMinifier**
  - Manuálne volaním CLI príkazu „dotnet bundle“ – ak sa používa NuGet balík **BundlerMinifier.Core** a v csproj nie je definovaný v tagu „**PackageReference**“ ale ako „**DotNetCliToolReference**“
- Pre lepšiu kontrolu nad Bundlingom sa dá použiť VS Extension „Bundler & Minifier“, ktorý pridá záložku do kontextového menu a automaticky generuje bundleconfig.json podľa pridaného obsahu
- Dá sa použiť aj Gulp alebo Grunt – v template SPA aplikácií sa ale používa **webpack**.

EX_8 – BUNDLING A MINIFIKÁCIA

Kroky:

1. Spustím projekt bez bundlera – sledujem output okno
2. Spustím projekt s nainštalovaným NuGet balíkom **BuildBundlerMinifier** – sledujem output okno, kde pred buildom zbehnú procesy z **bundleconfig.json**
3. Nainštalujem NuGet balík **BundlerMinifier.Core**
4. V *.csproj súbore zmením tag **PackageReference** na **DotNetCliToolReference**
5. Použijem príkaz **dotnet bundle**

Predpoklady:

- V kroku 1. bundling neprebehne
- V kroku 2. prebehne pri builde bundling nastavený v súbore **bundleconfig.json**
- V kroku 5. prebehne bundling pri volaní príkazu **dotnet bundle** v CLI nad projektom

ÚLOHA – NAČO TOTO VŠETKO??? (SVC, MDLWR, BNDLR)

- Vytvor si aplikáciu so SPA templatom na Angular alebo React
- Skús opísať aký je proces spustenia aplikácie s týmto templatom.
- Aké servicy a prečo sa používajú?
- Aký middleware sa používa, prečo a v akom kontexte?
- Používa sa bundling a minifikácia? Kde? Ako? Načo je v tomto prípade dobrý?

PRÍPRAVA NA NASADENIE APLIKÁCIE NA IIS

LEN PRE INFO

- Pred nasadením je nutných niekoľko prerekvizít, ktoré pri ASP.NET MVC neboli potrebné
- Net Core je nasadený ako nemanažovaný CLR kód (vlastnosť Application poolu v IIS)
- Metóda **CreateDefaultBuilder** už volá middleware **UseIISIntegration**() – vo verziách straších ako 2.0 musí byť explicitne zavolaný vo webhost builderi
- Nastavenia IIS je možné riadiť cez delegáta metódy **services.Configure<IISOptions>** v triede Startup.cs
- Pri nasadení je automaticky generovaný koreňový web.config s registrovaným modulom **aspNetCore**
- Ak časť adresárovej štruktúry stránky obsahuje aj iný manažovaný kód pre tento adresár musí byť odstránený handler **aspNetCore** vo **web.config** súbore platnom pre daný adresár <handlers> <remove name="aspNetCore" /> </handlers>
- IIS dokáže hostovať ASP.NET Core stránky len v prípade, že je doinštalovaný balík „.NET Core Hosting Bundle“

ČASŤ V

Projekty typu ASP.NET Core MVC

FRONT-END VIEWS

- Razor sytax – ??treba??
- Dôležité súbory v MVC šablóne
- Dependency Injection v Razor Views
- TagHelpers v ASP.NET Core
  - Vstavané TagHelpers
  - Vlastné TagHelpers
- ViewComponents – alternatíva k PartialViews
- R4MVC

RAZOR SYNTAX

- @ - jednoriadkový príkaz so zobraziteľnou návratovou hodnotou (string, int a pod.)
- @() – jednoriadkový príkaz so zobraziteľnou návratovou hodnotou – umožňuje formátovať výpis v ()
- @* - začiatok sekcie komentáru (*@ ukončuje) – narozdiel od HTML <!-- nie je súčasťou kódu stránky v browseri
- @{} – sekcia s kódom bez návratovej hodnoty
- @@ - escape príkaz ak je súčasťou výpisu „zavináč“

DÔLEŽITÉ VIEWS SÚBORY V ASP.NET CORE

- /Views/_ViewImports.cshtml – definuje všetky (using) namespace, časti kódu, importuje taghelpery a injectuje služby, ktoré používa celá aplikácia (všetky views). Ak je v inom adresári importy platia len pre views v danom adresári. Dobrou praxou je použiť **using** na namespace s modelmi – skráti sa tým referencia na modely v hlavičke view.
- /Views/_ViewStart.cshtml – spustí definovaný kód pred načítaním každého view (neplatí pre partial views ani layouty). Podobne ako **_ViewImports** môže byť v inom adresári, a teda platí len pre views v danom adresári.
- /Views/Shared/_Layout.cshtml

DEPENDENCY INJECTION V RAZOR VIEWS

- Jednoducho Razor directívou **@inject IRozhranie _objekt**
- Ak je služba injectnutá do _ViewImports.cshtml je použiteľná z každého view
- Ak je služba injectnutá do _Layout.cshtml – je použiteľná len v danom layoute (len v tomto súbore)
- Zvyčajne využitie
  - Vlastné služby – viď. nasledujúce demo
  - Autorizačné a autentifikačné služby (SignInManager, UserManager, RoleManager sú injectnuteľné)

EX_9 – RAZOR DIREKTÍVY **USING A INJECT**

Kroky:

1. Vytvorím nový projekt ASP.NET Core MVC
2. Vytvorím si službu, ktorá vracia aktuálny čas metódou **GetCurrentTime** a dátum metódou **GetCurrentDate**

   Pozn. Pre univerzálnosť môžem vytvoriť rozhranie a implementovať ho
3. Zaregistrujem službu ako Singleton
4. Do **_ViewImports.cshtml** pridám **using** na namespace, kde sa nachádza služba (resp. interface)
5. Injectnem službu do **_ViewImports.cshtml**
6. V **_Layout.cshtml** vymením aktuálny rok v časti footer za dátum zo služby

