Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj prototypu webové aplikace pro evidenci pracovní docházky pomocí technologií ASP.NET Core a cloudových služeb Azure

Gabriela Cimoradská

! ! !

Místo tohoto textu vložte PŘEDNÍ stranu zadání práce, které si můžete vyexportovat do PDF v IS.CZU.cz, pokud již máte schválené zadání i děkanem PEF.

! ! !

! ! !

Místo tohoto textu vložte ZADNÍ stranu zadání práce, které si můžete vyexportovat do PDF v IS.CZU.cz, pokud již máte schválené zadání i děkanem PEF.

V případě, že Vaše zadání je na více než 2 strany, vložte i další strany.

! ! !

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj prototypu webové aplikace pro evidenci pracovní docházky pomocí technologií ASP.NET Core a cloudových služeb Azure" jsem vypracovala samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autorka uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 30.března 2021 _____

Poděkování

Ráda bych touto cestou poděkovala panu Ing. Jiřímu Brožkovi, Ph.D. za odborné konzultace a vedení během diplomové práce. Dále děkuji svému manželovi za veškerou podporu při psaní této práce a osloveným respondentům uživatelských testů za cennou zpětnou vazbu.

Vývoj prototypu webové aplikace pro evidenci pracovní docházky pomocí technologií ASP.NET Core a cloudových služeb Azure

Souhrn

Tato práce pojednává o jednom z mnoha způsobů, jakým lze v dnešní době navrhnout, vyvinout a nasadit webovou aplikaci do cloudového prostředí.

Vlastní webová aplikace je navržena pomocí moderního návrhového vzoru MVC, využívá responsivní design. Funkčně může sloužit k evidenci pracovní docházky v malé až střední firmě.

V první části práce je čtenář seznámen s technologiemi, které byly použity pro vypracování řešení práce, tedy analýzu a rozklad projektu na dílčí pracovní položky, vývoj aplikace a poté její nasazení do cloudu. Jedná se především o technologie z ekosystému společnosti Microsoft, na nichž je postavena velká část celého řešení.

Druhá část se obšírně věnuje vlastnímu řešení a implementaci. Nejprve je provedena stručná analýza a rozklad projektu na dílčí úkoly pomocí nástroje Azure DevOps určeného zejména pro agilní vývoj. Dále je podrobně rozebrán zdrojový kód jednak vlastní aplikace, ale i ukázkových unit testů, které ověřují funkčnost některých metod aplikačního kódu. Implementační část pak uzavírá detailní vysvětlení postupu pro nasazení aplikace, včetně potřebné databáze, do veřejného cloudu Microsoft Azure.

Poslední část práce shrnuje výsledek uživatelského testu, jenž byl proveden několika dobrovolnými respondenty.

Klíčová slova: C#, Microsoft ASP.NET Core, Visual Studio, Azure Cloud, databáze, webová aplikace

Web application prototype development for work attendance registration using ASP. NET Core technologies and Azure cloud services

Abstract

The thesis deals with web application design, development, and deployment to cloud environment.

A custom web application is developed using modern MVC pattern with a responsive design. The main purpose of the application is to record work attendance in a small to a medium-sized company.

In the first part of the document, all the technologies used to implement the solution are introduced, i.e., online services used to prepare project analysis and breakdown into work items, platform and tooling used to develop the application and its deployment to the cloud. These technologies mostly originate from Microsoft development ecosystem.

The second part focuses on solution details and application implementation. First a brief analysis and project decomposition into smaller work items is performed using online service Azure DevOps purposed for agile development. Next the reader is guided through the application source code as well as sample unit test project. The unit tests demonstrate how to verify functionality of selected methods in the application code. The implementation part is concluded by a detailed explanation of the deployment steps which bring the application and backend database to the Azure public cloud.

The final part summarizes results of end user testing which was performed by several voluntary respondents.

Keywords: C#, Microsoft ASP. NET Core, Visual Studio, Azure cloud, database, web application

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V dnešní době plné informačních technologií propojených internetem hrají moderní webové dynamické aplikace zcela nezastupitelnou roli. V mnoha aspektech dokonce nahrazují tradiční nativní desktopové aplikace, které mají horší přenositelnost mezi různými platformami operačních systémů.

Něco podobného také platí o provozu IT systémů v prostředí veřejných cloudových služeb, jako jsou Microsoft Azure, Amazon AWS, nebo Google Cloud. Do nich se ve velkém migrují rozsáhlé instalace infrastruktury ze zastaralých soukromých datových center za účelem modernizace, finančních úspor, zvýšení bezpečnosti a lepší škálovatelnosti.

Jelikož webové aplikace a provoz v cloudu mají rozmanité použití v praxi, kdy navíc jejich rozmach bude v budoucnu téměř s jistotou ještě akcelerovat, stanovila jsem si za cíl poznat prostřednictvím této diplomové práce, jak lze takovou aplikaci navrhnout, vyrobit a nasadit do cloudu.

Ačkoliv prototyp aplikace pro evidenci pracovní docházky v malé až střední firmě je konkrétním vývojářským výsledkem této práce, jednalo se pouze o dobrý prostředek, na kterém jsem si mohla osvojit práci s moderními technologiemi vývojářského ekosystému společnosti Microsoft.

V dalších částech pak shrnu jejich hlavní vlastnosti a vysvětlím, k čemu byly v práci použity.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem diplomové práce je návrh a realizace prototypu funkční webové aplikace pro evidenci pracovní docházky a její nasazení do prostředí veřejného cloudu tak, aby byla zajištěna její vysoká dostupnost a snadná škálovatelnost. Dílčími cíli práce je popsat použité technologie a postupy.

2.2 Metodika

Práce se skládá ze dvou částí – teoretické a praktické. Metodika zpracování teoretické části je založena na studiu odborných informačních zdrojů pro zvolené technologie, které byly využity pro realizaci a s jejich seznámením pro praktické použití. Na základě syntézy zjištěných poznatků budou popsána teoretická východiska pro zpracování praktické části práce.

Praktická část bude spočívat v implementaci prototypu funkční webové aplikace pro evidenci pracovní docházky a jejím nasazení do prostředí veřejného cloudu tak, aby byla zajištěna její vysoká dostupnost a snadná škálovatelnost. Pro implementaci bude využita zejména rodina vývojářských nástrojů z ekosystému firmy Microsoft, jako jsou ASP.NET Core 3.1, PowerShell, Azure DevOps, Microsoft Azure, GitHub, Visual Studio, Visual Studio Code a testovací frameworky pro unit testy.

Nad rámec vlastní implementace, praktická část bude dokumentovat postup celé realizace s ukázkami zdrojového kódu a podrobnějším komentářem dílčích částí. Dále budou vysvětleny kroky pro nasazení aplikace do veřejného cloudu.

Na závěr bude výsledná aplikace uživatelsky otestována a na základě zpětné vazby budou navrženy možnosti jejího případného dalšího rozvoje a úprav.

3 Teoretická východiska a použité technologie

3.1 Přehled použitých technologií

Zásadní vliv na výsledek jakékoliv vývojářské práce má volba pracovních nástrojů a platformy pro vývoj cílových produktů, které nastavují určitý teoretický rámec, v němž se může vývojář pohybovat. Jiný výsledek a pracnost lze očekávat, pokud se rozhodneme použít technologii C++ pro vývoj webové aplikace, anebo pokud vývojář využije moderní cílenou platformu na vývoj webových aplikací, jako jsou např. React.JS, Angular.JS, Node.JS, Python, Ruby nebo třeba právě ASP.NET Core.

Použité technologie v této práci lze shrnout do těchto hlavních kategorií:

1. **Návrh a dokumentace projektu**, kde byly použity zejména online služby:
    - Azure DevOps
    - GitHub

2. **Vývoj aplikace a testování**, kde zdrojový kód celé aplikace byl vyvinut v jazyku C# prostřednictvím těchto platforem, knihoven a nástrojů:
    - .NET Core 3.1, což je platforma, která poskytuje bohatou základní implementaci jazyka C# a celou řadu doplňkových knihoven
    - ASP.NET Core framework, tedy rozšiřujícího souboru knihoven cílených na vývoj webových aplikací a API
    - XUnit, knihovna pro tvorbu unit testů
    - Moq, knihovna pro účely simulace složitějších objektů a závislostí při tvorbě unit testů
    - Visual Studio 2019 Community Edition, IDE (Integrated Development Environment) pro tvorbu zdrojového kódu a ladění aplikace

3. **Nasazení aplikace do cloudu.** jež proběhlo zejména pomocí skriptovacího jazyka PowerShell Core 7.0, resp. Az.RM modulu, který nabízí bohaté API pro práci s Microsoft Azure, automatizaci a konfiguraci lokálních i vzdálených systémů. PowerShell skripty pak byly vyvinuty pomocí universálního IDE Visual Studio

Code, v němž se velmi dobře pracuje mj. právě i s PowerShellem, díky vestavěným PowerShell rozšířením.

Vlastní aplikace využívá pro svůj běh následující cloudové služby:

- Azure App Service pro hostování webových aplikací a API
- Azure SQL Database pro hostování backend databáze

V následujících kapitolách budou uvedené technologie a služby podrobněji popsány.

3.2 .NET Core

.NET Core je open-source vývojová platforma pro tvorbu širokého spektra aplikací s rozsáhlým souborem knihoven, jenž lze sdílet mezi všemi druhy aplikací. Hlavní výčet nejběžnějších typů aplikací, které lze na této platformě vyvinout zahrnuje tyto [1]:

- Webové aplikace, API a microservices ASP.NET Core
- Cloudové aplikace
- Mobilní aplikace
- Desktopové aplikace, služby a hry
- Machine Learning (ML, aplikace se strojovým učením), Artificial Intelligence (AI, aplikace s umělou inteligencí) a Internet of Things (IOT, Internet chytrých zařízení) aplikace

Další důležitou vlastností .NET Core frameworku je skutečnost, že se jedná o cross-platform technologii, v níž lze vyvíjet aplikace pro celou řadu operačních systémů [1]:

- Windows
- macOS
- Linux
- Android
- iOS
- tvOS
- watchOS

.NET Core dále podporuje kompilaci artefaktů pro všechny hlavní procesorové architektury dnešní doby [1]:

- x64
- x86
- ARM32
- ARM64

Aplikace s podporou .NET platformy lze vyvíjet třemi podporovanými programovacími jazyky [1]:

- C#, jedná se o objektově orientovaný, typově silný jazyk, který má své kořeny v rodině jazyků na bázi C.
- F#, což je moderní funkcionální objektově orientovaný, typově silný jazyk, který se zaměřuje na definici typů a funkcí zejména pomocí výrazů
- Visual Basic, jazyk odvozený od staršího Basicu, který je charakteristický svým snadným porozuměním a podobností s lidskou řečí, která jej činí snadno pochopitelným i pro začínající vývojáře. Na druhou stranu je tento jazyk již na svém ústupu a nelze v něm vytvářet všechny výše uvedené typy aplikací, např. webové aplikace.

Pro vývoj v .NET Core, mohou vývojáři používat několik různých vývojových prostředí (IDE), jako jsou např.

- Visual Studio
- Visual Studio Code
- GitHub Codespaces (online prostředí)
- Rider od firmy JetBrains

3.2.1 **CLR (Common Language Runtime)**

.NET Core aplikace běží v prostředí, které se nazývá jako Common Language Runtime (CLR). .NET Core CLR je cross-platform runtime prostředí pro OS Windows, macOS a Linux. CLR je v podstatě virtuální stroj, ve kterém dochází ke kompilaci tzv. Intermediate Language (IL) do strojového kódu specifického pro daný procesor pomocí JIT (Just-In-Time) kompilátoru, jenž je součástí CLR a je specifický pro daný HW. Kód v

Intermediate Language, tedy souhrn instrukcí, které jsou nezávislé na HW platformě, a je výsledkem prvotní kompilace C#, VB nebo F# zdrojových kódů, lze získat kompilací původních zdrojových kódů pomocí nástroje MSBuild [1].

Důležitou vlastností CLR prostředí je automatická správa paměti prostřednictvím služby Garbage Collector (GC). Ta alokuje paměť pro nové objekty aplikace tak dlouho, dokud je místo v přidělené paměti (managed heap). Pokud již volné místo nezbývá, GC zkontroluje všechny existující alokované objekty v paměti z pohledu aplikace, tj. jestli jsou stále ještě využívány běžícím kódem, a pokud nejsou, jejich paměť je následně uvolněna [1].

Za zmínku stojí zmínit, že analogický princip pro kompilaci a běh kódu v runtime prostředí virtuálního stroje využívá také jazyk Java.

3.2.2 .NET Core a .NET 5

Původní .NET Framework byl poprvé vydán v roce 2002 ve verzi 1.0. Od té doby byla vydána cela řada nových verzí této .NET implementace. Aktuální a poslední verze je .NET Framework 4.8. a je primárně určena k vývoj aplikací pod OS Windows [2].

V roce 2014, Microsoft zahájil vývoj nové otevřené (open-source) cross-platform .NET implementace, která by nahradila původní .NET Framework. Tato nová implementace dostala jméno .NET Core a verze 1.0 byla vydána v červnu 2016. Dále pak následovaly další verze až do verze 3.1. Samotné označení .NET Core se používá až do verze 3.1, nicméně další nová verze po verzi 3.1 se již označuje pouze jako .NET 5.0. Verze 4.0 byla úmyslně přeskočena, aby nedošlo k záměně se staršími verzemi původního .NET Frameworku 4.x. Došlo také k vypuštění označení „Core“, ačkoliv .NET 5 je bez pochyb další verze této cross-platform .NET implementace, která je v dnešní době hlavní a do budoucna nejperspektivnější implementace platformy .NET [1].

3.2.3 Nasazení aplikace vyvinuté v .NET Core

.NET aplikaci je možné publikovat a distribuovat dvěma způsoby [1], tj. jako:

- Instalační balík, který obsahuje spustitelný EXE soubor včetně potřebných knihoven pro běh aplikace. Součástí balíku je potom také potřebné CLR prostředí. Takový instalační balík vždy adresuje konkrétní HW platformu a OS.

- Instalační balík, který obsahuje spustitelný EXE soubor včetně potřebných knihoven pro běh aplikace. Na cílovém stroji, kde aplikace poběží však musí být již předinstalováno potřebné CLR prostředí pro běh .NET aplikace. Spustitelný soubor je specifický pro cílovou HW platformu a OS, nicméně DLL knihovny jsou vždy již cross-platform.

3.2.4 .NET standard

.NET aplikace je zpravidla vyvinuta a běží v jedné konkrétní .NET implementaci. Existuje však celá řada .NET implementací, např. .NET Framework, .NET 5, .NET Core 3.1, Mono, Xamarin, Unity. Všechny tyto různé implementace .NET zpravidla spojuje podpora specifikace .NET standard.

.NET standard je specifikace minimálních rozhraní (API), které musí být implementovány základním souborem knihoven (Base Class Library) v rámci dané implementace .NET, jako jsou např. .NET Core, .NET Framework, Mono, aj., aby daná implementace mohla deklarovat shodu s patřičným .NET standardem [3].

Hlavním důvodem pro vznik a existenci .NET standardu coby univerzální specifikace základních rozhraní jazyka, je přenositelnost kódu mezi jednotlivým implementacemi .NET.

Každá implementace .NET obsahuje tyto základní komponenty [3]:

- Jedno nebo více runtime prostředí (CLR) pro různé platformy, myšleno operační systémy a procesorové architektury. Příkladem může být .NET Framework CLR, .NET Core 3.1 CLR, apod.
- Základní knihovnu tříd (Base Class Library), kde příkladem opět může být .NET Framework Base Class Library, .NET 3.1 Core Base Class Library, atd.
- Další doplňkové tzv. aplikační frameworky pro vývoj specifických typů aplikací, např. ASP.NET, Windows Forms, Windows Presentation Foundation (WPF).
- Další doplňkové nástroje pro vývoj a kompilaci kódu, např. MSBuild, Nuget, dotnetCLI.

Nejznámější .NET implementace podporované firmou Microsoft jsou tyto [4]:

- .NET Core a .NET 5
- .NET Framework
- Mono
- UWP (Universal Windows Platform)

V současné době je .NET 5 nejnovější dostupná implementace .NET, která je následníkem .NET Core 3.x a .NET Core 2.x. Tato práce byla vyvinuta pomocí .NET Core 3.1, který byl v době jeho zahájení práce poslední aktuální verzí.

.NET Framework je původní originální implementace .NET, která vznikla už v roce 2002. Počínaje verzí 4.5 a výše, .NET Framework začal implementovat a dodržovat .NET Standard specifikace, což znamená, že kód, který cílí na tyto specifikace může běžet na těchto verzích .NET Frameworku. Na rozdíl od .NET Core a .NET 5, .NET Framework není ještě cross-platform a lze ho využít pouze k vývoji a nasazení aplikací v prostředí OS Windows [4].

Mono, je .NET implementace, která je určena pro prostředí, kde je kladen důraz na malý a odlehčený runtime a podporuje všechny dostupné verze .NET Standardu. Je primárně určen pro běh Xamarin aplikací na platformách Android, macOS, iOS, tvOS a watchOS. Mono je také využíván jako runtime prostředí pro běh her vyvinutých na platformě Unity [4].

UWP je .NET implementace, která umožňuje vyvíjet jednotné vzhledově unifikované aplikace pro různé druhy zařízení jako jsou PC, tablety, smartphony, IOT zařízení, nebo herní konzole [4].

Jak už bylo zmíněno, různé .NET implementace deklarují shodu s .NET Standardem, který podporují a jsou s ním kompatibilní. Podpora určité verze .NET Standardu v dané verzi .NET implementace znamená, že podporuje všechny nižší verze .NET Standardu včetně uvedené verze, což přehledně shrnuje Tabulka 1 [3].

Tabulka 1 - Podpora .NET Standardu v různých implementacích .NET

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1	n/a
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.MacOS	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
UWP	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	n/a
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	n/a

Všechny verze .NET standardu jsou publikovány ve veřejném repositáři GitHub: https://github.com/dotnet/standard/tree/master/docs/versions a jsou udržovány zejména firmami Microsoft a Unity.

3.2.5 Číslování verzí .NET Core a .NET 5

Verze .NET Core runtime prostředí se skládá ze tří čísel, tj. <MAJOR>.<MINOR>.<PATCH>
Tak např. .NET Core **1**.0, **2**.0, **3**.0 a .NET **5**.0 vymezují čtyři různé major verze .NET Core implementace.

Dále .NET Core 3.**1** je první minor verze runtime, která byla vydána pro .NET Core 3.0 major verzi a .NET Core 3.1.**7** je sedmá patch verze runtime prostředí .NET Core 3.1 implementace [5].

Major verze se zpravidla vyznačuje tím, že zavádí nové funkce a nová veřejná rozhraní (API). Kromě toho také obvykle přináší opravu chyb v původních API. Tyto změny jsou zpravidla nekompatibilní s předchozí verzí .NET, a označují se jako nekompatibilní změny (breaking changes), což je právě důvodem pro vydání nové major verze .NET. Na jednom stroji mohou být nainstalovány paralelně vedle sebe dvě a více major verzí runtime dané .NET implementace [5].

Minor verze také přináší nové funkce, mění veřejná API a zahrnuje opravy chyb. Může obsahovat také nekompatibilní změny (breaking changes), které vyžadují změny ve zdrojovém kódu aplikace. Rozdíl mezi Major a Minor verzí je ten, že Minor verze zahrnuje podstatně menší množství těch změn, které ovlivňují kompatibilitu verze, tzn. že úprava

kódu např. z verze .NET Core 3.0 na 3.1 je jednodušší a méně pracná než při změně major verze. Také minor runtime verze lze instalovat na jeden systém paralelně vedle sebe [5].

Servisní aktualizace (patche) jsou vydávány pro podporované verze téměř každý měsíc, obvykle v druhé úterý měsíce. Zahrnují jak bezpečnostní, tak funkční opravy chyb. .NET Core 3.1.8 je příklad osmého bezpečnostního patche pro .NET Core 3.1 runtime. Tyto servisní aktualizace jsou plně zpětně kompatibilní s danou minor verzí, pro který jsou vydány. Instalace nové servisní aktualizace vždy nahrazuje předchozí starší verzi runtime na daném cílovém systému [5].

.NET SDK používá trochu jiný koncept pro své číslování, který se označuje jako Feature Bands, kde např. verze .NET Core SDK 3.1.100 až 3.1.199 vymezují jeden Feature Band, tedy Feature Band .NET Core SDK 3.1.1xx.

Novější verze v rámci jednoho Feature Bandu pak zcela nahrazují starší verzi. Vyšší Feature Band, např. .NET Core SDK 3.1.2xx, se však již nainstaluje na stejném systému paralelně k existujícímu Feature Bandu, např. tedy k .NET SDK 3.1.1xx, a oba tak mohou spolu koexistovat [5].

3.2.6 Životní cyklus a aktualizace verzí .NET Core

.NET Core a .NET 5.0 se řídí novým kalendářem podpory a aktualizací. Originální .NET Framework používal podporu v režimu 5+5, tj. 5 let hlavní podpory a 5 let prodloužené podpory. V období hlavní podpory byly vydávány opravy jak funkčních, tak bezpečnostních chyb, ale v prodloužené podpoře byly vydávány už jen opravy bezpečnostních chyb [5].

Naproti tomu, stávající model podpory je založen na vyšší kadenci nových verzí a kratší době podpory starších verzí. V tomto modelu momentálně existuji dva hlavní podporované tracky [5]:

- Kontinuální podpora poslední stávající verze, což znamená, že předchozí verze je podporována ještě 3 měsíce po vydání následující nové major nebo minor verze. Příkladem může být:
    - .NET Core 3.0 byl vydán v září 2019 a byl následován .NET Core 3.1 v prosinci 2019, což znamená, že podpora verze 3.0 skončila v březnu 2020, tedy 3 měsíce poté, co byla vydána verze 3.1 [5].

- Long Term Release (LTS) verze je podporována nejméně po dobu 3 let, nebo alespoň ještě 1 rok po vydání následující LTS verze, pokud je toto datum později. V praxi to znamená:
    - .NET Core 2.1 byl vydán v květnu 2018 a byl prohlášen za LTS verzi v srpnu 2018
    - .NET Core 3.1, jakožto novější LTS verze, byla vydána v prosinci 2019
    - Jelikož srpen 2021, tedy 3 roky od verze 2.1, je později než prosinec 2020, tedy 1 rok po vydání verze 3.1, bude .NET Core 2.1 podporován ještě až do srpna 2021 [5].

Ukončení podpory pro danou verze .NET Core znamená, že Microsoft přestává vydávat funkční a bezpečnostní opravy a není dostupná ani jiná technická asistence. Vývojáři by se tedy měli snažit o to, aby své aplikace včas aktualizovali a provozovali na novější podporované verzi, nejlépe verzích LTS, a to zejména z důvodu maximální bezpečnosti svých aplikací [5].

3.3 ASP.NET Core

ASP.NET Core je cross-platform, otevřený (open-source) aplikační framework pro vývoj internetových aplikací běžících jak v cloudové, tak privátní infrastruktuře.

Předchůdcem ASP.NET Core je ASP.NET 4.x Framework, který vyžaduje pro svůj běh původní .NET Framework 4.x určený pro OS Windows. ASP.NET 4.x aplikace lze tedy provozovat pouze na OS Windows.

ASP.NET Core je pak inovovaná a přepracovaná verze původního ASP.NET, která přináší několik zásadních výhod a novinek [6]:

- Unifikovaný návrh webových uživatelských rozhraní (UI, User Interface) a API.
- Vylepšená podpora testovatelnosti.
- Podpora Razor Pages, coby alternativního způsobu pro tvorbu webových aplikací, který je z pohledu kódu jednodušší než tradiční vzor MVC (Model-View-Controller).

- Podpora Blazor frameworku, který umožňuje vytvářet moderní client-side web aplikace pomocí jazyka .NET C#.
- Možnost vyvíjet a provozovat aplikace pro OS Windows, macOS a Linux.
- Má otevřený zdrojový kód (Open-source).
- Možnost hostovat aplikaci na celé řadě webových serverů (Kestrel, IIS, HTTP.sys, Nginx, Apache, Docker).
- Podpora souběžného běhu a paralelní instalace různých major a minor verzí .NET Core runtime na jednom serveru.

ASP.NET Core má dále tyto vlastnosti, optimalizované pro vývoj webových aplikací a API [6]:

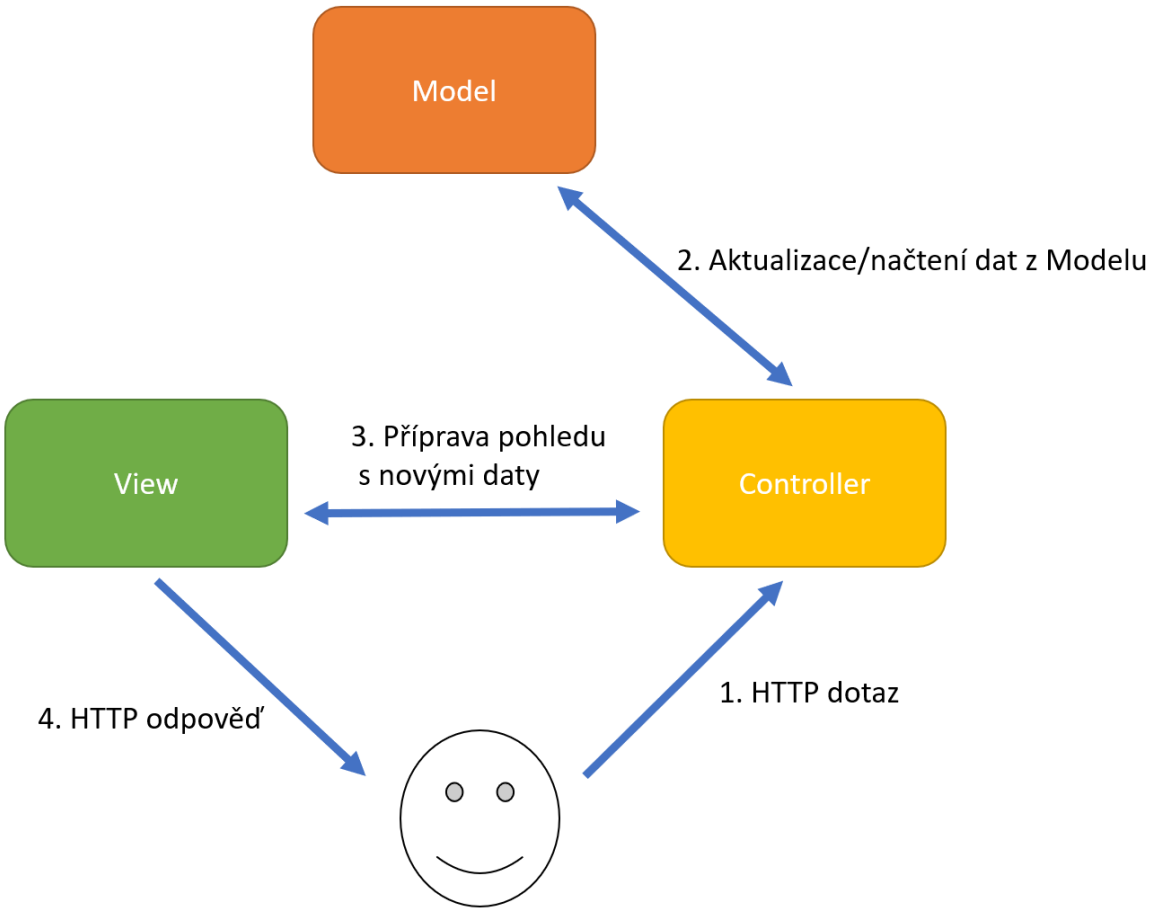
- Razor je v podstatě jazyková syntaxe, která kombinuje značkovací jazyk Razor markup, C# a HTML. Soubory obsahující Razor kód mají příponu *.cshtml nebo *.razor.
- Speciální pomocné značky (Tag Helpers), které umožňují na straně serveru vytvářet a ovlivňovat finální podobu HTML kódu, který se generuje (renderuje) pomocí Razor souborů. Tyto značky jsou kódovány v C# a cílí vždy na specifický HTML element na základě jména elementu, atributu nebo rodičovského tagu. Pomáhají tak generovat finální podobou HTML kódu.
- Automatický model binding (vazba), který transformuje data z HTTP requestu (dotazu) do parametrů volané metody Controlleru.
- Automatická validace modelu, jež zajišťuje validaci dat, jak na straně webového klienta, tak na straně serveru.
- Úzká integrace s client-side frameworky a knihovnami jako jsou Blazor, Angular, React nebo Bootstrap.

3.3.1 **MVC**

ASP.NET Core je vhodný framework pro tvorbu webových aplikací pomocí návrhového zdroje Model-View-Controller (Model-Pohled-Řadič). Tento návrhový vzor je také využit pro realizaci této práce. Byl vybrán z důvodu častého použití při vývoji firemních aplikací a webových API, a také z důvodu, aby mohla být vyzkoušena a demonstrována testovatelnost kódu.

MVC je obecný architektonický návrhový zdroj, který rozděluje aplikaci do tří hlavních skupin komponent: Model, View a Controller. Tento návrhový vzor od sebe pomáhá izolovat dílčí komponenty kódu tak, aby byly lépe udržovatelné, škálovatelné a testovatelné. Základní logika spočívá v tom, že uživatelské HTTP dotazy (requesty) jsou směrovány do daného Controlleru na základě URL, který je pak zodpovědný za interakci s datovým Modelem a provedení požadované akce obsažené v dotazu, případně za načtení požadovaných dat z Modelu. Následně Controller vybere vhodné View pro zobrazení výsledku, kterému předá dříve získaná modelová data [7].

Obrázek 1 - Schématické zobrazení činnosti aplikace podle MVC vzoru při zpracování HTTP dotazu



Toto rozdělení na tři hlavní komponenty pomáhá lépe udržovat celou aplikaci a provádět dílčí úpravy např. jen ve View, aniž by bylo nutné upravit ostatní dvě komponenty. Technicky vzato, jsou jak View tak Controller závislé na Modelu, avšak Model nezávisí na žádné z nich. To tedy umožňuje udržovat Model zcela nezávisle na prezentační vrstvě (View). Činnost modelu také ukazuje Obrázek 1 [7].

3.3.1.1 **Model**

Model v MVC aplikaci představuje datový stav aplikace a měla by se v něm odehrávat hlavní část business logiky. Business logika by měla být zahrnuta do modelu současně s veškerou logikou pro persistenci dat do databáze [7].

3.3.1.2 **View**

View se používá pro prezentování požadovaného obsahu přes uživatelské rozhraní, a tudíž generuje webovou stránku. V případě ASP.NET Core se View definuje pomocí Razor souborů, které kombinují .NET C# kód a HTML značkovací jazyk. View by obecně mělo obsahovat minimální množství logiky, která je nezbytná pouze pro zobrazení potřebného obsahu [7].

3.3.1.3 **Controller**

Controller přijímá a zpracovává HTTP dotazy, pracuje s Modelem a určuje jaké View bude zobrazeno v odpovědi na HTTP dotaz klienta. V rámci MVC vzoru, Controller je komponenta, která zpracovává veškerá vstupní data a odpovídá prostřednictvím View na jednotlivé interakce. Je to vstupní bod do aplikace, který určuje, s jakým modelem se bude pracovat a jaké View připravit pro odpověď. Obecně platí, že by Controller neměl být příliš složitý z hlediska aplikační logiky, která by spíše měla být obsažena právě v Modelu [7].

3.3.2 **ASP.NET Core Identity**

Klíčovou komponentou pro vývoj této aplikace je knihovna ASP.NET Core Identity. Jedná se o API, které poskytuje uživatelské rozhraní pro registraci, přihlášení a řízení uživatelů i jejich rolí. API má funkce nezbytné pro vytváření nových identit uživatele, jejich následné přihlášení do aplikace, správu hesel a profilových dat, řízení rolí, uživatelských oprávnění, ale i např. rozhraní pro dokončení uživatelské registrace skrz potvrzující email.

Knihovna Identity umožňuje jak správu vlastních uživatelských identit uložených v backendu aplikace, tak integraci s externími poskytovateli identity, jako jsou Facebook, Google, Microsoft nebo Twitter [8].

3.4 Entity Framework Core

Entity Framework (EF) Core je následníkem Entity Frameworku 6, který se primárně využíval v původním .NET Frameworku, ale dále se už nerozvíjí.

EF Core je otevřený (open-source) a cross-platform soubor knihoven pro práci s daty v relačních databázích a jejich mapování do objektového světa .NET Core, tedy tzv. objekt relační mapování (O/RM) [9].

EF Core podporuje celou řadu databází, jako jsou SQL Server, SQL Azure Database, SQL Lite, Azure Cosmos DB, MySQL, PostgreSQL, aj.

Dále také podporuje LINQ dotazy jazyka C#, sledování změn v datech modelu při běhu aplikace a migraci DB schématu podle potřeb aktuálního modelu [10].

Hlavní předností použití EF Core je možnost pracovat s daty v databází pomocí .NET objektů, což eliminuje potřebu psát vlastní SQL dotazy, které by přímo přistupovaly do databáze [9].

Základem práce s EF Core je vytvoření modelu, který se skládá z tzv. entitních tříd, do nichž se mapují data z databáze. Další nezbytnou součástí nastavení EF je objekt objektového contextu, který reprezentuje propojení do databáze.

EF Core umožňuje práci s modelem na základě [9]:

- Ručně připraveného modelu, který bude odpovídat záznamům v databázi.
- Vygenerovaného modelu z existující databáze.
- Ručně předpřipraveného modelu, který bude použit pro tvorbu a migraci do nové databáze.

V této práci je použit pro O/RM mapování EF Core verze 3.1. Model je předpřipraven ručně a migrován do databáze.

3.5 Unit testování

Za unit testování se označuje softwarová praktika, která se používá k vývoji automatizovaných testů, jež testují funkčnost nějaké minimální části kódu (unit), což je obvykle metoda v rámci třídy. Test je založen na tom, že testovaný kód se chová deterministicky a pro určitý vstup, který testovaný kód zpracuje známe, jaký by měl být vygenerovat výstup. Díky znalosti očekávaného výstupu a na základě přesně daného vstupu jsme schopni určit a porovnat, jestli testovaný kód funguje správně [11].

Unit testy tak pomáhají snížit množství chyb pří následných úpravách kódu, protože pokud vývojář způsobí svojí změnou logickou chybu v části kódu, která je pokrytá unit testem, tak je tato chyba okamžitě odhalena ještě ve fázi přípravy artefaktu, kdy jsou unit testy obvykle spouštěny. Vysoká míra pokrytí kódu unit testy nám usnadňuje údržbu kódu, snižuje celkovou pracnost při následném ručním testování a množství výskytu regresí (defekty v kódu způsobené v důsledku změny kódu) [12].

Aby byl kód dobře testovatelný, musí být správně organizován do malých funkčních bloků, pro které se snadno tvoří testy, spíš než do velkých bloků, kde se tvorba unit testů stává obtížná.

Technicky jsou unit testy v zásadě jen další naprogramované třídy a metody, nebo funkce, které porovnávají výstup získaný spuštěním testovaného kódu s očekáváným výstupem, který známe na základě známého vstupu. Jedním z dílčích úkolů během příprav unit testů je schopnost izolovat testovaný kód pro účely testování.

Unit testy se obvykle spouští automaticky po dokončení buildu a běží v paměti.

Každý unit test má zpravidla tyto tři dílčí části:

- **Arrange** – inicializace testovacích objektů a závislostí, příprava prostředí na test.
- **Act** – běh vlastního kódu, který je předmětem testování (Subject Under Test).
- **Assert** – kontrola a porovnání aktuálního výstupu s očekáváním.

Mezi další charakteristiky unit testů patří [12]:

- **Rychlost** - díky svému běhu v paměti jsou unit testy velmi rychlé, takže i projekty, které mají tisíce unit testů se ve výsledku velmi rychle otestují.
- **Izolovanost** – mohou běžet v izolaci na jakémkoliv stroji bez nutnosti se připojovat k vnějším systémům, jako jsou externí servery nebo databáze.
- **Opakovatelnost** – výsledek testu by měly být konzistentní. Jeho opakované běhy by měly vracet stále stejný výsledek, pokud nedošlo k žádné změně kódu.
- **Automatická detekce výsledků** – unit testy není potřeba ručně vyhodnocovat. Veškeré vyhodnocení výsledků probíhá automaticky v části Assert.

- **Časová přiměřenost** – z pohledu vývoje testu by nemělo trvat napsat test příliš dlouho. Pokud tomu tak je, může to indikovat, že testovaný kód je příliš komplexní a je potřeba ho rozdělit na menší funkční bloky.
- **Míra pokrytí aplikačního kódu testy** – označuje, jak velké množství aplikační logiky je pokryto testy. V praxi se pohybuje okolo 40-90 %.

Pro účely tvorby unit testů v této práci byl vybrán projekt xUnit.net (https://xunit.net), což je volně dostupný, otevřený (open-source) soubor knihoven k tvorbě unit testů pro kód vyvíjený v .NET Frameworku, .NET Core a jiných .NET implementacích.

Dále je pro tvorbu unit testů v práci také použita knihovna Moq (https://github.com/Moq/moq4/wiki/Quickstart), která se používá pro tvorbu falešných objektů, které simulují vnější závislosti (stub), nebo pro simulaci objektů (mock), jež se používají při vyhodnocení testů.

3.6 PowerShell

PowerShell je robustní cross-platform framework pro automatizaci a řízení konfigurace systému, který je tvořen příkazovým interpretem a skriptovacím jazykem.

Pro svůj běh využívá .NET CLR, což znamená, že umí pracovat s objekty na svém vstupu i výstupu na rozdíl od jiných skriptovacích jazyků, jež pracují na vstupu i výstupu obvykle pouze s textem. Právě schopnost práce s objekty činí z PowerShellu velmi mocný programovací nástroj, ve kterém lze automatizovat téměř cokoliv a využít k tomu většinu API dostupných v rámci .NET frameworku [13].

Příkazy v PowerShellu se nazývají CommandLets (zkráceně cmdlets). Tyto příkazy lze vzájemně řetězit prostřednictvím pipelines (znak |), takže si navzájem po jednom předávají objekty z výstupu jednoho příkazu na vstup druhého, a umožňují tak realizovat komplexní operace. PowerShell zprostředkovává snadný přístup nejen k souborovému systému, ale i k jiným datovým uložištím, jako jsou např. Windows Registry, uložiště certifikátů, nebo vzdálené systémy.

Rodina standardních příkazů PowerShellu je snadno rozšiřitelná pomocí modulů, které mohou obsahovat nové funkce napsané opět v PowerShellu, nebo již zkompilovaný kód, jenž zavádí nové cmdlets do příkazové řádky po importování modulu [13].

PowerShell má také velmi dobře zpracovanou podporu aliasů, což znamená, že tentýž cmdlet lze zavolat pomocí několika různých aliasů, které jsou obvykle známy z jiných starších příkazových interpretů (Bash, Cmd, apod.). To usnadňuje vývojářům práci s PowerShellem při přechodu z ostatních interpretů. Příkladem může být práce s příkazem `Clear-Host`, který se používá k vymazání obrazovky terminálu. Tento příkaz lze zavolat také aliasy `cls` nebo `clear`, které PowerShell interpretuje stejně jako originální cmdlet `Clear-Host`. Podstatnou výhodou PowerShell aliasu je fakt, že standardní aliasy nejsou definovány dodatečně na úrovni interpretu uživatelem, jako to bývá zvykem jinde, ale zavádí se současně s importem modulu obsahující cmdlet, čímž se podstatně redukuje potřeba jejich správy a uživatel k nim přichází automaticky [13].

Jak již bylo zmíněno, klíčovou roli v PowerShellu hraje funkce řetězení příkazů pomocí pipeline (znak |). Každý příkaz v řetězení předává svůj objektový výstup sériově, objekt za objektem, následujícímu příkazu. Příkazy v řetězení tak nemusí zpracovávat víc než jeden objekt najednou, což je také výhoda proti starším interpretům. To celkově snižuje spotřebu zdrojů a zvyšuje rychlost odezvy jednotlivých operací [13].

PowerShell má propracovaný systém nápovědy, který je srovnatelný např. s Linux `man` systémem. V případě PowerShellu se nápověda aktivuje příkazem `Get-Help [13].`

Za zmínku ještě stojí říci, že dnes existují dvě distribuce PowerShellu. Starší se označuje jako Windows PowerShell 5.1 a je standardní součástí operačního systému Windows, kde zajišťuje nezbytnou výbavu pro příkazovou řádku.

Novější verze se označuje pouze jako PowerShell (dříve ještě PowerShell Core), je cross-platform a je nejnovější verzí tohoto příkazového interpretu. Aktuálně existuje ve verzi 7.1 a lze ji provozovat na OS Windows, Linux a macOS [14].

3.6.1 **PowerShell Az modul**

Az PowerShell modul je rozšiřující soubor příkazů (cmdlets) pro práci a správu zdrojů ve veřejném cloudu Microsoft Azure. Funguje, jak s původním Windows PowerShellem 5.1 tak s novějším PowerShell (Core) od verze 7.0. Az PowerShell modul podporuje všechny dostupné Azure služby, což znamená, že pro ně lze v modulu nalézt odpovídající příkazy k jejich správě a konfiguraci [15].

V rámci této diplomové práce byl použit Az PowerShell modul verze 5.4.0 a PowerShell 7.0.4. S jejich pomocí byly vytvořeny cloudové služby Azure SQL Database a Azure App Service, a do nich následně nasazena aplikace a migrován datový model.

3.7 Git

Pro správu zdrojového kódu v rámci této diplomové práce je použit systém Git 2.16.1.windows.1. Git jakožto nástroj pro správu zdrojových kódů (Source Control Management SCM) byl vytvořen Linus Torvaldsem už v roce 2005, aby vylepšil kolaborativní práci mezi vývojáři při vývoji Linuxového jádra [16].

Jde o otevřený distribuovaný systém (open-source) pro správu zdrojových kódů a evidenci jejich změn (verzování) během vývoje. Je vhodný zejména pro textové soubory, nicméně nevhodný pro binární soubory. Git lze použít pro malé i velké projekty, kde vyniká svojí spolehlivostí, rychlostí a efektivitou. V dnešní době se patrně jedná o jeden z nejpopulárnějších a nejrozšířenějších systémů pro správu zdrojových kódů. Díky Gitu mohou vývojáři snadno koordinovat svoji práci a pracovat společně na stejném projektu, jelikož každý má k dispozici plnou lokální kopii repositáře zdrojových kódů, včetně historie všech změn, zatímco na vzdáleném Git serveru existuje primární zdroj repositáře. Vývojář pak pracuje na své lokální větvi (git branch), kterou pravidelně synchronizuje s hlavní vývojovou větví, např. master, main nebo develop, a kterou si naopak ostatní vývojáři pravidelně replikují na svůj lokální stroj. Tím dochází k časté synchronizaci zdrojových kódů mezi všemi vývojáři a Git serverem, což vede k omezení vzniku změnových konfliktů, tedy navzájem neslučitelných změn, např. v jednom souboru. [16].

3.8 GitHub

Pro publikaci zdrojového kódu této práce byla použita online služba GitHub. Je to veřejná služba, která je postavená na Git SCM systému a slouží k ukládání i publikování veřejných nebo privátních Git repositářů. Kromě této základní služby zdarma, je vývojářům a vývojovým teamům dostupná celá řada dalších doprovodných placených služeb, jako jsou nástroje pro agilní programování, tj. plánování práce a evidenci pracovních tabulí (board), podpora evidence a sledování defektů v kódu (bugy), vedení dokumentace, automatizovanou kompilaci (build) zdrojového kódu a nasazení artefaktů do cílové infrastruktury, a mnoho dalšího, co dnešní DevOps teamy ke své práci využívají.