TAG HELPERS

- TagHelper je alternatíva k HtmlHelperom, kt. pripomína vlastné direktívy a šablónovanie z Angularu alebo Reactu
- Vstavané TagHelpery zjednodušujú syntax HtmlHelperov pridávaním atribútov do HTML tagov a výmaz Razor syntaxe
- Najčastejšie používané TagHelpery
  - <a> - klasický anchor tag obohatený o linkovanie na akcie v controlleroch a route parametre
  - <select> - binding na **IEnumerable<SelectListItem>,** t.j. na model alebo jeho časť
  - <input> - klasický input tag s možnosťou bindovať obsah k modelu alebo jeho časti pomocou atribútu **asp-for**
  - <label> - klasický label tag – nie je linkovaný cez for k inputu ale cez **asp-for** k modelu alebo jeho časti modelu. Ak je rovnaká časť modelu použitá v TagHelperi typu **input**, tak je label priradený k danému inputu
  - <form> - klasický form tag obohatený o linkovanie na akcie v controlleroch a route parametre
  - - ak obsahuje validačný atribút **asp-validation-for**, tak generuje validačnú logiku nad bindovaným modelom alebo jeho časťou. Vyvolaný je štandardne po submitnutí formu, v ktorom sa validačný tag nachádza
  - - klasický div tag, ktorý môže obsahovať atribút **asp-validation-summary** pre zobrazenie sumárnej validácie formu. Vyvolaný je štandardne po submitnutí formu, v ktorom sa validačný tag nachádza

EX_10 – VSTAVANÉ TagHelpers v ASP.NET Core

Kroky:

1. Vytvorím novú ASP.NET Core MVC aplikáciu
2. Vytvorím si demonštračný viewmodel (triedu) DemonstrationTagHelperViewModel, ktorý obsahuje prop int **IntegerValue**, prop string **StringValue** a **IEnumerable<SelectListItem> ListCollection** s defaultne inicializovanou kolekciou **SelectListItem** položiek
3. Prop **StringValue** priradím anotáciu **[Required]**
4. Vo view Index.cshtml použijem model **DemonstrationTagHelperViewModel**
5. Pridám TagHelpery typu
    1. <a asp-controller="Home" asp-action="Index" asp-route-contactname="Janci">Open contact page
    2. <environment names="Development"> s obsahom a <environment include="Production" exclude="Development"> s obsahom
    3. <form method="post" asp-controller="Home" asp-action="Index"> s obsahom
    4. Do obsahu tagu form pridám
        1. Submit button
        2. Div s validačným sumarom
        3. Label span a input pre StringValue
        4. Label a input pre IntegerValue
        5. Select pre ListCollection z modelu a defaultnu option s value=-1
6. V HomeControlleri pridám metódu Index s anotáciou HttpPost, vstupom DemonstrationTagHelperViewModel a obsahom (napr. 2 status cody podla validity modelu)
7. V HomeControlleri pridám do metódy Contact vstup string **contactName** a vyplním obsah metódy tak, aby som videl či bolo niečo vo vstupe zadané

EX_10 – VSTAVANÉ TagHelpers v ASP.NET Core

Predpoklady:

- Post na metódu Index s neplatnými dátami prejde až na backend a neprejde cez validáciu na strane backendu (ak bola použitá property **ModelState.IsValid**
- Post na metódu Index s platnými dátami prejde a vygeneruje sa príslušný status code
- Pre validáciu na strane FE je potrebné pridať validačné knižnice jquery vrátane unobtrusive verzie
- Preklik pomocou linku na stránke na stránku **Contact** zobrazí custom hlášku a v URL bude parameter *contactName=Janci*
- Ak aplikáciu spustím development profilom zobrazí sa príslušný obsah. Podobne pre Production profil.
- Ak kolekcia **ListCollection** obsahuje prvky – tie sa zobrazia v select kontrolke

VLASTNÉ TagHelpers - PRAVIDLÁ

- Zvyknú sa definovať v samostatnom adresári Helpers
- Podľa konvencie každá TagHelper trieda by mala obsahovať suffix **Helper**
- Trieda definujúca tag helper musí dediť od base triedy TagHelper
- Trieda definujúca tag helper musí overridnuť metódu **Process** alebo async Task **ProcessAsync**
- **Dekorátor** (anotácia) triedy **HtmlTargetElement** špecifikuje cieľový HTML tag (jeho názov) pre naviazanie helpera
  - Môže obsahovať ešte parameter **ParentTag**, ktorý špecifikuje rodičovský nadradený tag
  - Môže obsahovať parameter **TagStructure**, ktorý špecifikuje, či je tag párový alebo nepárový
- Dekorátor (anotácia) triedy **RestrictChildren** špecifikuje tagy, ktoré môžu byť vnútri taghelperu

VLASTNÉ TagHelpers - PRAVIDLÁ

- Parameter output metódy process obsahuje výstupný obsah a správanie sa tag helperu
- Pre zistenie informácii o route môžete injectnuť do konštruktora **IActionContextAccessor** actionContextaccessor.ActionContext.RouteData – Values[“action”];
- Všetky atribúty v tagu v Razor sú public properties v TagHelper triede
- Ak použijete property s názvom neviemCo, v Razor syntaxi je atribút nazvaný neviem-co
- Koreňový namespace, ktorý deklaruje akékoľvek TagHelpery musí byť zaregistrovaný
  - v súbore **_ViewImports.cshtml**
  - Príkazom **@addTagHelper** *,{namespace/solution}
- Binding na časti modelu je možný pomocou property typu ModelExpression
- Zdieľanie dát medzi taghelpermi je realizované pomocou kontextu

VLASTNÉ TagHelpers - OUTPUT

- Output je vlastne „return“ hodnota tag helperu
- Pre vyskladanie output môžete použiť jeho metódy a properties
  - TagName – názov cieľového tagu
  - TagMode – definuje či je tag párový alebo nie
  - Attributes – kolekcia prvkov TagHelperAttribute(string nazov, string value)
  - Content – obsahuje viacero metód pre zápis obsahu tagu (napr. **AppendHtml, AppendLine, AppendFormat, Append**)
  - GetChildContentAsync – spustí obsah childov v helper tagu a vráti ich renderovanú reprezentáciu

  Pozn.: Ak použijete **GetChildContentAsync** metódu mali by ste použiť namiesto metódy **Process** metódu async Task **ProcessAsync** a awaitnuť metódu **GetChildContentAsync**

EX_11 – VLASTNÉ TagHelpers A) S CHILD OBSAHOM

- Kroky:
  1. V adresári Helpers vytvorím novú triedu FormGroupHelper, ktorá dedí od triedy TagHelper a overriduje metódu ProcessAsync.
  2. Triedu dekorujem atribútom HtmlTargetElement s obsahom „form-group“ a **RestrictChildren** s obsahom „label“, „input“
  3. Vytvorím verejnú property **Class** s getterom a setterom
  4. V metóde **ProcessAsync** priradím **TagName** hodnotu „*div*“, **TagMode** hodnotu **TagMode.StartTagAndEndTag**
  5. Do outputu pridám atribút class s obsahom form-group a obsah property Class ak nie je tento obsah prázdny
  6. Nakoniec pridám HTML obsah – všetky dcérske tagy definované vnútri Tagu form-group metódou GetChildContentAsync
  7. Vo _ViewImports.cshtml zaregistrujem taghelpery pomocou **@addTagHelper *,Ex_11_CustomTagHelpers**
  8. Rebuildnem projekt
  9. Vo view Index.cshtml pridám taghelper **<form-group>** a dovnútra label a input