Zdrojové kódy webové aplikace vyvíjené v rámci této práce, stejně jako PowerShell skript použitý pro nasazení aplikace do Azure cloudu, jsou uloženy v GitHub repositáři: https://github.com/xsubg001/dp2020 ve větvi master.

3.9 Azure DevOps

Pro analýzu hlavních úkolů této diplomové práce a podrobný rozpad celého pensa práce na podrobnější pracovní úkoly ve stylu agilního programování byla použita online služba Azure DevOps.

Podobně jako GitHub, je Azure DevOps také platforma nabízející soubor služeb zaměřený na vývojářské teamy a komunity, které vyvíjí a spravují softwarové projekty. Umožňuje teamům plánovat jejich práci, spolupracovat vývojářům, program manažerům a dalším přispěvatelům na vývoji zdrojových kódů a zadaní projektu, dále umožňuje automatizovanou kompilaci (build) a nasazení artefaktů do privátní nebo veřejné cloudové infrastruktury.

Stručný seznam služeb, které jsou dostupné v Azure DevOps platformě zahrnuje tyto [17]:

- **Azure Repos** – základní služba pro tvorbu a správu repositářů zdrojového kódu pomoci Gitu nebo TFVC (Team Foundation Version Control), což je centralizovaný systém správy zdrojových kódů vyvinutý firmou Microsoft.
- **Azure Pipelines** – služba která umožňuje automatizaci kompilace (build), testování a nasazení artefaktů do cílové infrastruktury.
- **Azure Boards** – služba, která umožňuje online plánování vývojářské práce, sledování jejího vývoje a nezbytnou podporu pro evidenci defektů v kódu. To vše prostřednictvím vizualizace na principu agilních metod Kanban nebo Scrum.
- **Azure Test Plans** – je soubor nástrojů, které pomáhají s automatizací testů aplikace, včetně možnosti vytvářet manuální test plány a sledovat jejich výsledky.
- **Azure Artifacts** – umožňuje správu a archivaci artefaktů, které jsou zkompilovány jak z vlastních zdrojových kódů, tak přejaty z jiných veřejných zdrojů. Podporovány jsou všechny obvyklé formáty balíků, jako jsou Maven, npm, Nuget, apod.

V rámci této práce byla použita především služba Azure Boards, ve které byly dokumentovány jednotlivé Features, User Stories a dílčí úkoly (Tasks) pro dokončení

projektu, stanové během úvodní analýzy, stejně tak, jako programové defekty (bugy) nalezené během vývoje.

3.10 Azure App Service

Pro hostování webové aplikace v prostředí veřejného cloudu je v rámci této práce použita služba Azure App Service. Jedná se o globální službu, která umožňuje hostování webových aplikací a API pomocí protokolu HTTP/HTTPS, bez nutnosti starat se o infrastrukturu, ve které aplikace běží. Služba je prakticky dostupná ve všech Azure regionech [18]. Jedná se o tzv. PaaS službu, tedy Platform as a Service. Vývojář si podle potřeby zvolí velikost diskového prostoru a výkon serverové hostující infrastruktury. V základní škále tarifů služby lze najít i takové, které jsou prakticky zdarma a postačí pro běžný testovací provoz a vývoj aplikace. V případě potřeby lze pak službu výkonově rozšířit na vyšší výkonový tarif. Infrastruktura služby je automaticky udržována z pohledu aktualizací OS, zabezpečení a aktualizace runtime prostředí v rámci služby Azure, takže se o to vývojář nemusí sám starat [19].

Do prostředí služby App Service lze nasadit celou řadu aplikací z různých zdrojů a vyvinutých na různých platformách, jako jsou .NET, .NET Core, Java, Ruby, PHP, Node.JS, Python, nebo také vlastní Docker kontejnery.

Jedná se o cross-platform službu, tzn. že si vývojář může zvolit, jestli chce aby runtime prostředí bylo hostováno na serverech s OS Windows nebo Linux.

Kromě velmi dobré podpory různého druhu aplikací a velké škály runtime prostředí, ve kterých může kód aplikace běžet, je Azure App Service integrována s celou řadou dalších doprovodných Azure služeb. Patří sem např. Azure Load Balancer, Azure Log Analytics, App Service autoscaling (automatické škálování dle zátěže), Automated Configuration Management (správa automatické konfigurace prostředí) nebo integrace s Azure databázovými službami [19].

Mezi další pokročilé vlastnosti patří integrace s online službami pro správu zdrojových kódů (SCM), jako jsou Azure DevOps, GitHub, DockerHub, apod. [19]. Integrace s GitHub je využita i v rámci této práce.

3.11 **Azure SQL**

Jako backend databáze pro vývoj této aplikace byla zvolena MS SQL databáze, která v je rámci cloudového nasazení aplikace representována službou Azure SQL.

Azure SQL je velmi rozsáhlá rodina služeb na bázi produktu MS SQL Server, která má celou řadu forem a dílčích služeb, jež se hodí pro různé formy použití a integrace do podnikové infrastruktury. Pro účely této práce byla konkrétně využita služba Azure SQL Database.

Podobně jako App Service, i Azure SQL Database je plně řízená služba typu Platform as a Service (PaaS) poskytující databázový stroj, kde je většina běžných správcovských aktivit automaticky pokryta vlastní službou. Jsou to např. funkce spojené s údržbou infrastruktury, aktualizacemi a zabezpečením OS i databázového softwaru, jenž je vždy aktualizován na nejnovější verzi, automatické zálohování serveru, a proaktivní monitoring serveru, aniž by se o to vývojář musel sám starat. Dostupnost služby je podle SLA 99.99 % [20].

Tato služba je ideální pro vývojáře, kteří nechtějí ztrácet čas přípravou a údržbou infastruktury pro SQL Server, jeho instalací, relativně komplikovanou konfigurací, nastavením zabezpečení a namísto toho, preferují soustředit se na vývoj vlastní aplikace a její vhodné škálování [20].

Samozřejmostí služby je možnost nastavit si požadovaný výkon serveru a databáze podle potřeby aplikace. Za zmínku stojí i fakt, že nejmenší výkonové stupně jsou prakticky zdarma a jsou využity i pro hostování projektu této práce. V případě potřeby lze výkon databáze bez výpadku navýšit.

SQL Database je tedy globální služba, která se snadno přizpůsobí většině moderních aplikací a vyniká stálým garantovaným výkonem, vysokou dostupností, kterou od databáze každý očekává. Dále nabízí rozsáhlé možnosti pro škálování podle potřeb aplikace, jež je navíc dynamické a nevyžaduje výpadek služby. Samozřejmostí je také bohatá nabídka bezpečnostních funkcí a dalších nastavení nad rámec standardní konfigurace [20].

Podobně jako App Service, také služba Azure SQL Database je dostupná prakticky ve všech Azure regionech, což je důležité zejména pro snížení latence v rámci aplikace [18].

3.12 **Nástroje Microsoft Visual Studio pro vývoj aplikace**

Pro vývoj aplikace, tj. vlastní kódování v jazyku .NET Core, ladění aplikace i její konfiguraci, stejně tak jako pro synchronizaci zdrojových kódů do GitHub SCM bylo použito vývojové prostředí IDE Microsoft Visual Studio 2019 Community Edition (https://visualstudio.microsoft.com/vs/community/), verze 16.9.0. Pro vývoj byly dále nainstalovány .NET SDK Core 3.1, verze 3.1.406.15636 (x64) a SQL Server Express 2016 jako lokální databázový stroj, který je volitelnou instalační komponentou Visual Studia.

Další nedílnou součástí potřebnou pro vývoj aplikace byly tyto balíky:

- Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore, verze 3.1.8, https://github.com/aspnet/AspNetCore/tree/c75b3f7a2fb9fe21fd96c93c070fdfa88a2fbe97
- Microsoft.AspNetCore.Identity.EntityFrameworkCore, verze 3.1.8, https://github.com/aspnet/AspNetCore/tree/c75b3f7a2fb9fe21fd96c93c070fdfa88a2fbe97
- Microsoft.AspNetCore.Identity.UI, verze 3.1.8, https://github.com/aspnet/AspNetCore/tree/c75b3f7a2fb9fe21fd96c93c070fdfa88a2fbe97
- Microsoft.EntityFrameworkCore.SqlServer, verze 3.1.8, https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Tools/3.1.8
- Microsoft.EntityFrameworkCore.Tools, verze 3.1.8, https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Tools/3.1.8
- Microsoft.VisualStudio.Web.CodeGeneration.Design, verze 3.1.4, https://github.com/dotnet/scaffolding

Pro vývoj unit testů byly dále využity tyto balíky:

- Coverlet.collector, verze 1.3.0, https://github.com/coverlet-coverage/coverlet
- Microsoft.EntityFrameworkcore.InMemory, verze 5.0.2, https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.InMemory/5.0.2

- Microsoft.NET.Test.Sdk, verze 16.7.1, https://github.com/microsoft/vstest/
- Moq, verze 4.15.2, https://github.com/moq/moq4
- MSTest.TestAdapter, verze 2.2.1, https://github.com/microsoft/testfx
- xunit, verze 2.4.1, https://github.com/xunit/xunit
- xunit.assert, verze 2.4.1, https://github.com/xunit/xunit
- xunit.runner.visualstudio, verze 2.4.1, https://github.com/xunit/visualstudio.xunit

Nad rámec Visual Studia, bylo v práci použito také IDE Microsoft Visual Studio (VS) Code (https://code.visualstudio.com/), verze 1.52.1 pro vývoj PowerShell skriptu. IDE bylo použito pro práci s veřejným cloudem Azure, vytvoření potřebných služeb a k následnému nasazení aplikace. V rámci VS Code byl také využit rozšiřující modul Microsoft PowerShell (https://github.com/PowerShell/vscode-powershell/releases), verze 2020.06.0, který poskytuje celou řadu příjemných developerských funkcí pro vývoj PowerShell skriptů, jako jsou kontrola a zvýraznění syntaxe, IntelliSense, automatické code snippety (kód bloky), lokální ladění (debuggování) a podpora speciálního PowerShell terminálu integrovaného přímo do IDE VS Code.

4 Vlastní práce

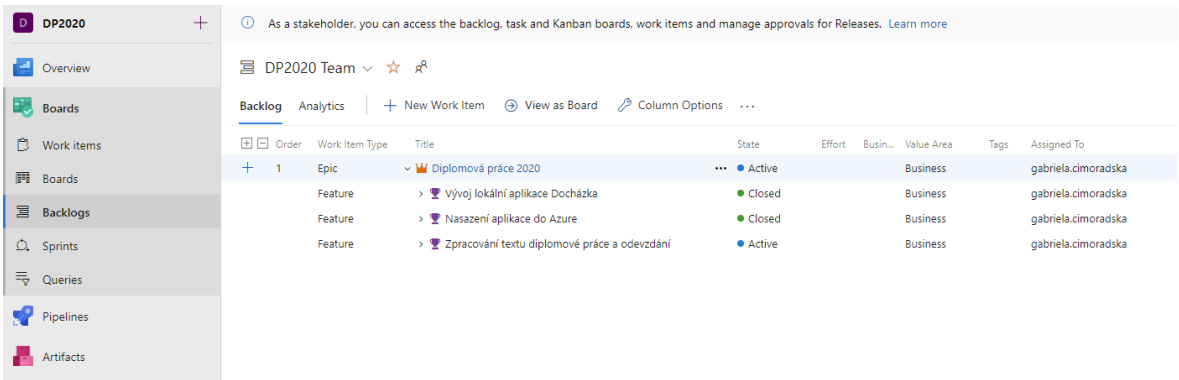
4.1 Analýza zadání a popis pracovních úkolů ve službě Azure DevOps Boards

Cíl diplomové práce, tedy vývoj webové aplikace pro evidenci docházky na platformě .NET Core a její nasazení do veřejného cloudu, byl v úvodu projektu podroben analýze a rozdělen do tří dílčích logických částí (Feature), které byly zdokumentovány v nástroji Azure DevOps Boards.

Hlavní pracovní položka, nazvaná jako Epic Diplomová práce 2020, jež reprezentuje implementační rozsah diplomové práce, je rozdělena do tří dílčích pracovních položek (Feature) takto, viz také Obrázek 2:

- Vývoj lokální aplikace Docházka – tato pracovní položka typu Feature dokumentuje a upřesňuje jednotlivé pracovní úkoly související s vývojem funkční lokální webové aplikace.
- Nasazení aplikace do Azure – tato pracovní položka typu Feature dokumentuje a upřesňuje jednotlivé pracovní úkoly související s nasazením aplikace do cloudové prostředí Microsoft Azure.
- Zpracování textu diplomové práce a odevzdání – tato pracovní položka typu Feature dokumentuje a upřesňuje jednotlivé pracovní úkoly související se sepsáním vlastního textu DP, konzultace s vedoucím a jejím finálním odevzdáním.

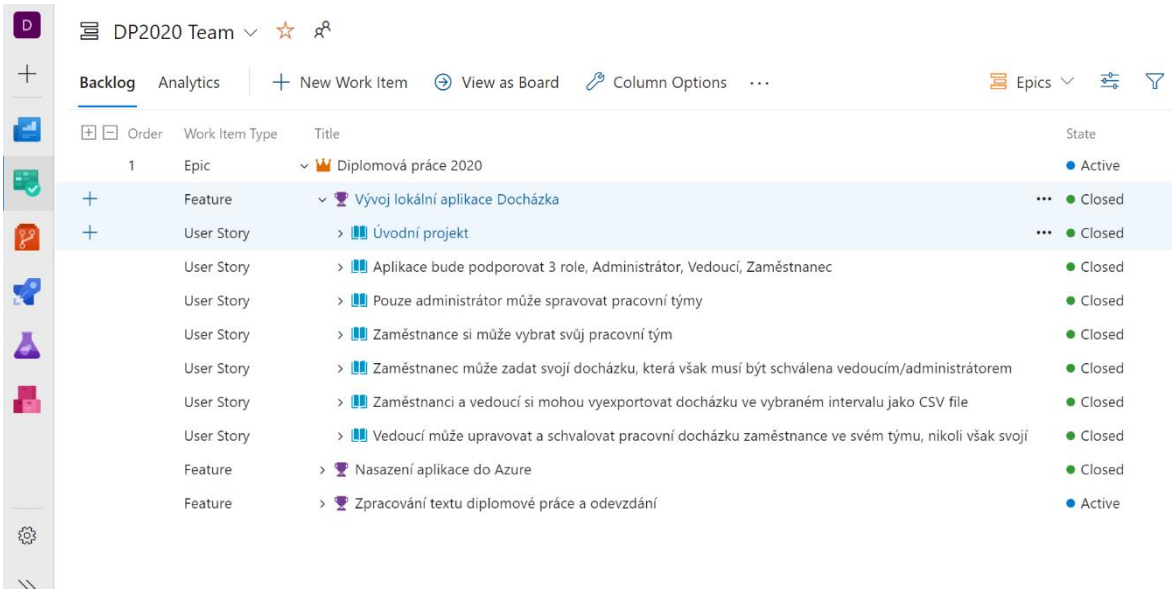
Obrázek 2 - Epic Diplomová práce 2020



4.1.1 Pracovní položka (Feature): Vývoj lokální aplikace docházka

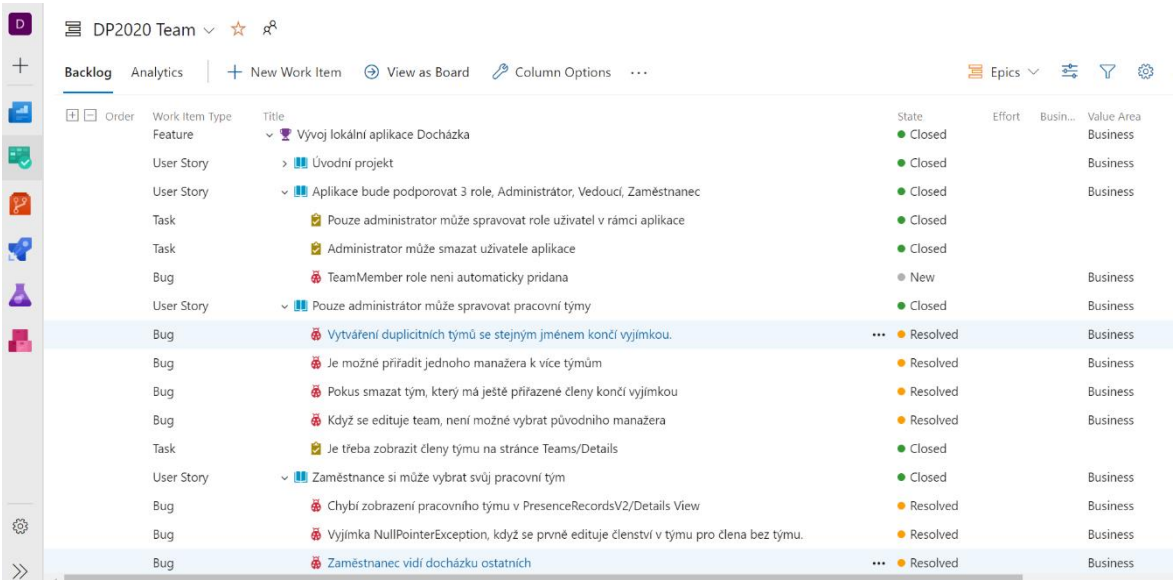
Tato pracovní položka byla dále rozdělena na menší dílčí položky typu User Story, jež popisují funkce aplikace a základní očekávané business požadavky, viz Obrázek 3.

Obrázek 3 - Rozdělení Feature pro vývoj lokální aplikace na dílčí User Story

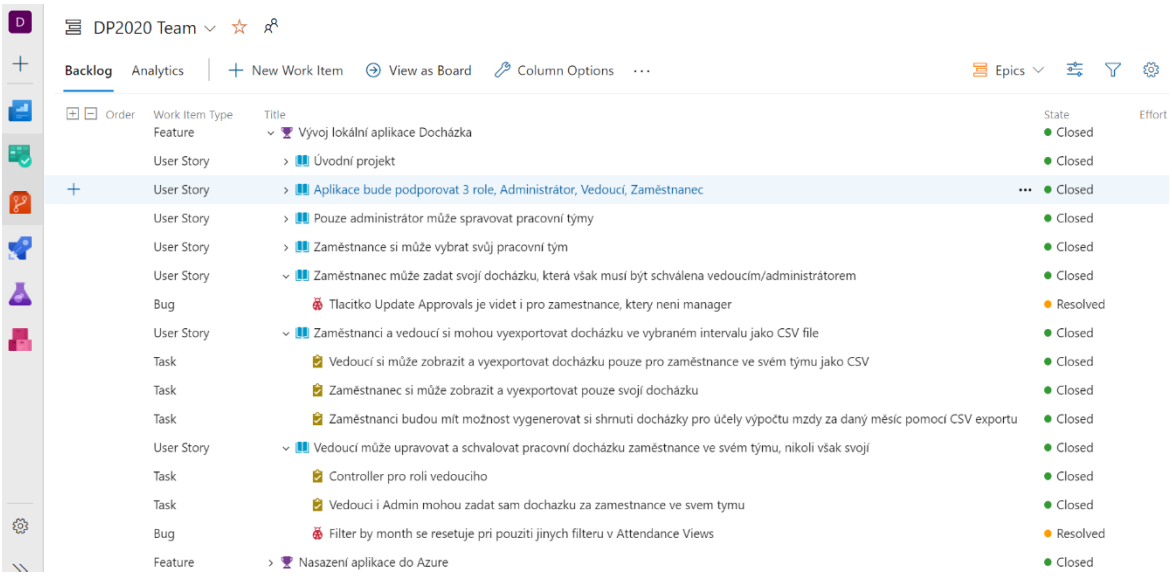


Každá z těchto User Story pak byla ještě ve většině případů podrobena dalšímu rozkladu na menší úkoly (tzv. Task, žlutá ikona) a zároveň k některým z nich byly během vývoje zaevidovány defekty (bug, červená ikona), jež byly odhaleny během ladění aplikace, viz Obrázek 4 a Obrázek 5.

Obrázek 4 - Podrobnější rozklad dílčích User Story na menší úkoly a evidence defektů

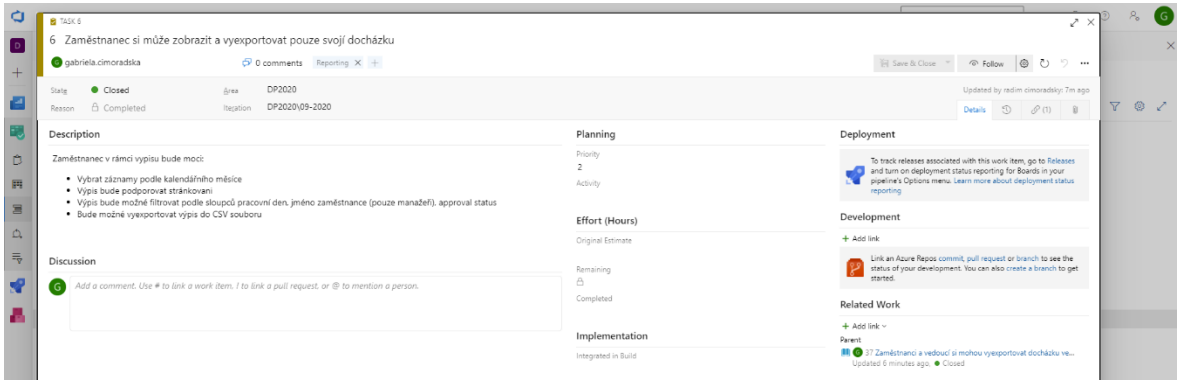


Obrázek 5 - Podrobnější rozklad dílčích User Story na menší úkoly a evidence defektů, pokračování



Nad rámec uvedeného rozkladu na pracovní položky typu User Story a evidenci defektů, byly některé pracovní položky ještě podrobněji rozepsány pomocí podrobných dialogových oken Azure DevOps Boards, aby byla upřesněna hlavní myšlenka daného úkolu či problému. Tato úroveň detailů často pomáhá vyjasnit a zdokumentovat zadání mezi produkt manažerem a vývojářem. Obrázek 6 pak ukazuje příklad takového detailu, tedy popis položky pro výpis docházky uživatele v roli zaměstnance.