EX_11 – VLASTNÉ TagHelpers A) S CHILD OBSAHOM

- Predpoklady:
  - V zdrojovom kóde stránky je len štandardný HTML kód vygenerovaný taghelperom
  - Obsah form groupu aj formgroup sa zobrazuje normálne aj na stránke aj v kóde ako DIV tag
  - V kóde View sa tag zobrazí hrubým fialovým písmom vo VS 2017

EX_11 – VLASTNÉ TagHelpers B) NEPÁROVÝ TAG S ENUM

Kroky:

1. V adresári Helpers vytvorím adresár HelperModels a v ňom triedu **Enums**, ktorá definuje **public enum DefaultStyles** so štýlmi, ktoré používa bootstrap (primary, warning, ...).
2. Vytvorím private metódu ktorá mi vráti **Dictionary<DefaultStyles,string>** a inicializujem jej getter pre všetky enum štýly
3. Napr. Enum štýl Primary ako {DefaultStyles.Primary, "-primary"}
4. Vytvorím public metódu vracajúcu string pre daný enum (na jednoduchší spôsob som neprišiel)
5. Triedu Enums registrujem v Startup.cs ako singleton službu
6. Vytvorím nový taghelper LabeledInputHelper, kt. dedí od TagHelper
7. Pridám dekorátor [HtmlTargetElement("labeled-input", TagStructure=TagStructure.WithoutEndTag)]
8. Injectnem Enums
9. Vytvorím properties pre parametre taghelpera (DefaultStyles LabelStyle, stringy Name, Value, Label, Type, Placeholder)
10. V metóde Process vytvorím štruktúru HTML s labelom a inputom. TagName nastavím na „", aby nebol vytvorený žiaden obaľovací Tag vo výslednej stránke.
11. Rebuildnem solution
12. Do Views/Home/Index.cshtml pridám nový taghelper aj s atribútmi

EX_11 – VLASTNÉ TagHelpers B) NEPÁROVÝ TAG S ENUM

Predpoklady:

- Ak definujem atribút label-style s prázdnym obsahom stránka mi vráti pri spustení exception
- Ak atribút label-style nedefinujem vôbec label by mal byť len čistý text
- IntelliSense mi ponúka možnosti pri label-style podľa enum, ktorý som vytvoril
- Tag LabeledInput je nepárový

OTÁZKY ... ?!

1. Do akej miery má zmysel robiť TagHelpery?
2. Kde je tá zdravá hranica „univerzálnosti“ a „konkrétnosti“?
3. Načo sú TagHelpery ideálne z pohľadu biznisu?
4. Načo sú TagHelpery mne ako developerovi?

Ad1: Projekt k použitiu kontextu na prenos dát a ModelExpression na binding modelu nájdete na github

Ad2: Ako typ property je možné použiť aj celú modelovú triedu

VIEWCOMPONENT

- Podľa docs (https://docs.microsoft.com/en-us/aspnet/core/mvc/views/view-components?view=aspnetcore-2.1)
  - Je identický s PartialView ale mierený na iné úlohy – znovupoužiteľnosť napr. v Layoutoch, navigačnom menu a pod.
  - Môže obsahovať parametre a biznis logiku
  - Nepoužíva model binding – pracuje len s poskytnutými dátami
- ViewComponent je možné vytvoriť (ideálne všetky 3 naraz)
  - Dedením od ViewComponent base triedy
  - Dekorovaním triedy anotáciou [ViewComponent] alebo dedením od triedy s týmto dekorátorom
  - Konvenciou, kde názov triedy obsahuje suffix ViewComponent
- ViewComponent trieda musí byť public, top-level, neabstraktná trieda.
- Ako výstup vracia typ IViewComponentResult, (dedí od ActionResult), t.j. je možné vrátiť View spolu s modelom

VIEWCOMPONENT

- Pridávajú sa pomocou v Razor view zavolaním a awaitnutím metódy Component.InvokeAsync
- !POZOR! Metódu Invoke bežne zavolať z Razor view nemôžete
- Vložiť je ho možné aj pomocou TagHelpera <vc:{nazov} atribut="hodnota atributu"> - aj ak je metóda Invoke
- Pre použitie TagHelper verzie vkladania ViewComponentov je nutné registrovať projekt ako tag helper vo _ViewImports.cshtml
- Podporuje dependency injection metódou constructor injection
- Nie je súčasťou životného cyklu controlleru

Disclaimer: Z môjho pohľadu nevidím veľký rozdiel medzi používaním ViewComponent a PartialView.

EX_12 - VIEW COMPONENTS

Kroky:

1. Vytvorím nový ASP.NET Core MVC projekt
2. Do adresáru ViewComponent pridám
    1. Triedu CurrentTimeViewComponent.cs
    2. Triedu CurrentDate a pridám jej dekorátor [ViewComponent]
    3. Triedu CurrentDay a oddedím ju od triedy ViewComponent
3. Zaregistrujem projekt ako taghelper v _ViewImports.cshtml
4. Sledujte kde sa vyskytol problém pri kompilácii

Oprava:

1. Všetky triedy dedia od ViewComponent aby bolo možné vrátiť View na výstupe
2. Vytvorím 3 views v nasledujúcej štruktúre: /Shared/{nazov_triedy}/{nazov_view}.cshtml
3. Views sú súčasťou returnu v každej z tried
4. Zavolám všetky TagHelpery v _Layout súbore – napr. vo footeri

R4 MVC

- Aj TagHelpery sú náchylné na chyby – plain textové špecifikovanie atribútov action, controller, a pod.
- R4 je NuGet balík - snaha o vygenerovanie tried s literálmi, t.j. metódami, ktoré je možné volať priamo z FE
- Odstráni problém preklepov pri volaní metód z FE a uľahčí generovanie odkazov na metódy contollera z FE
- Aktuálne je v alfa verzii a je v ňom viacero chýb. Problémom je aj nutnosť manuálne generovať referencie na metódy CLI príkazom Generate-R4MVC

Disclaimer: Neodporúčam ho používať v jeho aktuálnej podobe. Generovanie kódu spôsobuje občas chyby hlavne pri parametroch metód, ktoré majú zložité typy a pri async Task metódach.

ČASŤ VI

Projekty typu ASP.NET Core MVC

CONTROLLERS

- IActionResult interface
- Kontroléry s API funkciami
- Dekorátory kontrolérov, metód a parametrov
  - Štruktúrovanie REST API
  - Best match resolution – volanie API metódy, ktorá najviac vyhovuje requestu (nie je súčasťou .NET Core)
- Swashbuckle a Swagger pre testovanie a dokumentovanie API

IActionResult

- IActionResult je nový návratový typ pre rôzne typy akcií v ASP.NET Core
- Je univerzálnym typom pre veľkú väčšinu reponsov – akcie, response codes, obsah, súbory, atď.
- Je univerzálnym typom ak je možných viac responsov rôzneho typu z jednej akcie – napr. response kód a akcia
- Pre udržanie zrozumiteľnosti kódu sa preto môžu používať dekorátory [ProducesResponseType(int, Type)]

API kontroléry v ASP.NET Core MVC

- Štandardné kontrolery a WebApi kontroléry sú identické – oba dedia od triedy Controller
- Rozdiel je len v použitých dekorátoroch (pre routing, http verbs, názvy akcií)
- Návratový typ môže byť model, elementárny typ alebo pokojne IActionResult – univerzálny návratový typ
- WebApi tak môže vrátiť dve informácie naraz v jednom return statemente napr. pomocou metódy Ok(data)
  - Dáta – objekt data štandardne formátovaný na výstupe ako JSON
  - Response code – generuje ho metóda Ok – 200
- Návratová hodnota môže byť aj napr. metóda StatusCode(int code, object data)

DÔLEŽITÉ DEKORÁTORY - CONTROLLER

- Dekorátory conrollera
  - [Produces] – určuje výstupný formát - defaultná hodnota je application/json
  - [Consumes] – určuje povolený vstupný formát (Content-Type v hlavičke requestu)
  - [Route] – určuje routu k akciám controllera, ktorá nemusí byť mapovaná v Startup.cs
  - Pre XML response nestačí špecifikovať application/xml ale je nutné pridať formatter službu.
  - V Startup.cs metóde ConfigureServices fluent api metódou services.AddMvc().AddXmlSerializerFormatters()

DÔLEŽITÉ DEKORÁTORY - METÓDY

- Určenie metódy HTTP – HttpGet, HttpPost, HttpPut, HttpDelete, HttpHead, HttpOption, HttpPatch
- HttpVerbs trieda z ASP.NET MVC v ASP.NET MVC Core neexistuje – preto sa neodporúča používať dekorátor AcceptVerbs
- Route – špecifikovanie routy konkrétnej akcie
- ActionName – overridne názov akcie
  - vo WebApi je to brané ako bad practice
  - rôzne requesty na ten isty verb by sa mali riešiť overloadmi danej metódy
  - !!!ALE!!!

Best match resolution nefunguje ako v controlleroch, ktoré dedili od ApiController a dôležitý pre rozpoznanie metódy je len **počet** parametrov nie typy parametrov.

- Preto sa vytvárajú vlastné constraints ako samostatná trieda dediaca od triedy ActionMethodSelectorAttribute – nový dekorátor
- Mali by byť ale používané len v špeciálnych prípadoch – napr. pri volaniach z externých API a pod.
- Obchádzka je použitie vlastného názvu akcie s Route atribútom "[action]" prípadne vlastnou routov

DÔLEŽITÉ DEKORÁTORY – PARAMETRE AKCIÍ

- [FromBody] – max 1 parameter v akcii. Hodnota parametra sa získa z body časti requestu
- [FromForm] – parameter sa získa zo submission formu v requeste – určený len pre post, put a delete
- [FromService] – parameter sa získa z injectovanej služby method injection prístupom
- [FromHeader] – parameter sa získa z hlavicky requestu (je možné došpecifikovať názov pomocou Name atribútu). Ň

  Pozn.: Do verzie .NET Core 2.1 fungoval binding len pre string hodnoty v hlavičke.
- [FromQuery] – parameter sa získa z URL - query stringu
- [FromRoute] – niečo ako náhrada za FromUri – mapuje časti routy na parametre podľa ich názvu viď. Príklad

```
[HttpPut("{employeeId}/{coffeeType}")]
public void UpdateMostAdoredCoffeeType([FromRoute] int employeeId, [FromRoute] int coffeeType)
```

OTÁZKA - API

- Aká je správna konvencia?

1. /api/Coffee/FindByName?name=daco
2. /api/CoffeeByName?name=daco
3. /api/Coffee/FindByName/daco

Problémom môže byť niekedy nastavenie automatického dokumentačného systému (napr. Swagger) tak aby fungoval s tou mojou konvenciou.

- Viac o tejto téme nájdete napr. na https://restfulapi.net/resource-naming/
- Môžeme o tom porozprávať, ak ostane čas

EX_13 – IActionResult, WebApi a dokumentácia API

Kroky:

1. Vytvorím nový ASP.NET Core MVC projekt
2. Vytvorím interface IBeverageService s implementáciou CoffeeService a registrujem ho ako scoped v Startup.cs
3. V CoffeeService vytvorím metódy na získanie jedného nápoja Beverage a všetkých IEnumerable<Beverages>
4. Vytvorím nový API controller CoffeeController a injectnem službu CoffeeService pomocou constructor injection
5. V Get metóde bez id parametra zavolám All zo služby a vrátim ako súčasť vybraného response typu
6. V Get metóde s id parametrom zavolám GetBeverage(id) a vrátim ako súčasť vybraného response typu
    1. Problém s duplicitným routovaním vyriešim doplnením routy {id} do dekorátora HttpGet
7. Vytvorím HttpGet metódy SingleByName (vráti Beverage) a FindByName(vráti IEnumerable<Beverage>) s parametrom [FromQuery] string name
8. Problém s duplicitným routovaním vyriešim doplnením routy [action] do dekorátora HttpGet

EX_13 – IActionResult, WebApi a dokumentácia API

Predpoklady:

Názov	HTTP metóda	Endpoint	Parameter(zdroj)	Adekvátna zmena názvu endpointu
Get	GET	/api/Coffee	NaN	/api/Coffee/All
Get	GET	/api/Coffee/{id}	int id (route)	/api/Coffee/SingleById?id={id}
SingleByName	GET	/api/Coffee/SingleByName?name={name}	string name (query)	-
FindByName	GET	/api/Coffee/FindByName?name={name}	string name (query)	/api/Coffee/FilterByName?name={name}