Obrázek 6 - Příklad podrobného popisu pracovní položky typu úkol



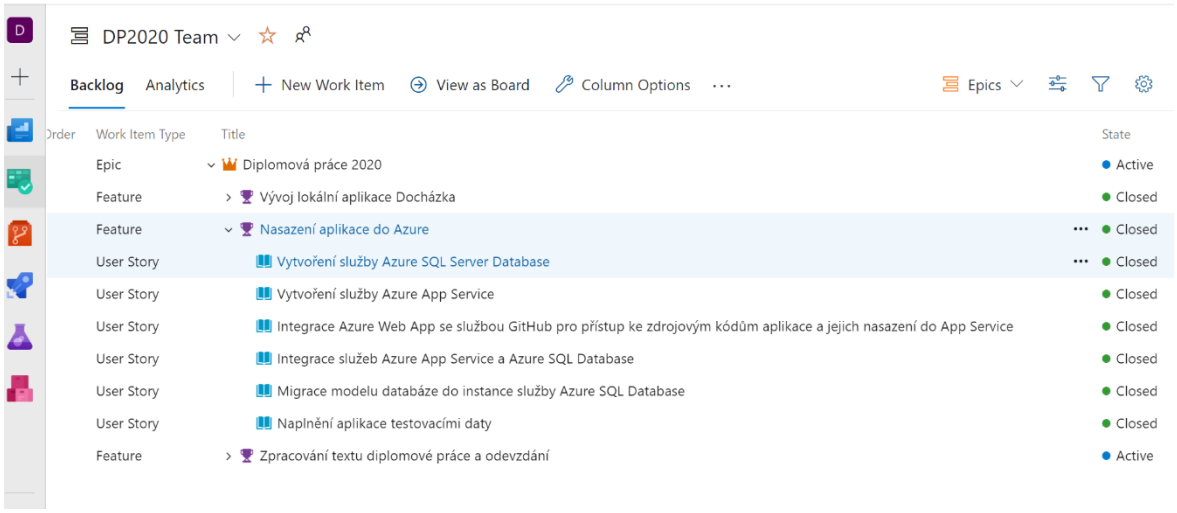
4.1.2 Pracovní položka (Feature): Nasazení aplikace do Azure

Podobně, jako předchozí pracovní položka pro vývoj lokální aplikace, i tato část byla rozložena na menší dílčí položky typu User Story, které ukazuje Obrázek 7.

Všechny položky, s výjimkou naplnění aplikace testovacími daty, byly realizovány pomocí PowerShell skriptu z repositáře diplomové práce:

https://github.com/xsubg001/dp2020/blob/master/Dochazka/scripts/PublishCommands.ps1

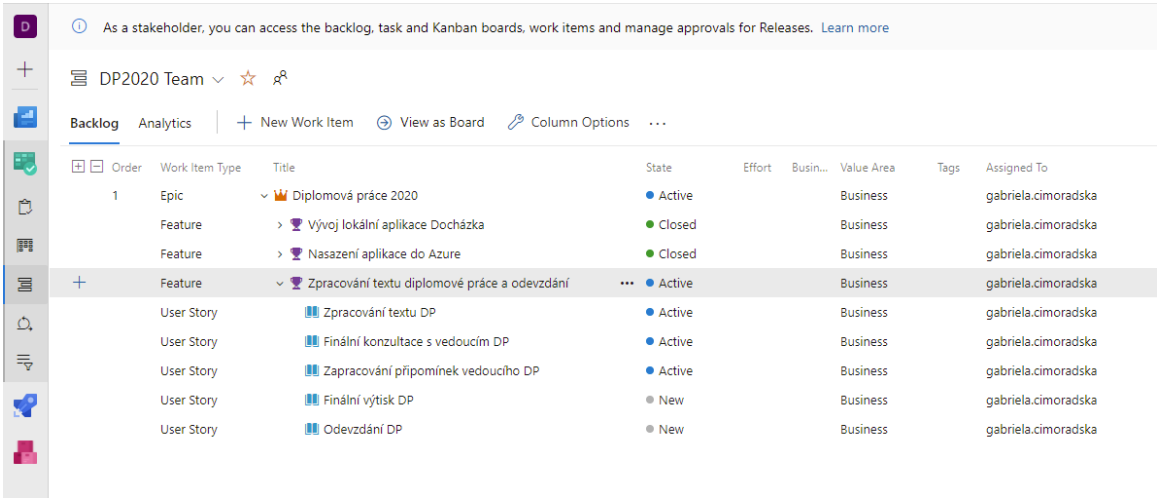
Obrázek 7 - Rozložení pracovní položky pro nasazení aplikace do prostředí Azure cloud



4.1.3 Pracovní položka (Feature): Zpracování textu diplomové práce

Finální pracovní položka pojednává o zpracování textu diplomové práce a byla též rozdělena do menších dílčí pracovních položek typu User Story, což ukazuje Obrázek 8.

Obrázek 8 – Rozložení pracovní položky pro zpracování textu diplomové práce a odevzdání



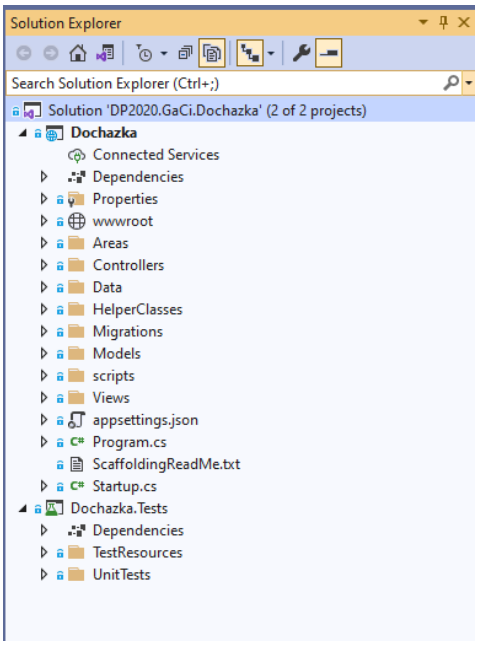
4.2 Popis .NET Core projektu lokální webové aplikace Docházka

Celé řešení (.NET Core solution) webové aplikace se skládá ze dvou hlavních .NET Core projektů:

- **Dochazka.csproj** – tento projekt obsahuje zdrojový kód vlastní webové aplikace.
- **Dochazka.Tests.csproj** – tento projekt obsahuje zdrojový kód unit testů některých částí aplikační logiky, které demonstrují možný způsob, jak testovat kód metodou popsanou v kapitole 3.5.

Náhled na projekty prostřednictvím Solution Exploreru ve Visual Studiu ukazuje Obrázek 9.

Obrázek 9 - Náhled celého řešení .NET Core aplikace prostřednictvím Visual Studia



4.2.1 Projekt Dochazka.csproj

Nyní k jednotlivým částem a složkám projektu podrobněji. Projekt byl založen jako ASP.NET Core MVC řešení pomocí průvodce Visual Studia. To následně vygenerovalo hlavní část adresářové struktury a některé základní soubory v rámci automatického scaffoldingu, tj. vygenerování úvodního skeletonu projektu podle průvodce [21].

4.2.1.1 Složka Properties

Složka obsahuje nastavení základních vlastností projektu a jeho externích závislostí. Vznikla automaticky pomocí průvodce Visual Studia, jímž se zakládá ASP.NET Core projekt (tzv. scaffolding).

`/Properties/`

`launchSettings.json` – obsahuje základní nastavení pro lokální web server IISExpres včetně nastavení lokálních proměnných prostředí pro projekt Dochazka a IISExpress

`serviceDependencies.json` – seznam externích závislostí aplikace, což je v tomto případě pouze závislost na lokálním MSSQL serveru a určení proměnné, jež bude specifikovat připojení do databáze `(connectionString)`

4.2.1.2 Složka wwwroot

Tato složka obsahuje veškerý statický obsah webového serveru, tedy výchozí CSS styly, JS soubory a doplňkové JS knihovny jako Bootstrap a JQuery. Tato část je automaticky přidána průvodcem v rámci úvodního vytvoření projektu (scaffolding) a nebylo potřeba do ní během práce více zasahovat.

```
/wwwroot/
        css/*.css
        js/*.js
        lib/*.js
```

4.2.1.3 Složka Areas/Identity

Složka a její soubory vznikly automaticky scaffoldingem během přidání modulu ASP.NET Core Identity podle průvodce [22].

```
/Areas/
        Identity/
                Data/
                        ApplicationUser.cs
                Pages/
                        Account/
                                Manage/*.cshtml
                                *.cshtml
                IdentityHostingStartup.cs
```

V této složce byla navíc nově vytvořena třída `Dochazka.Areas.Identity.Data.ApplicationUser`, viz Obrázek 10, která definuje model pro uživatele systému a je hlavní entitní třídou pro databázový context Entity Frameworku Core, v němž jsou spravováni uživatelé aplikace. Třída vznikla jako potomek třídy `IdentityUser` a zavádí nové property (vlastnosti modelu):

```
public string FirstName
```

- Obsahuje křestní jméno uživatele. Property je navíc dekorována ASP.NET Core atributy:

- `[Required]`, definuje property, jako nezbytnou pro validní model.
- `[DataType(DataType.Text)]`, specifikuje datový typ pro účely generování View.
- `[StringLength]`, specifikuje maximální délku vstupní hodnoty.
- `[Display(Name = "First Name")]`, určuje nový popisný název pole v HTML, jakmile se generuje View, které obsahuje tento model a property.

```
public string LastName
```

- Obsahuje příjmení uživatele. Property je opět dekorována podobnými atributy jako FirstName.

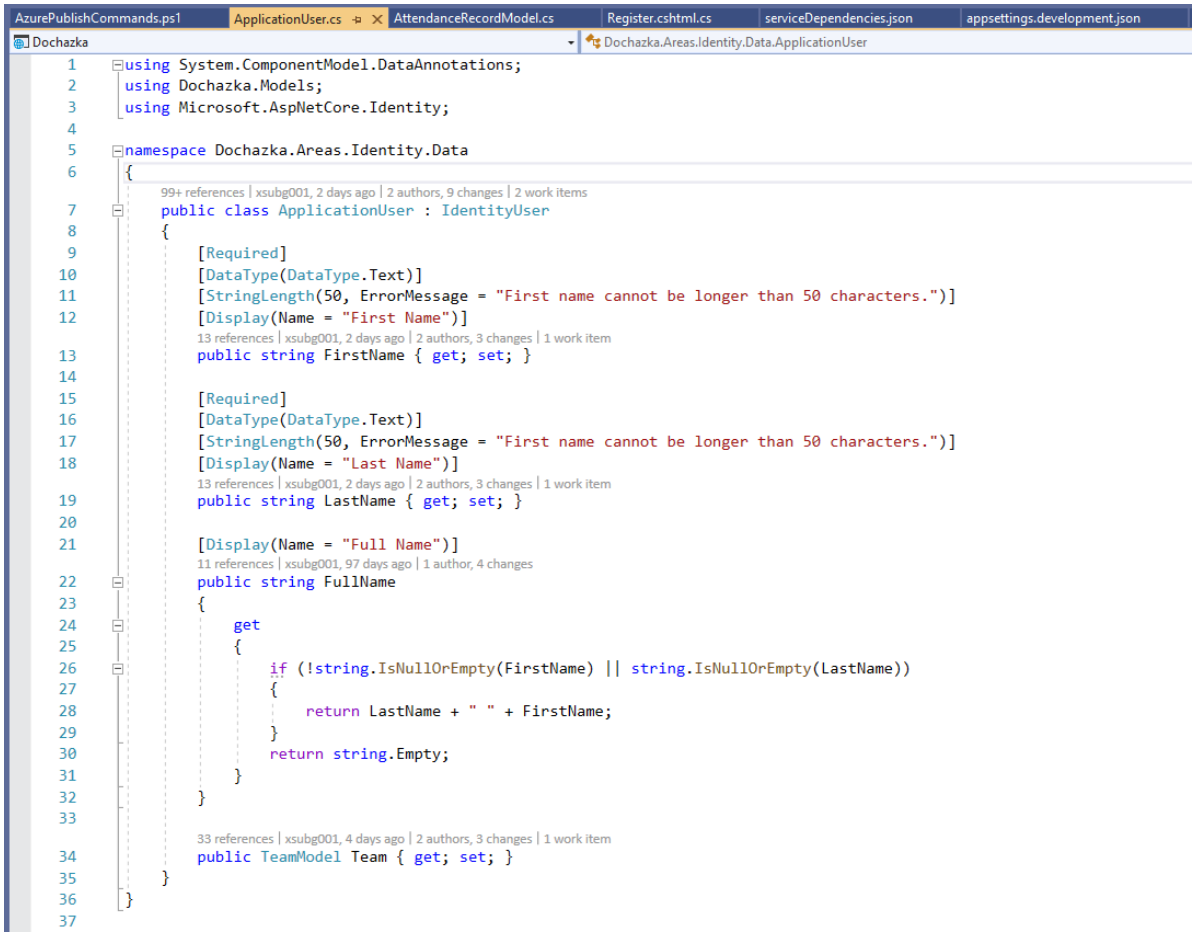
```
public string FullName
```

- Používá se pro získání celého jména pomocí get metody (property getter).

```
public TeamModel Team
```

- Je navigační property na model `TeamModel.cs`, tedy Team, jehož je uživatel členem.

Obrázek 10 - Entitní třída ApplicationUser.cs pro uživatele aplikace využívaná knihovnou ASP.NET Core Identity



Nad rámec standardního scaffoldingu Identity bylo nezbytné provést také několik změn do Razor a souborů a k nim asociovaných *.cs souborů, které se používají při registraci uživatele a při editaci jeho profilu.

Jedná se o tyto soubory používané při registraci uživatele:

`/Areas/Identity/Pages/Account/Register.cshtml.cs`

`/Areas/Identity/Pages/Account/Register.cshtml`

A tyto soubory používané při editaci uživatelského profilu:

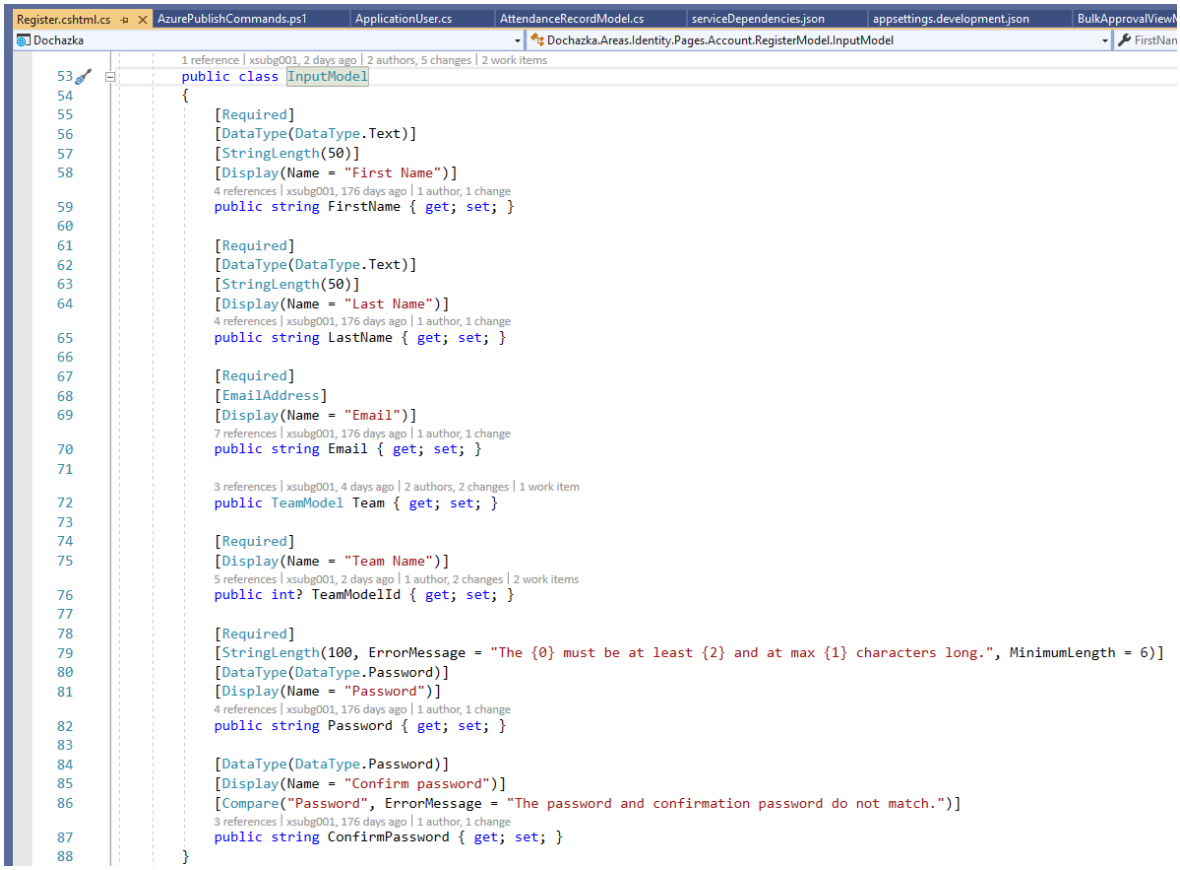
`/Areas/Identity/Pages/Account/Manage/Index.cshtml.cs`

`/Areas/Identity/Pages/Account/Manage/Index.cshtml.cs`

Podstata změn spočívala v rozšíření vstupních datových modelů, jež jsou zpracovány v rámci těchto View, tak aby stránky uměly načíst na vstupu od uživatele nová pole, která jsou definována pro uživatele modelem `ApplicationUser`, tj. `FirstName`, `LastName` a `TeamName` a důležitá pro správnou funkci aplikační logiky.

Proto byly modifikovány jak soubory tříd se vstupním modelem pohledu, viz Obrázek 11, ř. 55-76, tak Razor soubory *.cshtml, které generují View, viz Obrázek 12, ř. 15-29.

Obrázek 11 - Ukázka rozšíření vstupního modelu pro stránku Register.cshtml.cs



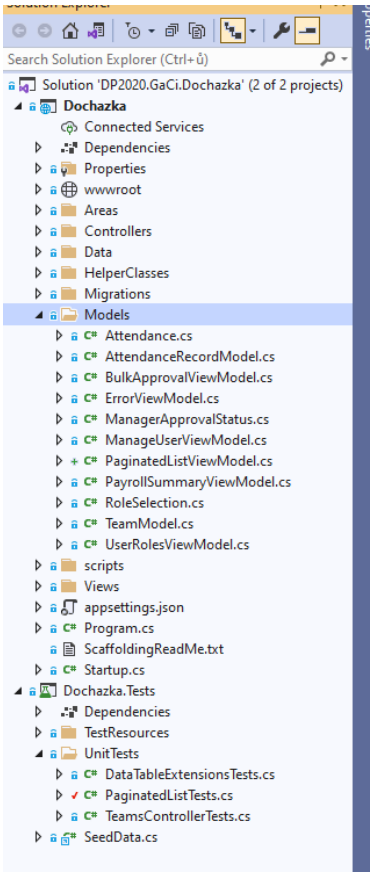
Obrázek 12 - Ukázka rozšíření Razor View pro stránku Register.cshtml

Za zmínku stojí zmínit, že knihovna Identity využívá jednodušší vzor pro generování stránek, kterým jsou tzv. ASP.NET Core Razor Pages. V důsledku toho dochází v asociovaných *.cs souborech ke sloučení modelu a aplikační logiky stránky controlleru, tudíž modely nejsou nutně definovány jako samostatné třídy.

4.2.1.4 Složka Models

V této složce jsou pomocné typy, datové modely jednotlivých entit a ViewModely. ViewModely jsou pomocné modely používané mezi Controllerem a View, jejichž data se skládají ze skutečných entitních modelů. S entitními modely pak Entity Framework Core pracuje na úrovni jejich persistence do databáze. Výčet všech tříd ve složce ukazuje Obrázek 13.

Obrázek 13 - Složka Models s třídami modelů

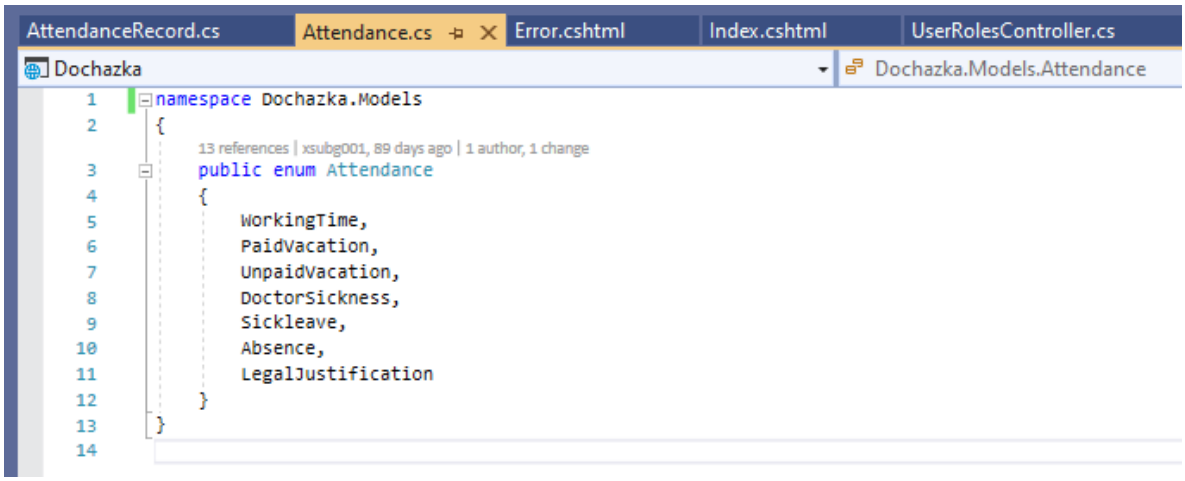


Nyní k jednotlivým modelům blíže.

4.2.1.4.1 Pomocný typ enum Attendance.cs

Obrázek 14 ukazuje zdrojový kód tohoto typu. Tento `enum` reprezentuje možné hodnoty zaznamenané docházky a je to pomocný typ pro model `AttendanceRecordModel.cs`. Každý zaznamenaný den docházky, má dvě hodnoty z tohoto typu, jednu pro ráno a druhou pro odpoledne, tzn. že pracovník může ráno vykázat např. `WorkingTime` (odpracovaná doba) a odpoledne `PaidVacation` (placená dovolená).

Obrázek 14 – Enum Attendance.cs



4.2.1.4.2 Pomocný typ enum ManagerApprovalStatus.cs

Podobně jako předchozí typ, i toto je pomocný typ pro model `AttendanceRecordModel.cs`, který definuje možné validní hodnoty pro stav schválení daného záznamu docházky. Každý nový záznam docházky je implicitně vytvořen s hodnotou `Submitted` (Odesláno) a následně je na manažerovi teamu, aby docházku schválil (stav `Approved`), nebo zamítnul (stav `Rejected`). Zdrojový kód typu ukazuje Obrázek 15.

Obrázek 15 - Enum ManagerApprovalStatus.cs

4.2.1.4.3 Model AttendanceRecordModel.cs

Obrázek 16 ukazuje zdrojový kód tohoto modelu. Jedná se o jeden z hlavních modelů aplikace, který dokumentuje pracovní docházku zaměstnance a jeho data jsou pomocí Entity Frameworku Core ukládána do databáze. Primární klíč modelu je kompozitní a je tvořen pomocí dvou properties (vlastností modelu) `WorkDay` a `EmployeeId`, což je specifikováno v `/Data/ApplicationDbContext.cs` pomocí konstrukce:

```
modelBuilder.Entity<AttendanceRecordModel>()
        .HasKey(p => new { p.EmployeeId, p.WorkDay});
```

Model má několik zajímavých properties:

```
public DateTime WorkDay { get; set; }
```

- Specifikuje kalendářní den pracovní docházky.

```
public Attendance MorningAttendance { get; set; }
```

- Záznam docházky pro ranní část pracovního dne, viz 4.2.1.4.1.

```
public Attendance AfternoonAttendance { get; set; }
```

- Záznam docházky pro odpolední část pracovního dne, viz 4.2.1.4.1.

```
public string EmployeeId { get; set; }
public ApplicationUser Employee { get; set; }
```

- Jedná se o dvojici navigačních property, které navigují na konkrétního uživatele, jehož se tento záznam docházky týká.

```
public ManagerApprovalStatus ManagerApprovalStatus { get; set; }
```

- Jedná se o property, která reprezentuje stav schválení daného výkazu docházky manažerem teamu, jehož je uživatel členem, viz 4.2.1.4.2.

```
public byte[] RowVersion { get; set; }
```

- Používá se pro detekci konfliktu, pokud více uživatelů modifikuje stejný datový objekt.

Některé z properties jsou navíc dekorovány dalšími ASP.NET Core atributy, které dále upřesňují chování property modelu, při jejím zobrazení a editaci ve View.

Příkladem může být:

```
[Display(Name = "Work Day Date"), DataType(DataType.Date)]
[DisplayFormat(DataFormatString = "{0:dd-MM-yyyy}", ApplyFormatInEditMode = false)]
public DateTime WorkDay { get; set; }
```

Tyto atributy upřesňují formát zobrazení pole ve View. Tak např. atribut `DataType(DataType.Date)]` zajišťuje, že se pole bude zadávat prostřednictvím HTML ovládacího prvku kalendář spíš, než jako obyčejný textový HTML formulář.

V modelu je také definován konstruktor, který zajišťuje výchozí inicializaci některých důležitých properties modelu, viz ř. 39-44, Obrázek 16.

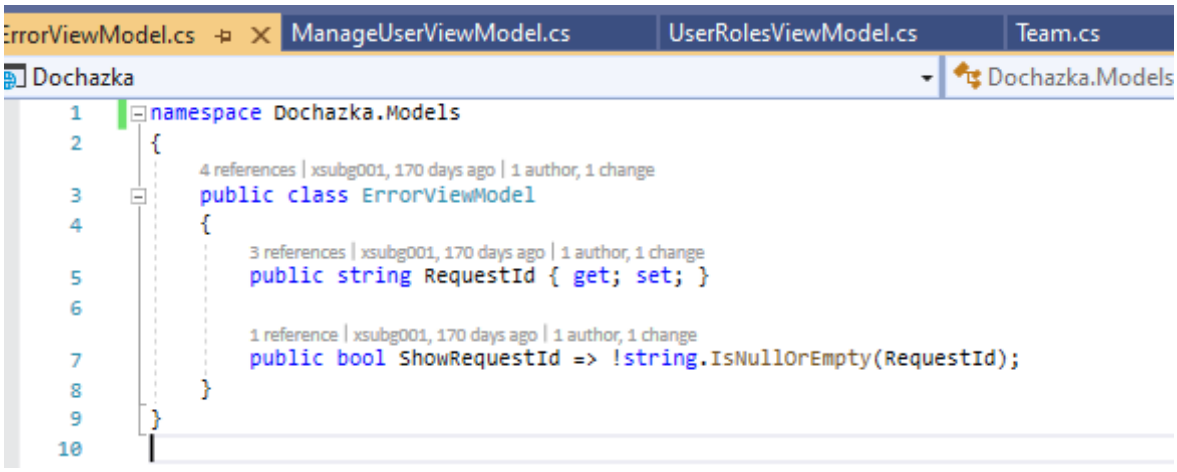
Obrázek 16 - Model AttendanceRecordModel.cs

Tento model je použit v součinnosti s Controllerem `AttendanceRecordsController.cs`, o němž bude pojednáno později.

4.2.1.4.4 ViewModel ErrorViewModel.cs

Jedná se o jednoduchý model vytvořený automaticky v rámci skeletonu projektu, který je použit v třídě `HomeController.cs` pro zobrazení chybové stránky pomocí akce `Error()`. Zdrojový kód modelu ukazuje Obrázek 17.

Obrázek 17 - Model ErrorViewModel.cs



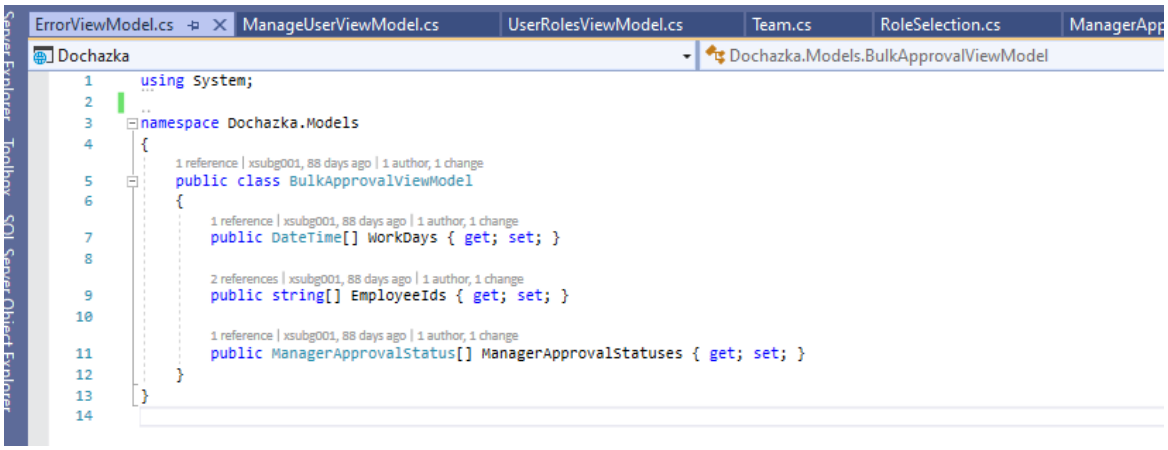
4.2.1.4.5 ViewModel BulkApprovalViewModel.cs

Jde o pomocný ViewModel, jenž má využití při zobrazení View prostřednictvím Controlleru `AttendanceRecordsController.cs` a `HttpPost` akční metody:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Index(BulkApprovalViewModel bulkApprovals, string currentFilter, string sortOrder, DateTime selectedMonth, int? pageNumber)
```

Touto akcí může manažer odpovídajícího teamu nebo administrátor hromadně schválit docházku zaměstnanců. Model není přímo ukládán do databáze, nicméně data z modelu jsou postupně použita pro schválení záznamů docházky, resp. aktualizaci jednotlivých záznamů modelu `AttendanceRecords.cs`, a to pomocí properties `WorkDays`, `EmployeeIds` a `ManagerApprovalStatuses`, jež jsou typu `Array`. Zdrojový kód ukazuje Obrázek 18.

Obrázek 18 - ViewModel BulkApprovalViewModel.cs



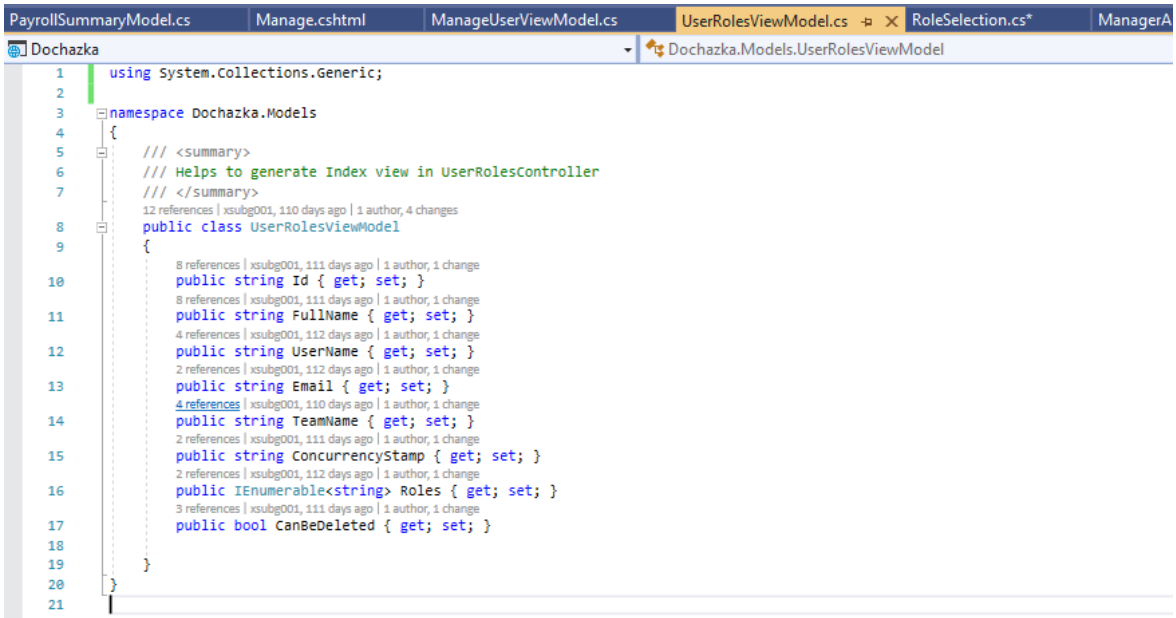
4.2.1.4.6 ViewModel UserRolesViewModel.cs

Pomocný ViewModel je zejména využit při zobrazení View `Index`, jež zobrazí matici uživatelských účtů, jejich přiřazení k teamu, a jejich aplikační role v rámci Controlleru `UserRolesController.cs` pomocí `HttpGet` akční metody:

```
public async Task<IActionResult> Index(string sortOrder, string currentFilter, string searchString, int? pageNumber)
```

Zdrojový kód modelu ukazuje Obrázek 19.

Obrázek 19 - ViewModel UserRolesViewModel.cs



Ani tento model není přímo ukládán do databáze, ale je použit pouze pro vygenerování dat pro dané View.

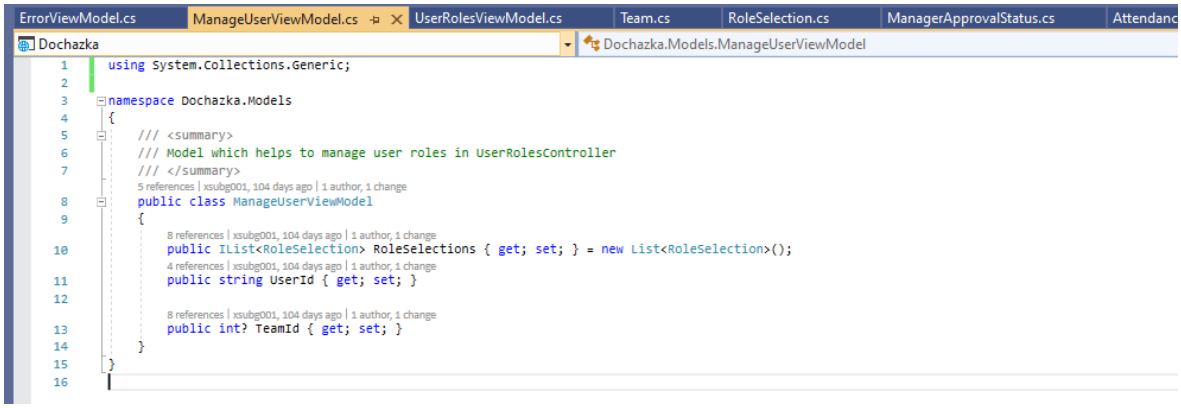
4.2.1.4.7 ViewModel ManageUserViewModel.cs

Pro úpravu matice mapování mezi uživatelem, jemu přidělených rolí a přiřazení do teamu je použit ViewModel `ManageUserViewModel.cs`, který má opět využití v Controlleru `UserRolesController.cs`. Konkrétně je použít v akcích `HttpGet Manage` a `HttpPost Manage`, v nichž dochází k zobrazení formuláře pro editaci přidělených rolí danému uživateli.

```
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Manage(string id)

[HttpPost]
[ValidateAntiForgeryToken]
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Manage(ManageUserViewModel input)
```

Pouze uživatel se systémovou rolí `TeamAdministratorRole` může volat tyto akce a modifikovat tak přidělené aplikační role uživateli. Omezení přístupu k akci na roli `TeamAdministratorRole` je zajištěno pomocí ASP.NET Core atributu `[Authorize(Roles = "TeamAdministratorRole")]`. Zdrojový kód ViewModelu ukazuje Obrázek 20.

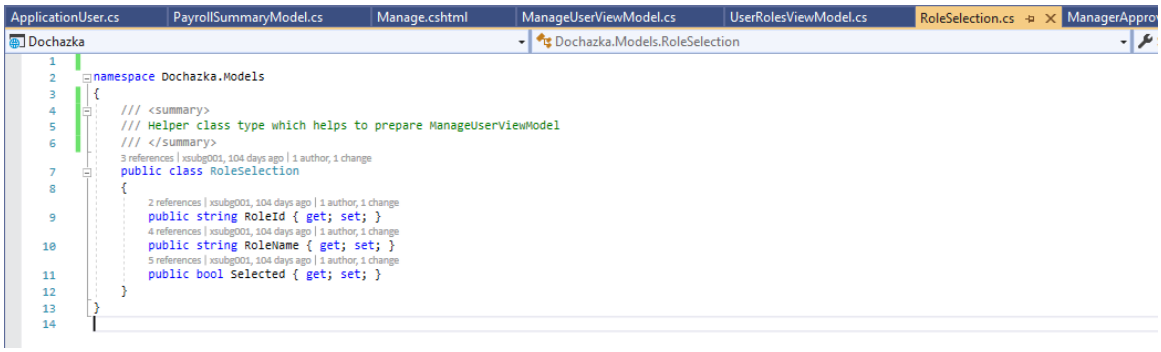
Obrázek 20 - ViewModel ManageUserViewModel.cs



4.2.1.4.8 Pomocná třída RoleSelection.cs

Tato pomocná třída je použita v předchozím ViewModelu `ManageUserViewModel.cs` pro uložení informace o tom, jaké konkrétní role jsou uživateli přiděleny. Zdrojový kód ukazuje Obrázek 21.

Obrázek 21 – Pomocná třída RoleSelection.cs



4.2.1.4.9 ViewModel PayrollSummaryViewModel.cs

Pro zobrazení agregovaného souhrnu pracovní docházky za daný kalendářní měsíc pro jednoho zaměstnance se používá ViewModel `PayrollSummaryViewModel.cs`, jehož zdrojový zachycuje Obrázek 22. Tento agregovaný souhrn docházky může mít např. využití při výpočtu mzdy.

Model je použit v rámci Controlleru `AttendanceRecordModel.cs` prostřednictvím akční metody:

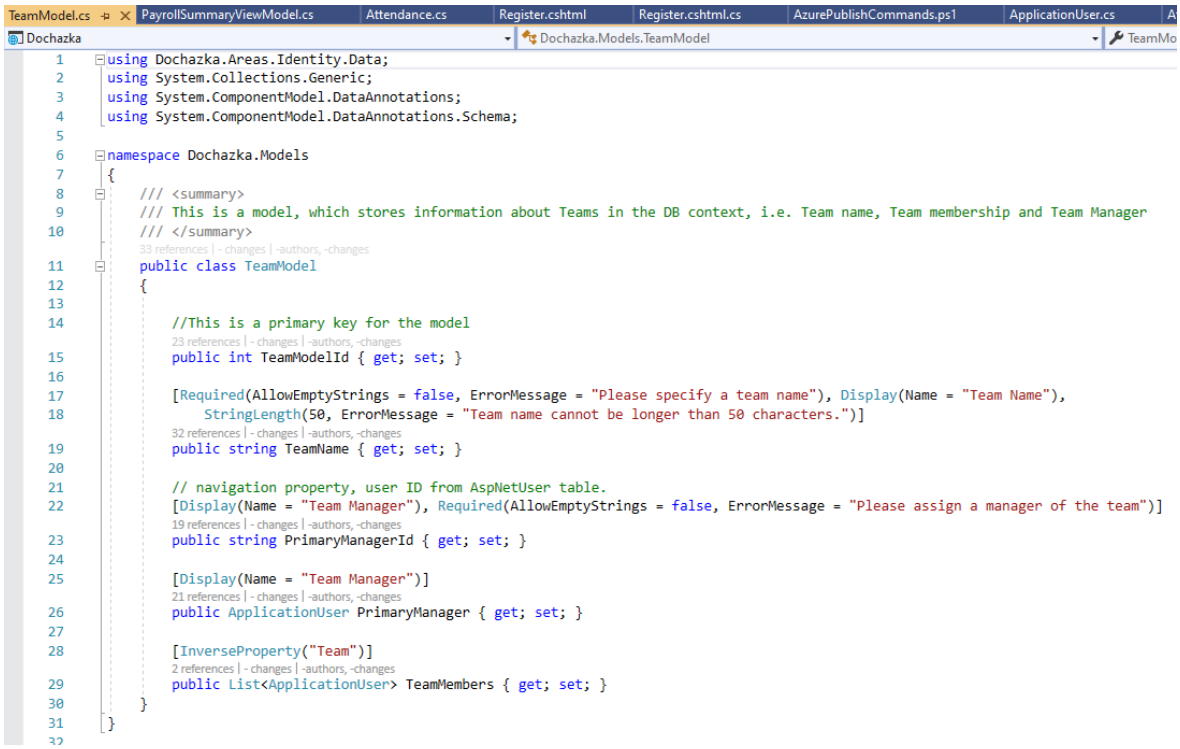
```
// GET: Payroll summary for selected employees
public async Task<IActionResult> PayrollSummary(string searchString, DateTime selectedMonth, bool getAsCsv)
```

Obrázek 22 - ViewModel PayrollSummaryModel.cs

4.2.1.4.10 Model TeamModel.cs

Jde o další z klíčových modelů, který slouží k vytvoření logické vazby mezi jednotlivými členy teamu a manažerem teamu. Objekty modelu jsou pomocí Entity Frameworku Core zapisovány do databáze. Primární klíč je definovaný pomocí property `TeamModelId`. Zdrojový kód ukazuje Obrázek 23.

Obrázek 23 - Model TeamModel.cs



Kromě zmíněného primárního klíče `TeamModelId`, třída definuje navíc tyto vlastnosti (properties) modelu:

```
[Required(AllowEmptyStrings = false, ErrorMessage = "Please specify a team name"), Display(Name = "Team Name"), StringLength(50, ErrorMessage = "Team name cannot be longer than 50 characters.")]
public string TeamName { get; set; }
```

- Je zde uložena informace o jménu teamu. Je dekorováno atributem jako nutné pole pro validní model a má nastaveno omezení na maximální délku řetězce.

```
[Display(Name = "Team Manager"), Required(AllowEmptyStrings = false, ErrorMessage = "Please assign a manager of the team")]
public string PrimaryManagerId { get; set; }
[Display(Name = "Team Manager")]
public ApplicationUser PrimaryManager { get; set; }
```

- Dvojice navigačních property, jež drží informaci o manažerovi teamu. Dekorováno atributem pro zobrazení volitelného popisu ve View a také atributem, jako požadované pole pro validní model.