ČASŤ VII

Projekty typu ASP.NET Core MVC

Entity Framework Core

- Zmeny oproti EF6.0 – stav implementácie EF Core
- Bezpečné miesto pre ConnectionStrings
- DB kontext a registrácia DB servera ako zdroja
- Code-first prístup – vytváranie DB kontextu
- Migrácie – princíp a CLI príkazy
- Database-first prístup – Scaffolding databázy
- Injektovanie DB kontextu
- Typy načítavania entít s reláciami
  - Eager loading - .Include().ThenInclude()
  - Explicit loading - .Load()
  - Lazy loading – podpora v EF Core 2.1
- Najlepšie praktiky pri načítavaní (platné pre EF Core 2.0)

MARKANTNÉ ZMENY OPROTI EF 6.0

- EF.Functions.Like – zjednodušuje používanie wildcardov v LINQ výrazoch
- EF.CompileQuery – zjednodušuje volanie parametrizovaných query v cykle
- HasQueryFilter FluentAPI metóda v modelBuilderi – umožňuje filtrovať vrátené záznamy (napr. pri použití flagov v DB a podobne)
- Seedovanie dát – funkcia HasData

CHÝBAJÚCE FUNKCIE

- Spomínal som ešte v úvode viac vecí
- Celý prehľad porovnania EF 6 a EF Core (EF7+) je na https://docs.microsoft.com/en-us/ef/efcore-and-ef6/features

SPRÁVA ConnectionStrings

- Odporúčané ukladacie miesto pre connection strings je appsettings.json sekcia „ConnectionStrings“
- Príklad:

  „ConnectionStrings“:{

  “DefaultConnection”: “…”

  }
- Získať connectionString je možné
  - Ako štandardnú konfiguračnú položku pomocou selektora
  - Ako štandardnú konfiguračnú položku pomocou metód GetSection a GetValue
  - Ako štandardnú konfiguračnú položku vytvorením POCO triedy a injectnutím IOption
  - Pomocou metódy GetConnectionString(„nazov_cs“)

DB KONTEXT A REGISTRÁCIA DB SERVERA AKO ZDROJA

- Každý DB kontext musí byť registrovaný, aby mohol byť neskôr injectovaný do služieb
- Každá služba používajúca DB kontext by mala mať rovnaký lifetime (Scoped)
- Registruje sa v Startup.cs v metóde ConfigureServices príkazom services.AddDbContext<trieda>()
- Pre nastavenie zdroja (providera) je možné použiť delegáta
- MSSQL ako zdroj nastavíte v delegátovi metódou .UseSqlServer(connection_string)
- Alternatívne je možné použiť .UseSqlite(connection_string) alebo .UseInMemoryDatabase(db_name)
- Pre iné databázy zvyčajne existujú drivery (napr. MongoDB, Postgres, MySQL, a.i.)
- Pre niektoré z 3rd party databáz existuje aj podpora v EF Core priamo

CODE-FIRST PRÍSTUP

- Vytváranie štruktúry DB v triede databázového kontextu
- prop DbSet<model_trieda> - viaže modelovú triedu k dátovému modelu ako entitu a vytvára alias podľa názvu
- Štruktúra entít v DB je definovaná v samostatných modelových triedach
- Konvencie pre modelové triedy
  - Ako primárny kľúč sa použije int alebo GUID atribút s názvom modelovej triedy a suffixom Id. Napr. prop int CoffeeId. Override konvencie je možný pomocou dekorátorov.
  - Relácia one to many sa definuje
    - ako prop virtual ICollection<model_trieda> Navigacia; na strane ONE
    - ako prop model_trieda refId; na strane MANY
  - Relácia many to many sa definuje väzobnou entitou, kde sa spájajú 2 kolekcie a ich cudzie kľúče (EF Core momentálne nepodporuje many to many bez väzobnej entity)
- Pre lepšiu kontrolu nad reláciami, konvenciami a štruktúrou DB je možné použiť model builder v DB kontexte

CODE-FIRST PRÍSTUP - DEKORÁTORY

- [Key] – špecifikuje prop, ktorý je primárnym kľúčom (pozor ak existuje aj Id generované konvenciou). Kompozitný kľúč je možné generovať len metódou .**HasKey**() pomocou model buildera. Konvencie budú ďalší kľúč ignorovať.
- [ForeignKey] – špecifikuje prop, ktorý je cudzím kľúčom
- [DatabaseGenerated] – umožňuje overridnúť kľúč generovný konvenciou
- [Required] – mení príznak NULL a NOT NULL v atribúte
- [Table] – mení alias pre názov entity v DB
- [Column] – mení alias pre názov atribútu v DB
- [DataType] – môže overridnuť dátový typ v DB ak sú typ prop a DataType typ navzájom konvertovateľné
- [Timestamp] – vytvorí timestamp atribút. Nutné použiť príkaz .IsConcurrencyToken v modelbuilderi
- [MaxLength] – atribút maximálnej dĺžky stringu/čísla v DB
- [Index] – vytvorí index nad daným stĺpcom. Podľa konvencie je každý cudzí kľúč indexovaný aj bez dekorátora.

POUŽITIE DÁTOVÉHO ZDROJA

- Nemal by byť silne naviazaný na controller – prístup k dátovej vrstve by mal byť riešený službami
- DB kontext by nemal byť vytvorený ako inštancia pomocou "new" ale injectnutý do služieb, kde je používaný
- Všetky zmeny v dátovom kontexte sú sledované a zmeny sa do DB ukladajú až po uložení kontextu príkazmi
  - SaveChanges
  - await SaveChangesAsync
- Ak chcete dáta len pasívne zobraziť bez možnosti úprav môžete použiť pri LINQ príkazoch .AsNoTracking metódu
- Ak si to predsalen rozmyslíte, musíte použiť metódu Attach aby bol znovu objekt sledovaný

MIGRÁCIE – CLI PRÍKAZY A COMMON PRACTICE

- Add-Migration {nazov} – pridá migráciu s vybraným názvom
- Remove-Migration {nazov} – vymaže migráciu podľa názvu.
- Update-database – aktualizuje databázu pomocou poslednej vytvorenej migrácie
- Odporúčania:
  - Migrácie by mali byť vytvárané pri elementárnych zmenách
  - Návrat späť pri veľkých zmenách medzi migráciami je náročný
  - Názov migrácie by mal vystihovať realizované zmeny v dátovom modeli
  - Migrácie nikdy neodstraňujte manuálne ale CLI príkazom. Po migrácii ostávajú referencie v projekte aj v DB tabuľke _EFMigrationHistory