```
[InverseProperty("Team")]
public List<ApplicationUser> TeamMembers { get; set; }
```

- Inverzní navigační property na členy teamu, tedy `List` typu `ApplicationUser`. Je dekorováno atributem `[InverseProperty("Team")]`, který svazuje tuto property s property v modelu v `ApplicationUser.cs`. Technicky jde o relační vazbu 1:N, tj. jeden team může mít N členů.

4.2.1.4.11 Model PaginatedListViewModel.cs

Jde o speciální generický ViewModel, jenž má využití v akčních metodách `Index` většiny Controllerů, kde se vypisuje dlouhý seznam entit, který je obvykle potřeba stránkovat.

Zdrojový kód modelu zachycuje Obrázek 24 a byl inspirován průvodcem pro ASP.NET Core [23]. Model dědí z generické třídy `List<T>` a přidává několik členů, které umožňují obyčejný vstupní seznam `IQueryable<T> source` rozdělit na pevný počet segmentů (stránek) s požadovaným počtem prvků, tedy podle velikosti stránky `int pageSize`. Potřebný seznam prvků pro danou stránku lze získat jako výsledek metody `Create`. Ta v návratové hodnotě vrací vybrané prvky, jež odpovídají všem položkám pro požadovanou stránku výpisu podle hodnoty `int pageIndex`. V modelu je metoda definovaná jak v synchronní, tak asynchronní podobě, jelikož některé vstupní seznamy `IQueryable<T> source`, jež jsou v projektu použity, nepodporují metodu `CountAsync()`.

Dále jsou v modelu zavedeny dvě nové properties `public bool HasPreviousPage` a `public bool HasNextPage`, které indikují, zda aktuální stránka má předchozí stránku a následující stránku. Tyto properties se používají ve View `Index`, pro aktivaci tlačítek umožňujícími přechod mezi stránkami.

Obrázek 24 - ViewModel PaginatedListViewModel.cs

PaginatedListViewModel.cs | Index.cshtml | _ValidationScriptsPartial.cshtml | Create.cshtml | Index.cshtml | Delete.cshtml | Edit.cshtml

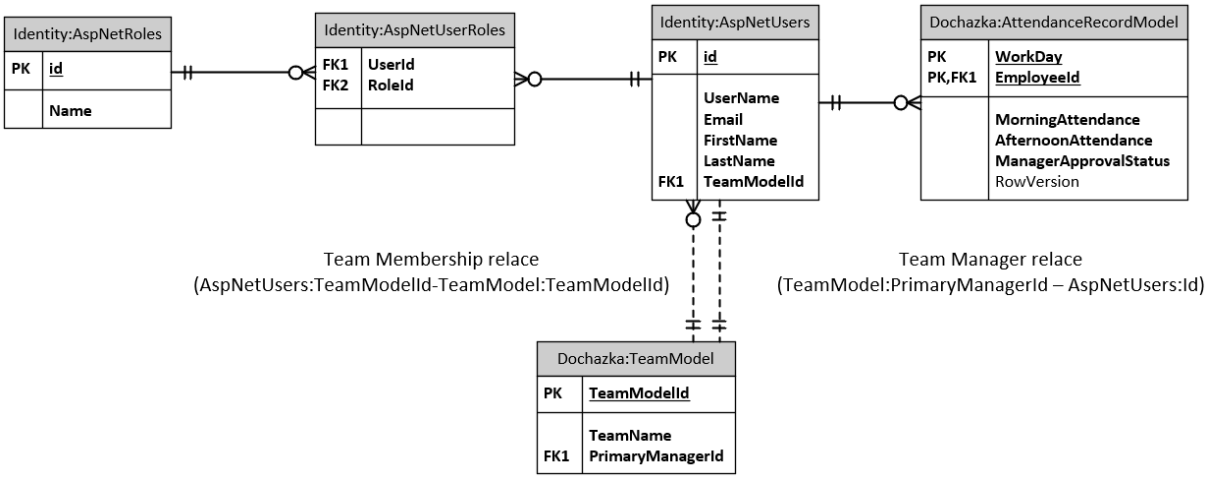
Dochazka | Dochazka.Models.PaginatedListViewModel<T>

```
1   using Microsoft.EntityFrameworkCore;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Threading.Tasks;
6
7   namespace Dochazka.Models
8   {
        19 references | 0 changes | 0 authors, 0 changes
9       public class PaginatedListViewModel<T> : List<T>
10      {
            14 references | 0 changes | 0 authors, 0 changes
11          public int PageIndex { get; private set; }
            4 references | 0 changes | 0 authors, 0 changes
12          public int TotalPages { get; private set; }
13
            5 references | 0 changes | 0 authors, 0 changes
14          public bool HasPreviousPage
15          {
16              get
17              {
18                  return PageIndex > 1;
19              }
20          }
21
            5 references | 0 changes | 0 authors, 0 changes
22          public bool HasNextPage
23          {
24              get
25              {
26                  return PageIndex < TotalPages;
27              }
28          }
29
            2 references | 0 changes | 0 authors, 0 changes
30          public PaginatedListViewModel(List<T> items, int count, int pageIndex, int pageSize)
31          {
32              PageIndex = pageIndex;
33              TotalPages = (int)Math.Ceiling(count / (double)pageSize);
34              AddRange(items);
35          }
36
            2 references | 0 changes | 0 authors, 0 changes
37          public static async Task<PaginatedListViewModel<T>> CreateAsync(IQueryable<T> source, int pageIndex, int pageSize)
38          {
39              var count = await source.CountAsync();
40              var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
41              return new PaginatedListViewModel<T>(items, count, pageIndex, pageSize);
42          }
43
            3 references | 0 changes | 0 authors, 0 changes
44          public static PaginatedListViewModel<T> Create(IQueryable<T> source, int pageIndex, int pageSize)
45          {
46              var count = source.Count();
47              var items = source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToList();
48              return new PaginatedListViewModel<T>(items, count, pageIndex, pageSize);
49          }
50      }
51  }
52
```

4.2.1.4.12 ER diagram aplikace

Obrázek 25 zachycuje Entitní Relační Diagram (ERD), který vychází z entitních tříd modelů popsaných výše.

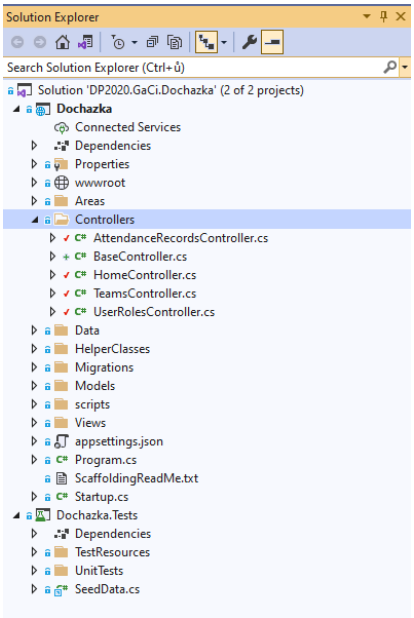
Obrázek 25 - ERM model aplikace



4.2.1.5 Složka Controllers

Tato složka obsahuje kód MVC Controllerů, viz Obrázek 26.

Obrázek 26 - Solution Explorer pohled na složku Controllers aplikace



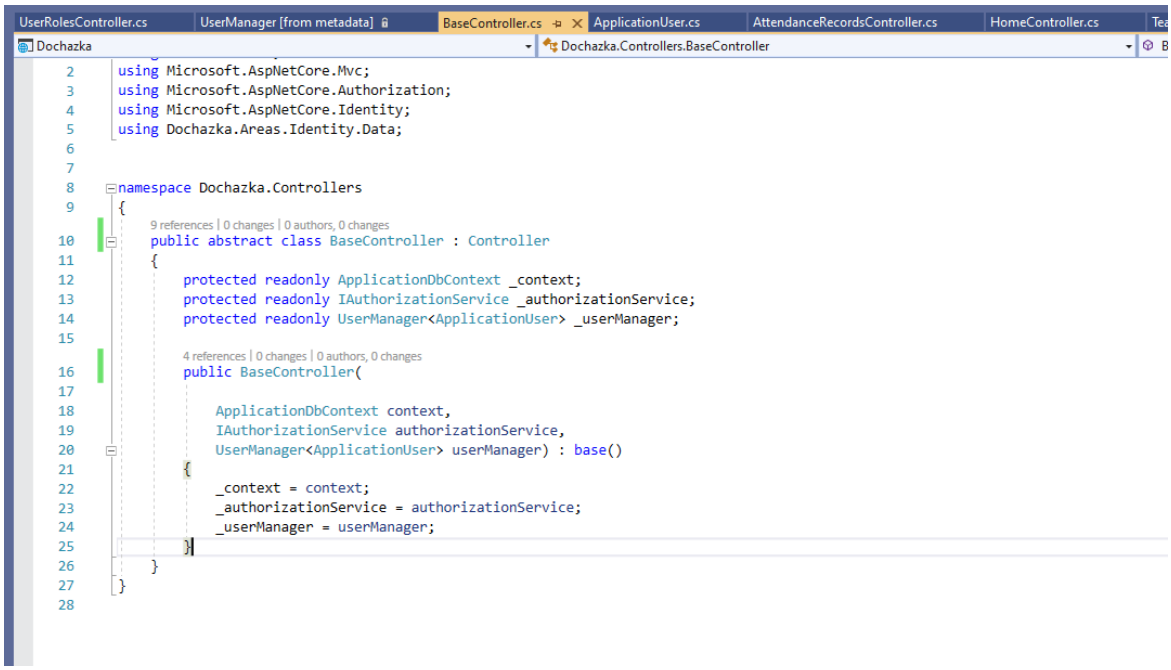
Nyní podrobněji k jednotlivým Controllerům.

4.2.1.5.1 Třída BaseController.cs

Třída byla vytvořena z důvodu redukce duplikace kódu. Jeho zdrojový kód ukazuje Obrázek 27. Neobsahuje žádnou logiku a slouží pouze jako Base Class (rodič) pro ostatní Controllery níže, které z něj dědí tyto členy:

```
protected readonly ApplicationDbContext _context;
protected readonly IAuthorizationService _authorizationService;
protected readonly UserManager<ApplicationUser> _userManager;
```

Obrázek 27 - Kód třídy Controlleru BaseController.cs



Za zmínku stojí, že člen `authorizationService` je nastaven pomocí mechanismu Dependency Injection prostřednictvím konstruktoru třídy, kde Dependency Injection je prostředek umožňující tvorbu softwaru pomocí návrhového principu zvaného jako Dependency Inversion [24]. Zbývající dva členi `_authorizationService` a `_userManager` využívají tradiční přímou závislost inicializace a jsou deklarovány pomocí konkrétní třídy.

4.2.1.5.1.1 Návrhový princip Dependency Inversion

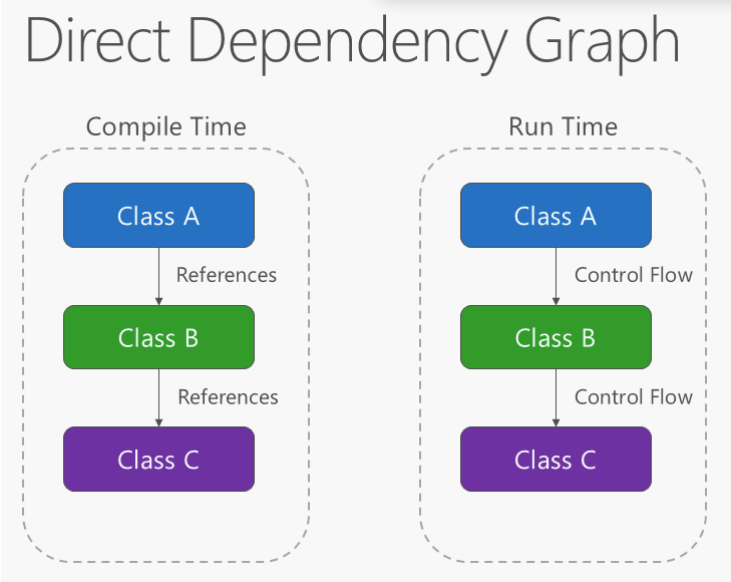
Dependency Injection je programovací styl nebo vzor, který pomáhá realizovat softwarový princip zvaný jako Dependency Inversion.

Dependency Inversion je součástí skupiny návrhových principů označovaných jako „SOLID“, kde „D“ reprezentuje princip Dependency Inversion [25].

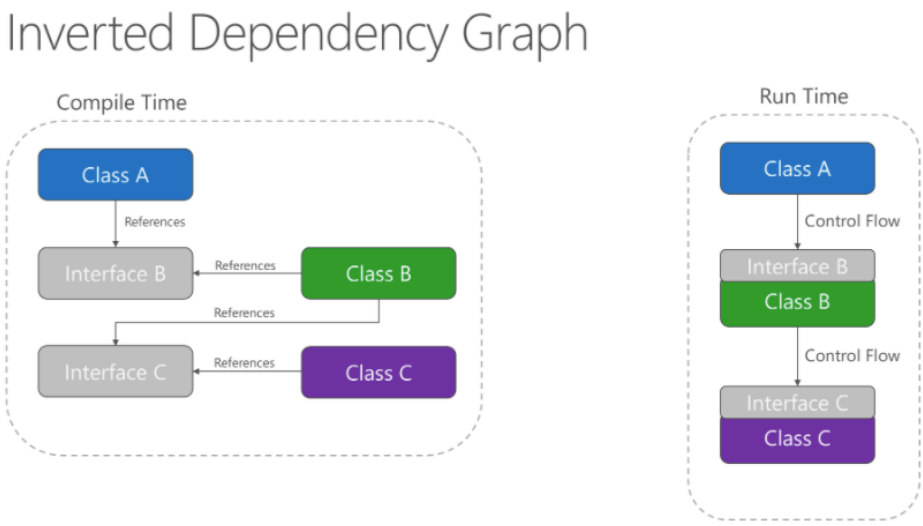
Tento konkrétní princip usiluje o vytváření závislosti mezi komponentami aplikacemi na základě abstrakce, spíš než konkrétní implementace třídy. Abstrakcí se v tomto případě myslí závislost na rozhraní (interface), jež je dále implementováno nějakou třídou, která poskytuje potřebnou funkcionalitu. Situace mezi přímou závislostí komponent (Direct Dependency) a inverzní závislostí (Inverted Dependency) ukazují Obrázek 28 a Obrázek 29 [26].

Výhodou tohoto přístupu je možnost nahradit konkrétní implementace rozhraní, tedy jednu funkční třídu za jinou se stejným rozhraním, aniž by bylo nutné měnit kód, jenž jejich funkcionalitu potřebuje a závisí na nich [26].

Obrázek 28 - Graf tradiční přímé závislosti (Direct Dependency) mezi komponentami (třídami)



Obrázek 29 – Graf inverzní závislosti (Inverted Dependency) mezi komponentami (třídami)



Dependency Inversion je důležitý princip pro tvorbu aplikací, jež mají mít volně provázané komponenty. Výsledné aplikace jsou pak lépe testovatelné, modulární a dobře se udržují. Dependency Injection je pak metoda, která umožňuje dosáhnout tento návrhový princip [26].

4.2.1.5.2 Třída HomeController.cs

Zdrojový kód tohoto jednoduchého Controlleru zachycuje Obrázek 30.

Obrázek 30 - Zdrojový kód Controlleru HomeController.cs

HomeController.cs

Dochazka | Dochazka.Controllers.HomeController

```
1   using System.Diagnostics;
2   using Microsoft.AspNetCore.Mvc;
3   using Microsoft.Extensions.Logging;
4   using Dochazka.Models;
5   using Microsoft.AspNetCore.Authorization;
6   using Dochazka.Data;
7   using Microsoft.AspNetCore.Identity;
8   using Dochazka.Areas.Identity.Data;
9
10  namespace Dochazka.Controllers
11  {
        3 references | xsubg001, 5 days ago | 2 authors, 4 changes | 1 work item
12      public class HomeController : BaseController
13      {
14          private readonly ILogger<HomeController> _logger;
15
            0 references | xsubg001, 5 days ago | 2 authors, 2 changes | 1 work item
16          public HomeController(
17              ILogger<HomeController> logger,
18              ApplicationDbContext context,
19              IAuthorizationService authorizationService,
20              UserManager<ApplicationUser> userManager)
21              : base(context, authorizationService, userManager)
22          {
23              _logger = logger;
24          }
25
26          [AllowAnonymous]
            0 references | xsubg001, 178 days ago | 1 author, 2 changes
27          public IActionResult Index()
28          {
29              return View();
30          }
31
32          [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
            0 references | xsubg001, 179 days ago | 1 author, 1 change
33          public IActionResult Error()
34          {
35              return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
36          }
37      }
38  }
39
```

Jde o výchozí Controller, který zajišťuje zobrazení domácí stránky aplikace přes položky menu Home a AttendanceSheet. Kromě toho také dokáže vygenerovat chybovou stránku Error se specifickými informacemi pro vývojáře.

Nasměrování na tento Controller je zajištěno prostřednictvím sdílené šablony View `_Layout.cshtml`, jež je použita také ve všech ostatních Views a definuje mj. podobu hlavního navigačního menu aplikace.

Zdrojový soubor sdílené View šablony, kde jsou odkazy na tento i ostatní Controllery ukazuje Obrázek 31, viz ř. 14, 23, 28, 35, 38, 54.

Obrázek 31 - Zdrojový kód sdílené View šablony _Layout.cshtml

_Layout.cshtml | HomeController.cs

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - AttendanceSheet</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">AttendanceSheet</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <partial name="_LoginPartial" />
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
                        </li>
                        @if (User.Identity.IsAuthenticated)
                        {
                            <li class="nav-item">
                                <a class="nav-link text-dark" asp-area="" asp-controller="AttendanceRecords" asp-action="Index">My Attendance</a>
                            </li>
                        }
                        @if (User.Identity.IsAuthenticated)
                        {

                            <li class="nav-item">
                                <a class="nav-link text-dark" asp-area="" asp-controller="Teams" asp-action="Index">Teams</a>
                            </li>
                            <li class="nav-item">
                                <a class="nav-link text-dark" asp-area="" asp-controller="UserRoles" asp-action="Index">Users</a>
                            </li>
                        }
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            <a asp-area="" asp-controller="Home" asp-action="Index">&copy; 2021 - Diploma dissertation project, Gabriela C.</a>
            <div hidden>v1.02</div>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @RenderSection("Scripts", required: false)
</body>
</html>
```

`HomeController` odkazuje skrz metody `Index()` a `Error()` na svoje dvě Views:

`/Views/Home/Index.cshtml`

`/Views/Shared/Error.cshtml`

4.2.1.5.3 Třída TeamsController.cs

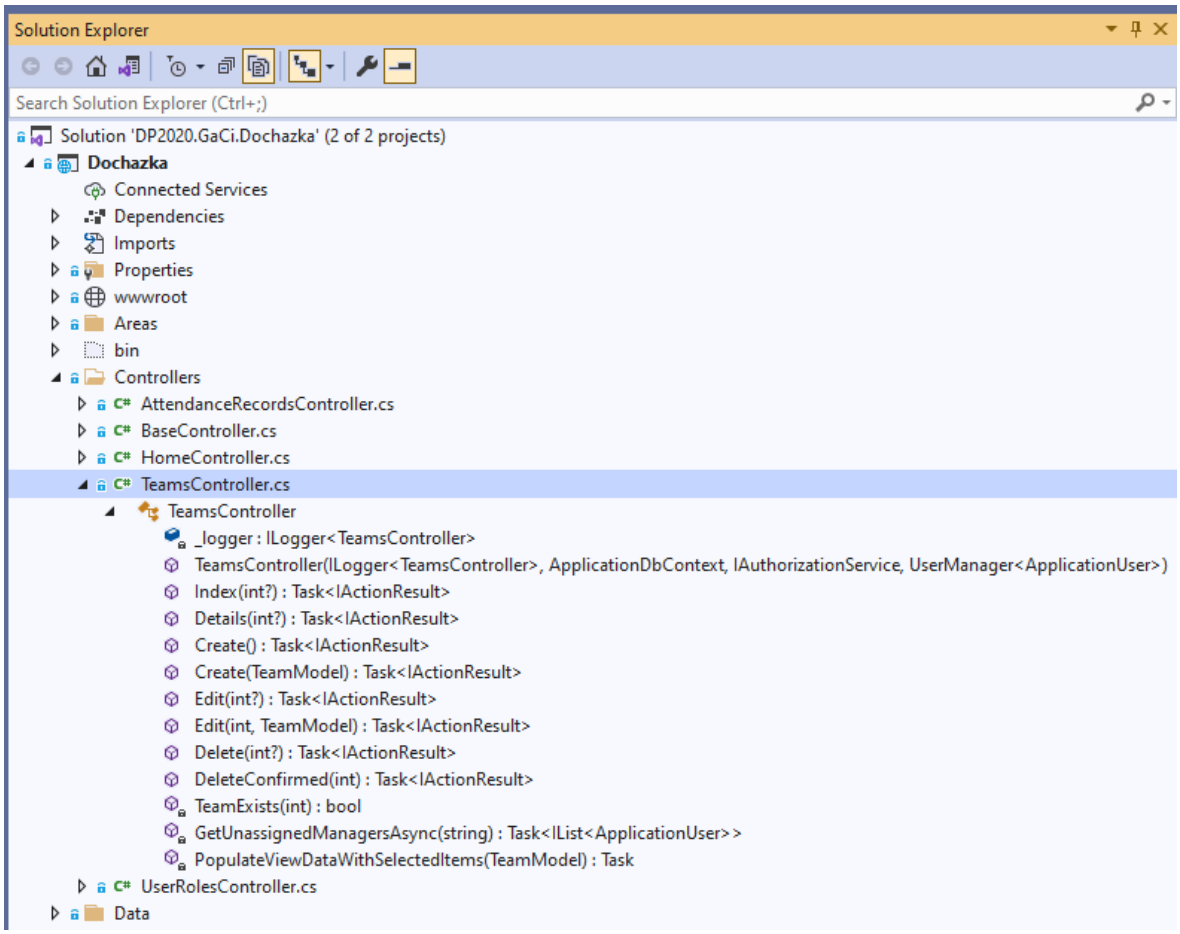
Třída TeamsController.cs implementuje funkce potřebné ke správě teamů v rámci aplikace. Třída dědí z `BaseController.cs`, o níž bylo pojednáno v 0.

Pohled na třídu v Solution Exploreru ukazuje Obrázek 32.

.