SPÔSOBY NAČÍTAVANIA VIAZANÝCH ENTÍT

- Explicit loading – načíta všetky naviazané entity FluentAPI metódou Include. Každá nižšia úroveň väzby sa načítava metódou ThenInclude()
- Eager loading – dáta nie sú automaticky viazané a musia byť načítané metódou Load() pre záznam (Entry) nad
  - Reláciou – metóda Relation() – jeden záznam
  - Kolekciou – metóda Collection() – viac záznamov
- Lazy loading – súčasťou balíčka Microsoft.EntityFrameworkCore.Proxies.
  - Pri registrácii kontextu je potrebné použiť lambda výrazom metódu .UseLazyLoadingProxies()
  - Do modelov entít je potrebné injectovať lazy loader a upraviť gettery a settery nad viazanými kolekciami a entitami

https://docs.microsoft.com/en-us/ef/core/querying/related-data#lazy-loading

MOŽNÝ PROBLÉM PRI NEKONEČENÝCH REFERENCIÁCH

- Nekonečné referencie môžu vzniknúť pri many to many reláciach
- Preto sa zvykne používať konfigurácia hlavne pri webApi výpisoch JSONu ako fluentApi metóda pri pridávaní služby AddMvc().AddJsonOptions(o=>o.SerializerSettings.ReferenceLoopHandling = ReferenceLoopHandling.Ignore)

EX_14 – DBCONTEXT A CODE-FIRST PRÍSTUP

- Kroky:

1. Vytvorím nový ASP.NET Core MVC projekt
2. V adresári Data vytvorím dve triedy CoffeeDbContext a CoffeeDbContextLite, kt. dedia od DbContext
3. Obom vytvorím prázdny public konštruktor s parametrom DbContextOptions<trieda_ctx> options, kt. posiela parameter do base konštruktora
4. V adresári Models/DbModels vytvorím triedy Coffee, Employee a EmployeeCoffee
5. V kontexte pridám prop DbSet<model_trieda> modelTrieda pre všetky 3 triedy
6. V Startup.cs registrujem kontexty pomocou AddDbContext<>. Pre prvý použijem SQL Server, pre druhý SQLite.
7. Connection strings pridám do appsettings.json a použijem ich pri registrácii kontextov.

EX_14 – DBCONTEXT A CODE-FIRST PRÍSTUP

8. V triede Coffee pridám tieto props:
   - Guid CoffeeId = Guid.NewGuid()
   - string Name
   - string Brand
   - ICollection<EmployeeCoffee> EmployeeCoffee
9. V triede Employee pridám tieto props:
   - Guid EmployeeId = Guid.NewGuid();
   - string Name
   - string Workplace
   - ICollection<EmployeeCoffee> EmployeeCoffee
10. V triede EmployeeCoffee pridám tieto props:
    - Guid EmployeeCoffeeId = Guid.NewGuid()
    - Guid EmployeeId
    - Guid CoffeeId
    - Coffee coffee s foreign key smerujúcim na CoffeeId
    - Employee employee s foreign key smerujúcim na EmployeeId

EX_14 – DBCONTEXT A CODE-FIRST PRÍSTUP

11. V triede kontextu overridnem OnModelCreating a za volaním base konštruktoru pridám „data seed“ (len v 2.1).
    - Vytorím si privátnu metódu PopulateDbIfEmpty(ref ModelBuilder modelBuilder) a zavolám ju z OnModelCreating()
    - Každý záznam pridám ako modelBuilder.Entity<entita>().HasData(new entita{DATA});
    - Pridám takto všetky dáta aj väzby ktoré chcem/potrebujem.
12. Pre každý kontext vytvorím samostatné migrácie príkazom Add-Migration Init –Context {trieda} –OutputDir Migrations\{provider}
13. Pre každý kontext aktualizujem databázu príkazom Update-Database –Context {trieda}
14. V HomeController injectnem do konštruktora oba kontexty
15. Pre akciu Index použijem CoffeeDbContext a EagerLoading metódu - vrátim model
16. Pre akciu About použijem CoffeeDbContextLite a ExplicitLoading metódu – vrátim len .Coffee ako model
17. V Index.cshtml view zobrazím obsah celého modelu v 2 cykloch foreach
18. V About.cshtml view zobrazím obsah modelu v 2 cykloch foreach a naviazané entity postupne loadujem. Injectne preto do About.cshtml databázový kontext SQLite.

EX_14 – DBCONTEXT A CODE-FIRST PRÍSTUP

Predpoklady:

- Ak nepoužijem pri migrácii a update špecifikáciu kontextu – dostanem Exception
- Databázy sa vytvoria automaticky po updatnuti databázy CLI príkazom
- Eager loading funguje bez toho aby som musel vo view očakávať NullPointerException
- Explicit loading môže vrátiť NullPointerException v prípade, že vo view neloadnem naviazané entity

Disclaimer:

- *Práca s DB nemá byť súčasťou controllera – je to pre demonštráciu*
- *Eager loading by nemal byt používaný na strane Views podobne ako ani DB kontext – je to pre demonštráciu*

INTERMEZZO – WORKSHOP ÚLOHA

- Vytvorte ASP.Net Core MVC (2.1) projekt s touto špecifikáciou
  - Dátový zdroj je SQLite s DB uloženou lokálne
  - Dátový model obsahuje štruktúru z projektu Ex_14, t.j. entity Coffee, Employee a EmployeeCoffee (verzia modelu v1)
  - Do dátového modelu pridajte entity CoffeeType (CoffeeTypeId, type, description), EmployeePreferedCoffees(id, CoffeeId, EmployeeId) – verzia modelu v2
  - Naseedujte dáta pomocou HasData metódy
  - Vytvorte v databáze pohľad V_EmployeePreferedCoffeesCount a nalinkujte ho na code-first model (group podľa Employee)
  - Vytvorte v databáze procedúru GetCoffeeTypes s parametrom @type a použite túto procedúru v controlleri
  - Výstupy z pohľadu zobrazte v Index.cshtml a výstupy z procedúry v Contact.cshtml

DATABASE-FIRST PRÍSTUP

- Celý objektový model je vygenerovaný z už existujúcej databázy
- Dizajnér je možné použiť len v rozhraniach DB Servera (t.j. napr. SSMS pre MSSQL, WorkBench pre MySQL a pod.)
- Vygenerovaný model nie je EDMX – ADO.NET ale Code-first model
- Scaffold-DbContext „CS“ provider –o output_dir
- Provider je napr. Microsoft.EntityFrameworkCore.SqlServer
- Vo VS2017 funguje scaffolding na úrovni projektu, t.j. pri volaní príkazu cez PackageManagerConsole sa musíte dostať najprv do adresáru projektu

EX_15 – DATABASE FIRST PRÍSTUP - SCAFFOLD DB KONTEXTU

Kroky:

1. Vytvorím nový ASP.NET Core MVC projekt
2. Vygenerujem štruktúru databázy pomocou Scaffold-DbContext {CS} {provider} –OutputDir {addr} CLI príkazu
3. „Server=mssql02.qsh.eu,1481;Database=db1007848-coffee;User=db1007848-coffee;Password=HESLO;MultipleActiveResultSets=true“
4. Connection string presuniem do appsettings.json a DbContext zaregistrujem v Startup.cs
5. Vo vygenerovanom kontexte pridám prázdny konštruktor s volaním base konštruktora (ako v Code-first)
6. Implementujem výber dát v HomeControlleri (ako v code-first)
7. Implementujem zobrazenie dát v Index.cshtml (ako v code-first)

EX_15 – DATABASE FIRST PRÍSTUP - SCAFFOLD DB KONTEXTU

Predpoklady:

- Scaffold-DbContext môže ale nemusí fungovať na úrovni solutionu – niekedy je potrebné prejsť na úroveň projektu
- Vygenerované triedy sú v jednom adresári s kontextom – zvykom je oddeľovať tieto časti
- Generovaný model je menej náchylný na chyby ako Code-first
- Dáta sa zobrazia správne po spustení aplikácie v Home/Index view
- Často sa používa aj prístup, že prvá migrácia je generovaná code-first identitami, zvyšok entitného modelu sa tvorí dizajnérom a finálne tunovanie modelu sa rieši code-first prístupom

ČASŤ VIII

Projekty typu ASP.NET Core MVC

Identity framework

- Autentifikácia a autorizácia
- Roles based autorizácia
- IdentityRole – model pre roly
- IdentityUser – model pre userov
- Autentifikačné a autorizačné služby
  - SignInManager
  - RoleManager
  - UserManager

AUTENTIFIKAČNÉ A AUTORIZAČNÉ SLUŽBY

- V Program.cs musí byť volaný príkaz AddUserSecrets v prípade, že sa nepoužíva metóda CreateDefaultBuilder
- V metóde ConfigureServices sa pridá služba AddIdentity<userClass,roleClass>
- Delegát tejto metódy umožňuje nastaviť vlastnosti napr. sily hesla
- Naväzujúca fluent api metóda AddEntityFrameworkStores<Dbkontext>() prepája info o identitách s databázou
- Naväzujúca fluent api metóda AddDefaultTokenProviders() pridá službu providera autentifikácie.
- V metóde Configure je potrebné zavolať middleware app.UseAuthentication(); Metóda UseIdentity je deprekovaná
- DbContext, ktorý je používaný pre identity musí dediť od triedy IdentityDbContext<IdentityUser> namiesto DbContext
- Vzťahy identity modelu a vlastnej DB je možné modifikovať overridom metódy OnModelCreating() za volaním base konštruktora v tejto metóde

IdentityUser A IdentityRole – Kustomizácia obsahu DB

- Stĺpce o používateľovi je možné pridávať doplnením props do vlastnej triedy, ktorá dedí od IdentityUser
- Táto trieda je potom použitá pri registrácii v metóde AddIdentity<> ako prvý typ
- Podobne aj informácie o rolách používateľa je možné doplniť pridaním props do vlastnej triedy, ktorá dedí od IdentityRole
- Táto trieda je potom použitá pri registrácii v metóde AddIdentity<> ako druhý typ
- Všetky väzby – kľúče a kolekcie je tiež možné dodefinovať v týchto modeloch

Role based autorizácia a správa používateľov

- Dekorátory metód alebo controllera
  - [Authorize] – prístup len pre prihláseného používateľa
  - [Authorize(Roles=“”)] – prístup len pre určené roly
  - [AllowAnonymous] – povolá prístup neprihlásenému používateľovi
- RoleManager – manažér spravujúci roly v aplikácii
- SignInManager – manažér spravujúci prihlasovanie a odhlasovanie
- UserManager – manažér spravujúci používateľské účty v aplikácii

Ex_16 – Identity framework, role based auth. a individálne účty

Kroky:

1. Vytvorím novú ASP.NET Core MVC aplikáciu s autentifikačnou schémou pre ukladanie individuálnych účtov v databáze aplikácie
2. V Startup.cs v metóde AddIdentity upravím options pre heslo a podmienky registrácie nového používateľa
3. V adresári Models/DbModels vytvorím triedy Coffee (rovnaká ako v code-first) a EmployeeCoffee
4. EmployeeCoffee bude mať namiesto väzby na triedu Employee väzbu na ApplicationUser a na jeho kľúč Id
5. V Models/ApplicationUser doplním props string Name, string Workplace, ICollection<EmployeeCoffee> …
6. V Data/ApplicationDbContext v metóde OnModelCreating doplním väzby
   - Entity<ApplicationUser>().HasMany(b=>b.EmployeeCoffee).WithOne(b=>b.User);
   - Entity<Coffee>().HasMany(b=>b.EmployeeCoffee).WithOne(b=>b.Coffee);

Ex_16 – Identity framework, role based auth. a individálne účty

7. V Models/AccountViewModels/RegisterViewModel doplním props pre Name a Workplace s dekorátormi Display a Required
8. V Views/Account/Register doplním do formu atribúty Name a Workplace
9. V AccountControlleri injectnem RoleManager<IdentityRole> a upravím async metódu Register
   - Aby boli vytvorené roly Admin a User ak neexistujú (pomocou RoleManagera)
   - Aby boli do nového objektu ApplicationUser pridané aj Meno a Workplace
   - Aby bol používateľ po registrácii pridaný do roly User (pomocou UserManagera)
10. Pre UserManagera môžem vytvoriť extension metódu na získanie používateľa podľa pracovného miesta FindByWorkplace(string workplace)
11. V HomeControlleri pridám dekorátory pre autorizáciu podľa uváženia
12. V Views/Home/Index pridám výpis ak je prihlásený používateľ patrí do roly User a vyhľadám jeho Id podľa Workplace

Pridám novú migráciu a updatnem databázu

Ex_16 – Identity framework, role based auth. a individálne účty

Predpoklady:

- Pôvodná migrácia obsahuje základnú štruktúru pre autentifikáciu
- Nová migrácia dopĺňa databázu o ďalšie položky a väzby
- Po registrácii sa používateľ automaticky pridá do zvolenej roly (check DB)
- Používateľ ma prístup povolený do jednotlivých záložiek podľa dekorátorov v controlleri