`TeamsController` umožňuje provádět základní CRUD (Create, Read, Update, Delete) operace na modelu `TeamModel.cs`.

Obrázek 32 - Třída TeamsController.cs v Solution Exploreru



V třídě jsou tři pomocné metody:

```
private bool TeamExists(int id)
```

- Metoda vrací hodnotu `bool` na základě skutečnosti, jestli team s daným `id` v databázi již existuje a je použita v akční metodě `HttpPost Edit`.

```
private async Task<IList<ApplicationUser>> GetUnassignedManagersAsync(string? id)
```

- Metoda vrací seznam manažerů, kteří ještě nemají přidělený žádný team. Výsledek může být rozšířen i o manažera s daným `id`, pokud není `null`. To se hodí zejména ve View při editaci teamu, kde je potřeba získat seznam všech volných manažerů a současně zahrnout i aktuálního manažera teamu.

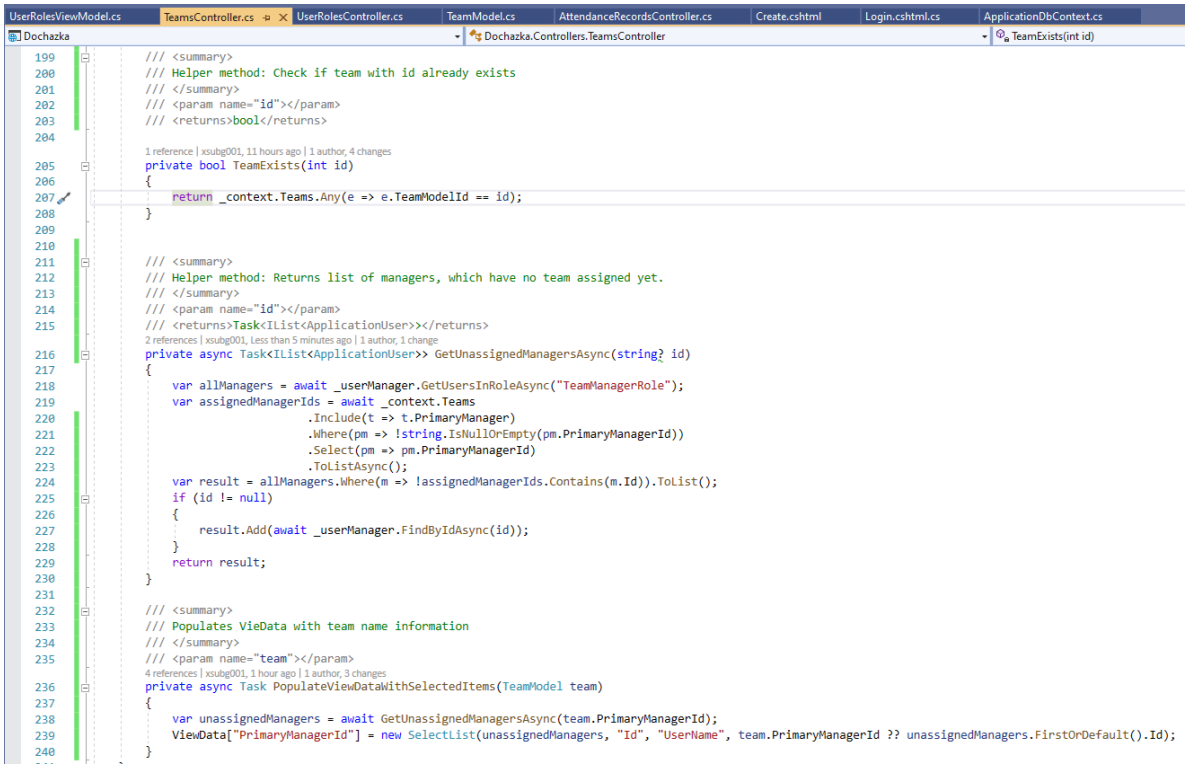
```
private async Task PopulateViewDataWithSelectedItems(TeamModel team)
```

- Je asynchronní `void` metoda, která zajišťuje naplnění hodnot `ViewData` dictionary, jenž se používá v několika Views. `ViewData` je property typu `ViewDataDictionary`, jenž Controller dědí z třídy `Controller` a používá se pro přenos různých

dynamických dat mezi Controllerem a View. V případě tohoto Controlleru obsahuje objekt `Microsoft.AspNetCore.Mvc.Rendering.SelectList`, v němž je seznam nepřiřazených manažerů, případně včetně i aktuálního manažera, kteří mohou být přiřazeni danému teamu při jeho editaci nebo vytvoření.

Zdrojový kód metody ukazuje Obrázek 33.

Obrázek 33 - Zdrojový kód pomocných metod v TeamsController.cs



Dále stručně k ostatním akčním metodám třídy `TeamsController`, jež zajišťují CRUD operace s modelem `TeamsModel.cs`:

```
public async Task<IActionResult> Index(int? pageNumber)
```

- `HttpGet` akce, jež slouží k výpisu všech teamů. Akce zobrazí formulář pomocí View ze složky `/Views/Teams/Index.cshtml`.

```
public async Task<IActionResult> Details(int? id)
```

- `HttpGet` akce, jež slouží k zobrazení detailu o konkrétním teamu. Akce zobrazí formulář pomocí View ze složky `/Views/Teams/Details.cshtml`.

```
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Create()
```

- `HttpGet` akce, jež slouží k zobrazení formuláře pro vytvoření nového teamu. Je chráněna atributem `[Authorize(Roles = "TeamAdministratorRole")]`, tzn. že ji mohou volat pouze uživatelé s požadovanou rolí "`TeamAdministratorRole`". Stejný atribut se objevuje také u ostatních metod `Create`, `Edit`, `Delete` a `DeleteConfirmed`. Akce zobrazí formulář pomocí View ze složky `/Views/Teams/Create.cshtml`.

```
[HttpPost]
[ValidateAntiForgeryToken]
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Create([Bind("TeamName,PrimaryManagerId")]
TeamModel team)
```

- `HttpPost` akce, která zpracuje Create formulář a v případě úspěšné validace modelu vytvoří nový Team do databáze prostřednictvím databázového contextu. Metoda je dekorována atributem `[ValidateAntiForgeryToken]`, jenž poskytuje ochranu proti útoku typu podvrhnutí formuláře v HTTP dotazu (cross-site request forgery).

```
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Edit(int? id)
```

- `HttpGet` akce, jež slouží k zobrazení formuláře pro edici vybraného teamu. Akce zobrazí formulář pomocí View ze složky `/Views/Teams/Edit.cshtml`.

```
[HttpPost]
[ValidateAntiForgeryToken]
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Edit(int id,
[Bind("TeamModelId,TeamName,PrimaryManagerId")] TeamModel team)
```

- `HttpPost` akce, která zpracuje Edit formulář a v případě úspěšné validace modelu provede změnu pro daný Team do databáze. Pomocí této akce lze změnit manažera a jméno teamu.

```
// GET: Teams/Delete/5
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Delete(int? id)
```

- `HttpGet` akce, jež slouží k zobrazení formuláře pro smazání vybraného teamu. Akce zobrazí formulář pomocí View ze složky `/Views/Teams/Delete.cshtml`.

```
// POST: Teams/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> DeleteConfirmed(int id)
```

- `HttpPost` akce, která provede smazání teamu v databázi, pokud je akce potvrzena na formuláři `Delete`.

Vzhledem k tomu, že není z objemového hlediska možné podrobně zdokumentovat všechny metody všech Controlleru do detailu, budou v následující části podrobněji rozebrány pouze dvě vzorové metody tohoto vybraného Controlleru.

4.2.1.5.3.1 HttpGet metoda public async Task<IActionResult> Details(int? id)

Zdrojový kód metody ukazuje Obrázek 34. Jde o asynchronní metodu, která vrací typ `Task<IActionResult>`. `IActionResult` je standardně očekávaný návratový typ pro akci Controlleru, který bude výsledkem metody, jakmile metoda úspěšně asynchronně doběhne.

Metoda je volána po přijetí HTTP GET dotazu, např. na adresu `https://localhost:44385/Teams/Details/8`.

Logika metody je taková, že na základě parametru `id` je nalezen patřičný team pomocí databázového contextu ukazujícího na všechny entity modelu TeamModel.cs (ř. 48), který splňuje podmínku, že `TeamModelId == id` (ř. 50) a současně se z databázového contextu načtou data o manažerovi teamu (ř. 49) přes navigační property `PrimaryManagerId`.

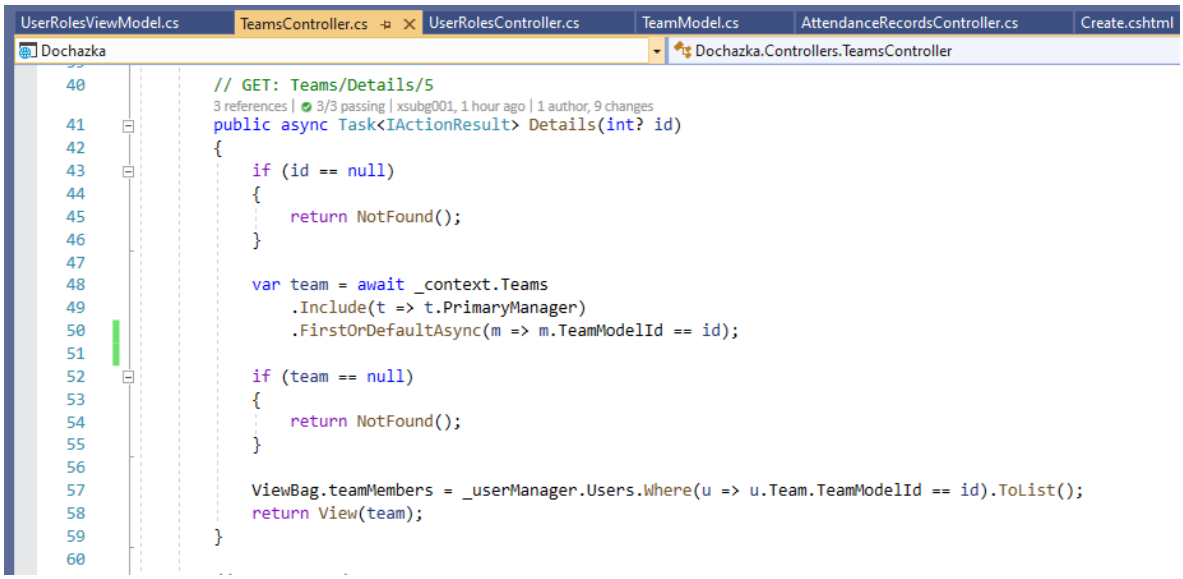
Pokud team s daným `id` neexistuje, vrátí metoda HTTP odpověď `NotFound (404)`, ř. 54.

Pokud team existuje, potom se z databázového contextu `Users` načtou na ř. 57 všichni uživatelé, jejichž property `TeamModelId == id`. Tento seznam slouží k zobrazení všech členů teamu a je uložen v dynamické property `ViewBag.teamMembers`.

Následně je vráceno View `Views/Teams/Details.cshtml` (ř. 58), které pracuje s modelem `TeamModel`. View vygeneruje HTML stránku, jež zobrazí podrobnosti o daném

teamu a také jeho členy pomocí dynamické property `ViewBag.teamMembers`, která je rovněž předána do View.

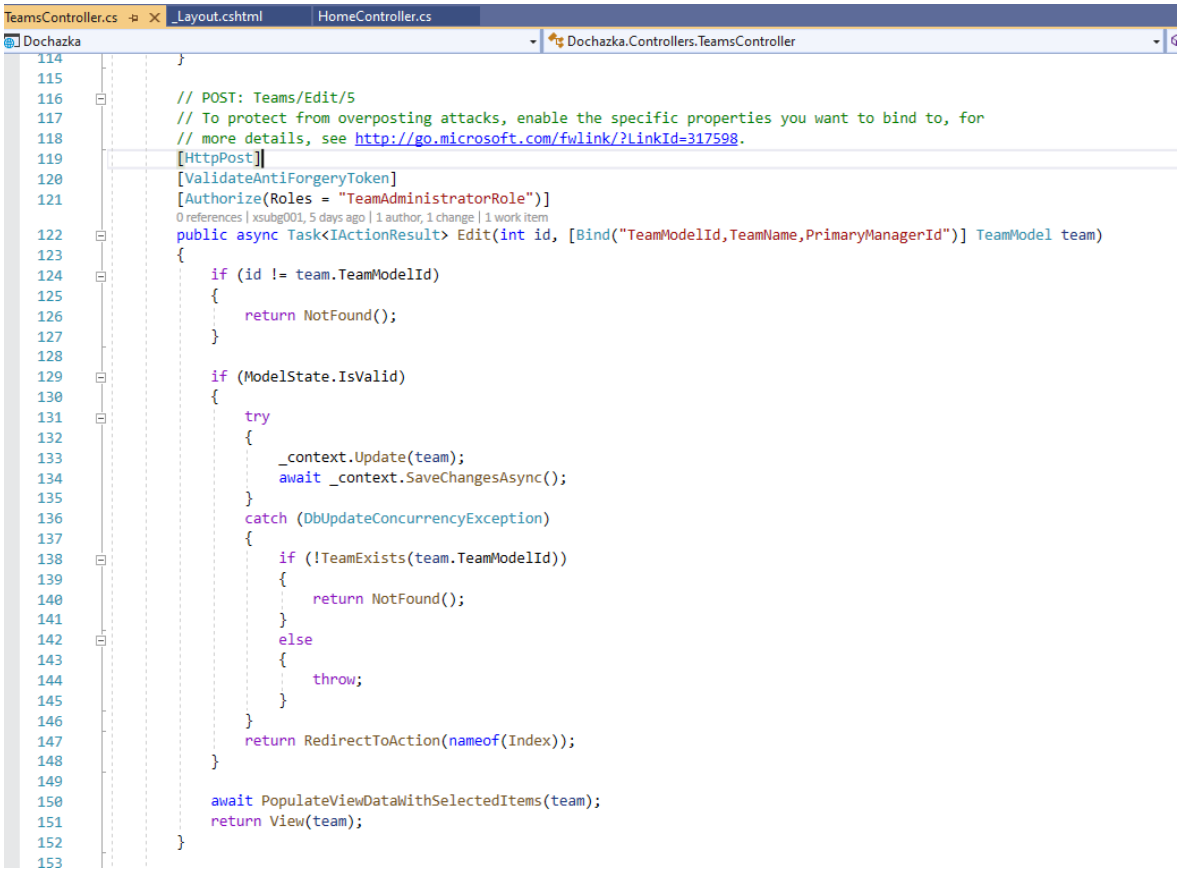
Obrázek 34 - Zdrojový kód HttpGet metody Details(int? id)



4.2.1.5.3.2 HttpPost metoda public async Task<IActionResult> Edit(int id, [Bind("TeamModelId,TeamName,PrimaryManagerId")] TeamModel team)

Metoda je v zásadě podobná, jako předchozí příklad. Nicméně se liší tím, že je dekorována atributem `[HttpPost]`, který zajišťuje, že je metoda zavolána pouze po přijetí HTTP POST dotazu, např. na adresu `https://localhost:44385/Teams/Edit/8`, který se používá k modifikaci nebo vložení dat od klienta na server. Zdrojový kód zachycuje Obrázek 35.

Obrázek 35 - Zdrojový kód HttpPost metody Edit(int id, [Bind("TeamModelId, TeamName, PrimaryManagerId")] TeamModel team)



Na vstupu metoda pracuje s parametry `int id` a `TeamModel team`. Identifikátor editovaného teamu je v parametru `id` a do parametru `team` se navážou hodnoty z HTTP dotazu, zadané prostřednictvím HTML formuláře. Navázaní je omezeno pouze na properties modelu `TeamModelId`, `TeamName`, `PrimaryManagerId`, pomocí atributu `[Bind("TeamModelId,TeamName,PrimaryManagerId")]`.

Pokud se `id` a `team.TeamModelId` nerovnají, je vrácena `NotFound (404)` odpověď (ř. 127), jelikož se jedná o chybný vstup.

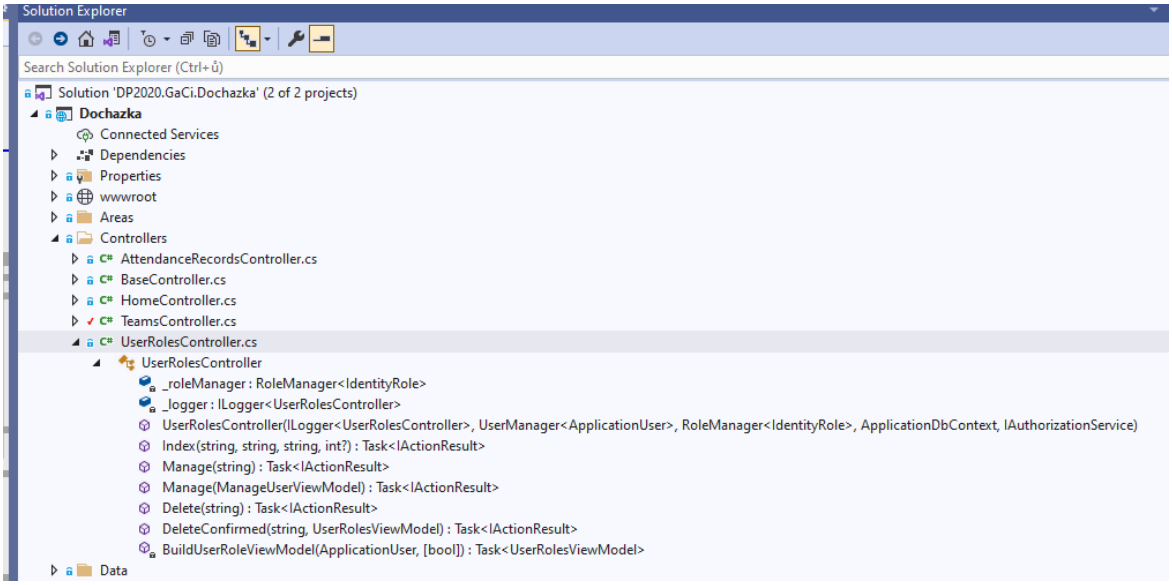
Pokud je Model validní, dojde k aktualizaci daného teamu v databázovém contextu (ř. 134) a následně k uložení do databáze (ř. 135). Tato část je také ošetřena z pohledu možné výjimky `DbUpdateConcurrencyException`, pro případ, že by záznam teamu byl v mezidobí aktualizován, nebo dokonce smazán jiným uživatelem, viz ř. 137-147. V případě úspěšné aktualizace databáze, je uživatel přesměrován na akci `Index` (ř. 148), kde se mu již zobrazí aktualizovaná data teamu.

Nevalidní model naopak vede k opětovnému zobrazení editační stránky, tedy View ze složky `/Views/Teams/Edit.cshtml`, ř. 151-152.

4.2.1.5.4 Třída UserRolesController.cs

Controller se používá zejména k administraci uživatelských účtů z pohledu přiřazení aplikačních rolí a k případnému smazaní účtu. Dědí členy z `BaseController.cs`. Třída poskytuje několik jednoduchých akcí, jimiž lze zobrazit, či editovat přiřazené role uživateli, a také smazat vybraného uživatele nebo změnit team, jehož je členem.

Obrázek 36 - Solution Explorer pohled na třídu UserRolesController.cs

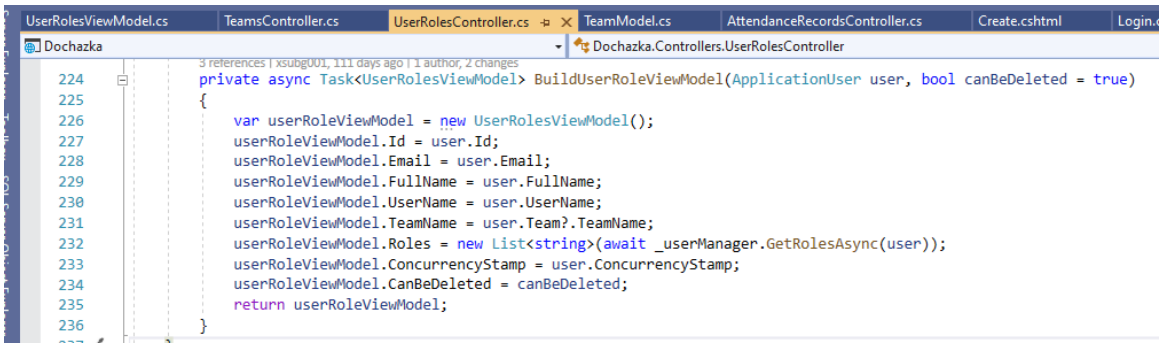


Kromě akčních metod Controlleru, je v třídě jedna pomocná metoda:

```
private async Task<UserRolesViewModel> BuildUserRoleViewModel(ApplicationUser user,
bool canBeDeleted = true)
```

- Zdrojový kód metody ukazuje Obrázek 37. Úkolem metody je připravit objekt `UserRolesViewModel` na základě vstupních parametrů metody `ApplicationUser user` (model uživatele) a volitelného parametru `bool canBeDeleted`. Parametr `canBeDeleted` je použit v akční metodě `HttpGet Delete`, kde pomáhá určit, jestli daný uživatel není stále přiřazen jako manažer teamu, a tudíž může být bezpečně smazán.