Nedostatky:

- Migrácie by logicky mali byť 3
  - Pôvodná konfigurácia pre identity
  - Doplnené položky pre identity
  - Doplnené nové tabuľky s väzbami na identity

ČASŤ IX

Unit testy

- xUnit + Moq
- Konvencie a časté chyby
- dotnet test

KONVENCIE A ČASTÉ CHYBY

Konvencie

- Je dôležité udržiavať kód na dostatočne nezávislej úrovni – napr. DDD architekúra
- Minimálna konvencia je separovať definície a implementácie, biznis logiku prenášať do služieb v projektoch .NET Standard

Časté chyby

- Test runner vo VS 17 je háklivý na dlhé argumenty v PATH premennej prostredia
- Mocking frameworky by mali byť používané len na virtuálne metódy alebo abstraktné triedy a rozhrania nie na ich implementácie
- Ak predsa je nevyhnutné testovať implementáciu je nutný konštruktor implementácie bez objektov
- Mockovanie DBContext tried je tiež potrebné riešiť pomocou interfacu

XUNIT

- Vytvára nový objekt testovacej triedy pre každý test – zabezpečí sa tak vždy „čerstvý“ obsah objektu
- Assert.Throws – namiesto [ExpectedException] v Nunit
- Dekotátory
  - [Fact] – definuje jeden testcase
  - [Theory] – definuje sadu testcasov s rôznymi dátami
  - [InlineData] – vstupné dáta do testcase s jednoduchou štruktúrou (pole hodnôt) = počet vstupov testovacej metódy
  - [ClassData] – definuje typ vstupných dát. Trieda väčšinou dedí od cieľového typu
  - [MemberData] – vstupné dáta sú generované metódou – dáta sú získané referenciou nameof() a typ referenciou typeof()
- Testy je možné spustiť pomocou testrunnera vo VS 17 alebo CLI príkazom dotnet test

EX_17 – XUNIT A MOQ

Kroky:

1. Na testovanie použijem projekt Ex_2_2
2. Vytvorím nový projekt s Xunit testom a vymažem premenujem existujúcu triedu na CoffeeTest.cs
3. V projekte Services vytvorím CoffeeTypeNotAllowedException a túto výnimku hodím v metóde StartDutyCycle triedy PressoMachine
4. V testovacej triede vytvorím test EmployeeMakeCoffee_CoffeeMakerLaunch_CoffeeMakerWorks()
    1. Použijem Mock na vytvorenie kávovaru podľa ICoffeeMaker
    2. Použijem Mock na vytvorenie nového zamestnanca so vstupom mocknutého ICoffeeMaker
    3. Nad zamestnancom zavolám metódu MakeCoffee()
    4. Nad coffeeMakerom overím, či bola jeho metóda zavolaná raz.
5. V testovacej triede vytvorím test CanMakePressoCoffee_CoffeeTypeNotAllowed
    1. Použijem Mock na vytvorenie kávovaru podľa PressoMachine *NEODPORÚČANÉ*
    2. Použijem Mock na vytvorenie zamestnanca ako v bode 4.2.
    3. Použijem Assert.Throws s anonymnou zátvorkovou metódou pre zavolanie metódy MakeCoffee nad objektom zamestnanca

EX_17 – XUNIT A MOQ

Predpoklady:

- Oba testy passnu
- Ak v prvom teste použijem It.IsAny<ICoffeeMaker>() test failne pretože je volaný anonymný mock objekt namiesto pripraveného mock objektu mockCoffeeMaker
- Ak by som použil napr. PressoMachine namiesto ICoffeeMaker test failne kvôli použitiu implementácie namiesto interfacu/virtualnej metódy

ČASŤ X

.NET Core a Docker

- Zdrojové Docker image
- Dokerizácia aplikácie – Integrácia Dockeru a VS 2017
- Kompozícia projektov v .NET Core
- Spustenie a redirect portov

DOCKER

- Je vlastne virtuálka
- s oklieštenou verziou OS
- so sadou nainštalovaných frameworkov a služieb
- Pre vývoj v .NET Core sa hodia image
  - microsoft/dotnet – kompletný balík pre všetky typy aplikácií
  - microsoft/aspnetcore – optimalizovaný balík pre ASP.NET Core
  - microsoft/mssql-server-linux:2017-latest

DOCKER CLI PRÍKAZY

- docker pull – stiahne docker image
- docker run [–it|-d|-p x:y] –name {nazov} {kontajner}– spusti docker kontajner
- docker images – zobrazenie existujúcich docker imagov
- docker rm {name|id} – vymazanie docker kontajnera
- docker rmi {image} – vymazanie docker imagu
- docker ps –a – spustené kontajnery
- docker push – odošle docker image do docker registra
- docker build –t {tag} . – buildne docker image podľa dockerfile súboru v projekte
- docker tag {povodnyTag} {cielovyTag}:{verzia} – premenuje image podľa určeného tagu
- docker exec [-it] {kontajner}{prikaz} – spustí príkaz v bežiacom kontajneri

SPUSTENIE VIACRÝCH IMAGOV V TANDÉME

- Napr. MSSQL ako databázový server + ASP.NET Core aplikáciu
- docker-compose.yml

```
ex_18_dockerapp:
 image: ${DOCKER_REGISTRY}ex18dockerapp
 build:
  context: .
  dockerfile: Ex_18_DockerApp/Dockerfile
 depends_on:
   - db
db:
 image: "microsoft/mssql-server-linux"
 environment:
  SA_PASSWORD: Str0ngPassword!
  ACCEPT_EULA: Y
```

EX_18 – DOCKER COMPOSE VIACERÝCH PROJEKTOV

- Čo sa mi podarilo
  - Podarilo sa mi rozbehnúť oba image
  - Komunikácia medzi oboma imagmi funguje z databázy viem čítať aj sa k nej pripojiť
- Čo sa mi nepodarilo
  - Automaticky spustiť update databázy
  - Použiť CLI príkazy pre dotnet ef v bash dotnet imagu napriek pridaným balíkom pre CLI príkazy

```
ex_18_dockerapp:
 image: ${DOCKER_REGISTRY}ex18dockerapp
 build:
  context: .
  dockerfile: Ex_18_DockerApp/Dockerfile
 depends_on:
   - db
db:
 image: "microsoft/mssql-server-linux"
 environment:
  SA_PASSWORD: Str0ngPassword!
  ACCEPT_EULA: Y
```