Obrázek 37 – Pomocná metoda BuildUserRoleViewModel



Nyní k ostatním akčním metodám `UserRolesControlleru` a konstruktoru třídy.

```
public UserRolesController(
        ILogger<UserRolesController> logger,
        UserManager<ApplicationUser> userManager,
        RoleManager<IdentityRole> roleManager,
        ApplicationDbContext context,
        IAuthorizationService authorizationService) : base(context,
authorizationService, userManager)
```

- Konstruktor třídy volá jak rodičovský konstruktor, tak navíc inicializuje dva další nové členy třídy, viz Obrázek 38. Jsou to následující properties:

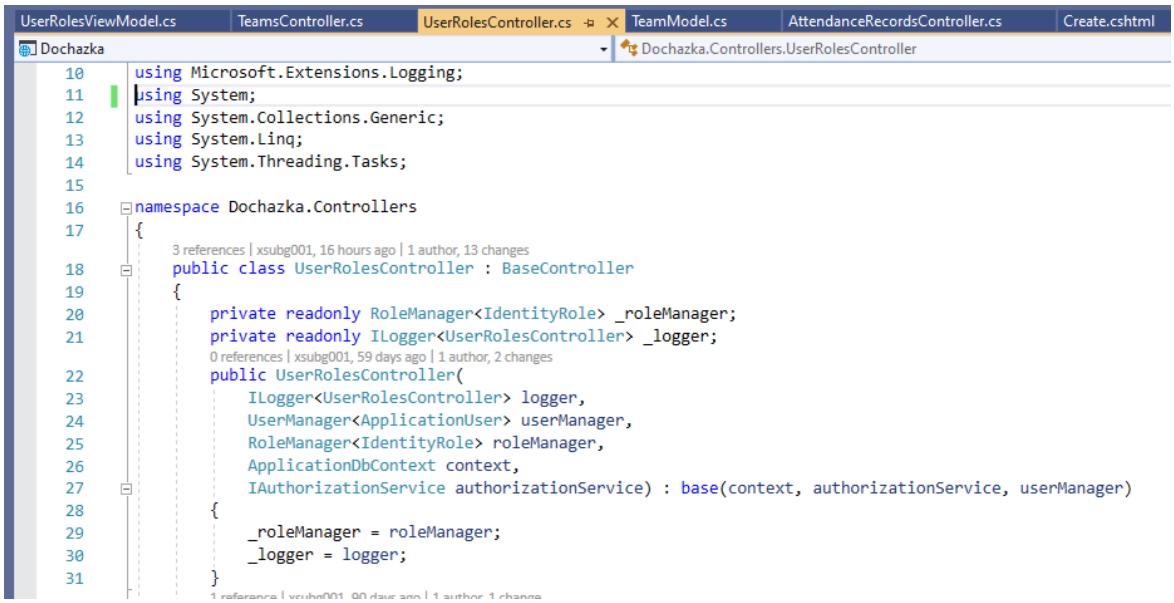
```
private readonly ILogger<UserRolesController> _logger;
```

- Nastavení loggeru třídy `<UserRolesController>` pomocí Dependency Injection, který se obvykle používá pro logování interních zpráv pro vývojáře. Nicméně v této třídě není aktivně použito.

```
private readonly RoleManager<IdentityRole> _roleManager;
```

- Tento lokální člen poskytuje API pro přístup k rolím aplikace, jež jsou řízeny prostřednictvím knihovny ASP.NET Core Identity diskutované dříve. V Controlleru se pomocí tohoto objektu čtou všechny dostupné role.

Obrázek 38 - Konstruktor třídy UserRolesController.cs




```
public async Task<IActionResult> Index(string sortOrder, string currentFilter, string searchString, int? pageNumber)
```

- `HttpGet` metoda, která zajišťuje výpis matice uživatelských účtů a jim přiřazených rolí. Metoda pracuje s `UserViewModel.cs` a zobrazí View ze složky `/Views/UserRoles/Index.cshtml`.

```
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Manage(string id)
```

- `HttpGet` metoda, která zobrazí formulář pro editaci přiřazených systémových rolí a přiřazení uživatele do teamu. Metoda pracuje s modelem `ManageUserViewModel.cs` a zobrazí View ze složky `/Views/UserRoles/Manage.cshtml`. Přístup k této i některým dalším metodám `UserRolesControlleru` je také zabezpečen atributem `[Authorize(Roles = "TeamAdministratorRole")]`, aby operaci mohli volat pouze oprávnění uživatelé.

```
[HttpPost]
[ValidateAntiForgeryToken]
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Manage(ManageUserViewModel input)
```

- `HttpPost` metoda, která zpracuje vstup formuláře z View `Manage.cshtml`, tedy model `ManageUserViewModel`, a na jeho základě provede validaci dat a případnou úpravu v přiřazení rolí danému uživateli, resp. změnu jeho členství v teamu.

```
// GET: UserRoles/Delete/5
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> Delete(string id)
```

- `HttpGet` metoda, která zobrazí formulář pro smazaní vybraného uživatele. Metoda pracuje s `UserViewModel.cs` a zobrazí View `/Views/UserRoles/Delete.cshtml`.

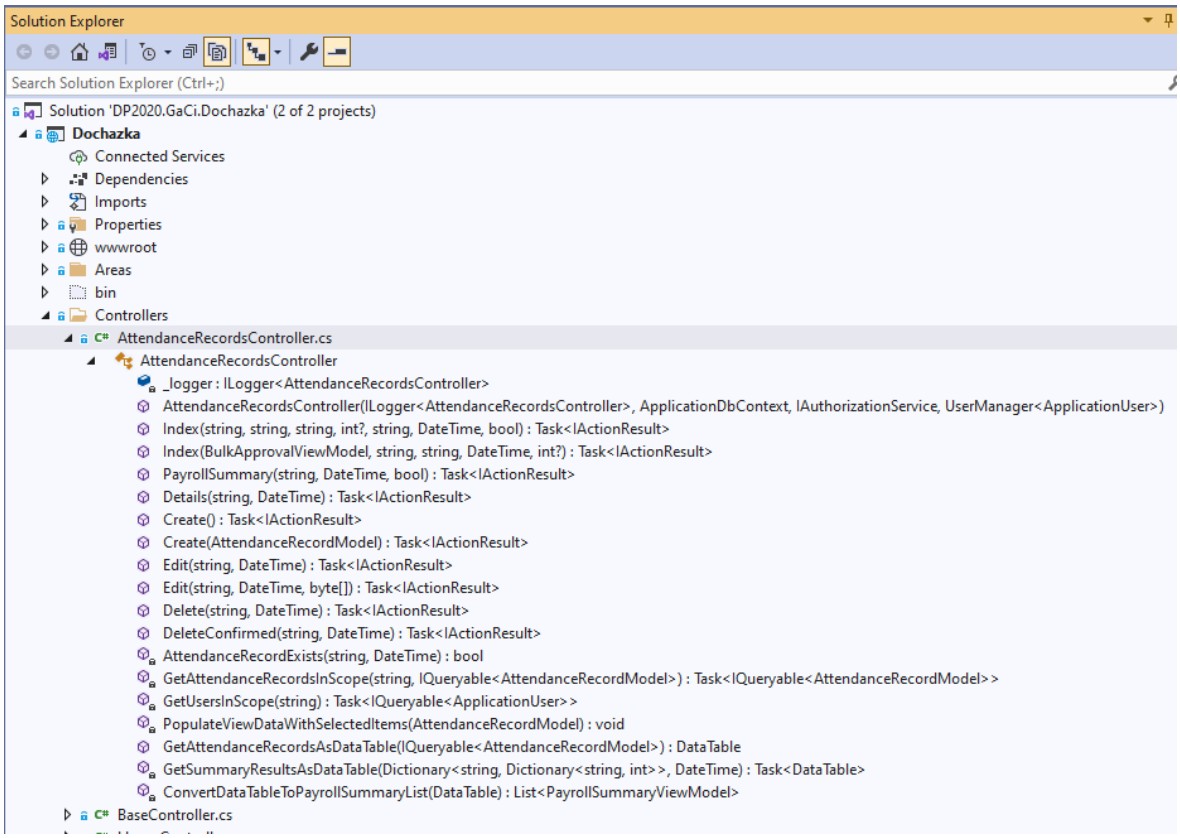
```
// POST: UserRoles/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
[Authorize(Roles = "TeamAdministratorRole")]
public async Task<IActionResult> DeleteConfirmed(string id, UserRolesViewModel user)
```

- `HttpPost` metoda, která zpracuje vstup formuláře z View `Delete.cshtml` a pokud jsou jeho vstupní data validována, dojde k odebrání rolí danému uživateli, smazání uživatele a přesměrování na akci `Index` v případě úspěšného dokončení. Metoda také ošetřuje výjimku `DbUpdateConcurrencyException`, která by mohla nastat, pokud byl mazaný záznam v mezidobí aktualizován jiným uživatelem.

4.2.1.5.5 Třída AttendanceRecordsController.cs

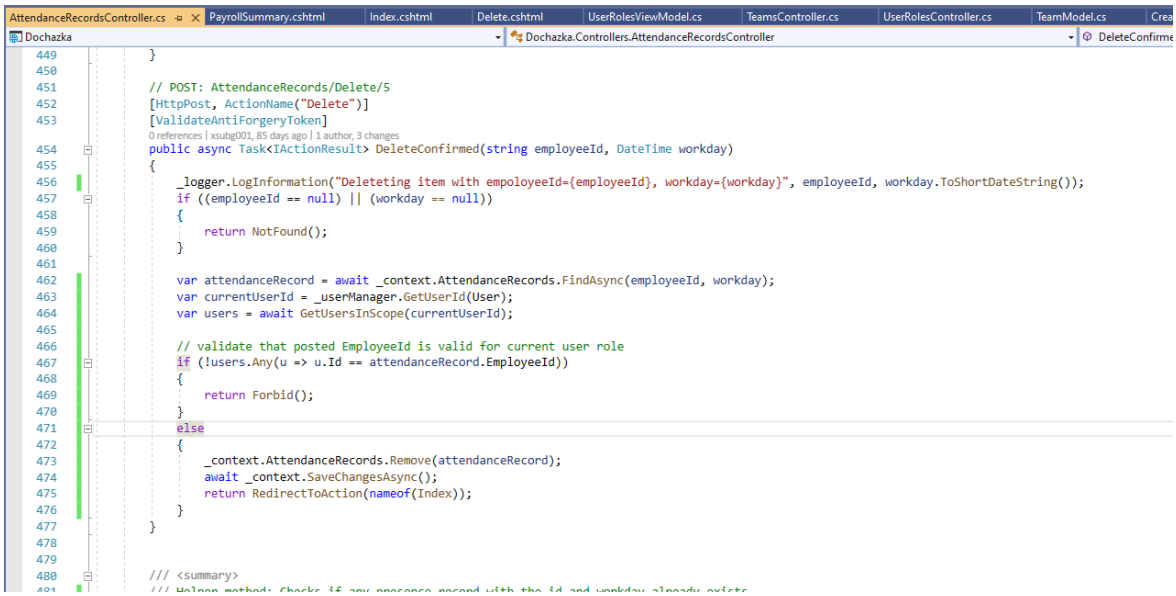
Tento, zřejmě nejsložitější Controller celé aplikace, má na starosti akce související s evidencí pracovní docházky (`attendanceRecords`) a rovněž dědí z `BaseController.cs`. Třída čítá několik pomocných i akčních metod. Jejich přehled ukazuje Obrázek 39.

Obrázek 39 - Solution Explorer pohled na třídu AttendanceRecordsController.cs



Konstruktor třídy není příliš zajímavý. Podobně jako u ostatních Controlleru v něm dochází pouze k inicializaci vlastního člena `ILogger<AttendanceRecordsController> _logger` pomocí Dependency Injection a zavolání rodičovského konstruktoru. Objekt `_logger` je v několika akčních metodách použit k logování zpráv, např. v metodě `DeleteConfirmed` na ř. 456, viz Obrázek 40.

Obrázek 40 - Zdrojový kód metody DeleteConfirmed



Nyní blíže k dalším metodám Controlleru.

4.2.1.5.5.1 Pomocné metody Controlleru AttendanceRecordsController.cs

```
private bool AttendanceRecordExists(string id, DateTime workday)
```

- Jednoduchá metoda, která vrací `bool` podle toho, jestli v databázi již existuje `AttendanceRecord` s kompozitním primárním klíčem `string id, DateTime workday`.

```
private async Task<IQueryable<AttendanceRecordModel>>
GetAttendanceRecordsInScope(string currentUserId, IQueryable<AttendanceRecordModel>
attendanceRecords)
```

- Asynchronní metoda, která filtruje vstupní seznam záznamů docházky, tj. `IQueryable<AttendanceRecordModel> attendanceRecords`, a vrací aktualizovaný seznam záznamů docházky `IQueryable<AttendanceRecordModel>` podle role přihlášeného uživatele s `currentUserId`. Uživatel s rolí `TeamAdministratorRole` má přístup ke všem záznamům docházky (`attendanceRecords`) všech uživatelů, uživatel s roli `TeamManagerRole` má přístup pouze ke všem záznamům docházky všech svých podřízených a uživatel pouze s rolí `TeamMemberRole` má přístup jen k vlastním záznamům docházky. Všechny ostatní záznamy jsou z původního vstupního seznamu vyloučeny podle role přihlášeného uživatele. Metoda je použita v akčních metodách `Index` a `PayrollSummary`.

```
private async Task<IQueryable<ApplicationUser>> GetUsersInScope(string
currentUserId)
```

- Asynchronní metoda, která vrací seznam uživatelů, pro které může aktuálně přihlášený uživatel s `currentUserId` vytvářet, editovat a smazat záznamy docházky (`attendanceRecord`). Logika je obdobná jako v předchozí metodě, tj. uživatel s rolí `TeamAdministratorRole` může vytvářet a modifikovat záznam docházky pro jakéhokoliv jiného uživatele aplikace, uživatel s rolí `TeamManagerRole` může vytvářet a modifikovat záznam docházky pouze pro své podřízené a uživatel pouze s rolí `TeamMemberRole` může vytvářet a modifikovat jen své vlastní záznamy. Metoda je použita v akcích Controlleru `Create`, `Edit`, `Delete`.

```
private void PopulateViewDataWithSelectedItems(AttendanceRecordModel
attendanceRecord)
```

- Pomocná metoda, která připravuje `ViewData Dictionary` pro akční metody `Create` a `Edit`, s jehož pomocí jsou generovány vstupní formuláře v odpovídajícím View.

```
public static DataTable
GetAttendanceRecordsAsDataTable(IQueryable<AttendanceRecordModel> attendanceRecords)
```

- Tato metoda konvertuje vstupní seznam `IQueryable<AttendanceRecordModel> attendanceRecords` do nové datové struktury `System.Data.DataTable`. Tato datová struktura je pak použita pro vyexportování záznamů docházky do *.csv souboru. Využití je v akční metodě `Index`.

```
private async Task<DataTable> GetSummaryResultsAsDataTable(
      Dictionary<string, Dictionary<string, int>> byEmployeeIDResults, DateTime
selectedMonth)
```

- Obdobná metoda jako výše. Opět konvertuje strukturu `Dictionary<string, Dictionary<string, int>> byEmployeeIDResults` do datové struktury `System.Data.DataTable`, jež je později použita pro vyexportování agregovaných dat docházky do *.csv souboru. Využití je v akční metodě `PayrollSummary`.

```
private List<PayrollSummaryViewModel> ConvertDataTableToPayrollSummaryList(DataTable
exportTable)
```

- Jde opět o konverzní metodu, která transformuje výstup předchozí metody `GetSummaryResultsAsDataTable` na seznam `List<PayrollSummaryViewModel>`. Ten je následně zpracován akční metodou `PayrollSummary` pro zobrazení odpovídajícího View, jež pracuje s ViewModelem `List<PayrollSummaryViewModel>`.

4.2.1.5.5.2 Akční metody Controlleru AttendanceRecordsController.cs

```
public async Task<IActionResult> Index(string sortOrder, string currentFilter, string searchString, int? pageNumber, string infoMessage, DateTime selectedMonth, bool getAsCsv)
```

- `HttpGet` metoda, která je zodpovědná za výpis záznamů docházky, jejich filtrování, třídění a případný export do souboru *.csv. Metoda má řadu vstupních filtračních a třídících parametrů, jež umožňují měnit podobu základního výpisu. V metodě je také parametr `bool getAsCsv`, který je nastaven přes odkaz ve View a indikuje, jestli uživatel potřebuje aktuálně zobrazený seznam záznamů docházky vyexportovat jako *.csv soubor, tzn. že mu je následně nabídnut soubor ke stažení.
- Součástí asociovaného View `/Views/AttendanceRecords/Index.cshtml` je také formulář, který umožňuje oprávněným uživatelům s rolí `TeamRoleManager`, nebo `TeamRoleAdministrator` hromadně schválit záznamy docházky, k nimž jsou oprávnění na základě pomocné metody `GetAttendanceRecordsInScope`. Akční metoda `Index` pracuje s ViewModelem `PaginatedList<AttendanceRecordModel>`, který je uzpůsoben pro stránkování výpisu.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Index(BulkApprovalViewModel bulkApprovals, string currentFilter, string sortOrder, DateTime selectedMonth, int? pageNumber)
```

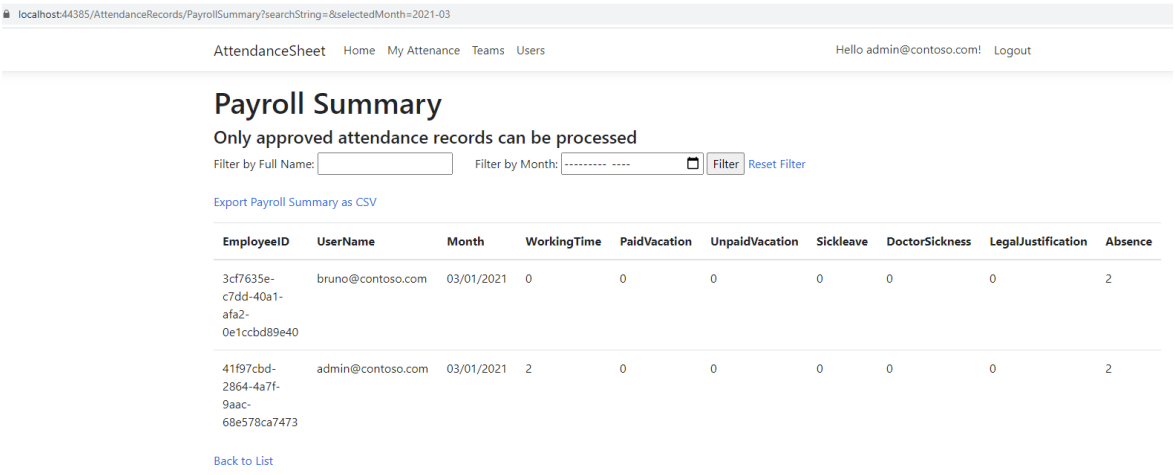
- `HttpPost` metoda, která umožňuje zpracovat data z formuláře zakomponovaného do View `/Views/AttendanceRecords/Index.cshtml`, jež umožňuje hromadné schválení záznamů docházky oprávněným uživatelem. Metoda pracuje na vstupu s ViewModelem `BulkApprovalViewModel bulkApprovals` a s dalšími parametry, které určují nastavení filtru zobrazení. Na konci metody je uživatel přesměrován zpět na akční metodu `HttpGet Index` s původním nastavením zobrazovacích filtrů

a informační zprávou `infoMessage`, jež zobrazí ve View `Index.cshtml` informaci o tom, jaký byl výsledek předchozí schvalovací operace.

```
public async Task<IActionResult> PayrollSummary(string searchString, DateTime selectedMonth, bool getAsCsv)
```

- `HttpGet` metoda slouží k zobrazení ViewModelu `List<PayrollSummaryViewModel>` pro vybraný kalendářní měsíc prostřednictvím View ze složky `/Views/AttendanceRecords/PayrollSummary.cshtml`. Alternativně lze tento souhrn vyexportovat do *.csv souboru na základě vstupního parametru metody `bool getAsCsv`, který je nastaven přes odkaz ve View.
- Stěžejní část teto metody spočívá v použití několika C# LINQ `GroupBy` výrazů, jež jsou použity k agregaci vybraných záznamů docházky z databázového contextu `_context.AttendanceRecords` do datové struktury typu `Dictionary<string, Dictionary<string, int>>`, kde klíče vnějšího `Dictionary` jsou nastaveny podle `attendanceRecord.EmployeeId` a jejich hodnoty tvoří vnitřní `Dictionary<string, int>`. V něm jsou pak klíčovány součty hodnot jednotlivých druhů docházky za vybraný kalendářní měsíc. Obrázek 41 ukazuje výsledný výpis.

Obrázek 41 - Výpis agregovaného výpisu PayrollSummary



```
public async Task<IActionResult> Details(string employeeId, DateTime workday)
```

- `HttpGet` metoda, jež slouží k zobrazení View `/Views/AttendanceRecords/Details.cshtml`, jež zobrazí podrobnosti o daném záznamu docházky. Tato akce a View pracují s modelem `AttendanceRecordModel.cs`.

```
public async Task<IActionResult> Create()
```

- `HttpGet` metoda, která slouží k zobrazení formuláře pro model `AttendanceRecordModel.cs`. Jeho prostřednictvím může uživatel vytvořit nový záznam pracovní docházky (`attendanceRecord`) pro sebe, případně další uživatele, což je dáno dle jemu přidělené role v rámci aplikace, viz metoda `GetUsersInScope(currentUserId)` diskutovaná v 4.2.1.5.5.1. Tato akční metoda zobrazí formulář pomocí View `/Views/AttendanceRecords/Create.cshtml`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("EmployeeId, WorkDay,
MorningAttendance, AfternoonAttendance")] AttendanceRecordModel attendanceRecord)
```

- `HttpPost` metoda, která na vstupu zpracuje model `AttendanceRecordModel attendanceRecord` a v případě úspěšné validace modelu vytvoří novou entitu záznamu docházky v databázi. Předmětem validace je to, zda uživatel má oprávnění vytvořit záznam docházky s daným `EmployeeId` a také ověření, že v databázi ještě neexistuje jiný záznam se stejným kompozitním primárním klíčem, tj. `EmployeeId` a `WorkDay`. Po úspěšném uložení záznamu do databáze je uživatel přesměrován na akci `Index`. V opačném případě je mu znovu zobrazeno View `Create.cshtml` s chybovou zprávou, např. že záznam docházky pro daný den a uživatele již existuje.

```
public async Task<IActionResult> Edit(string employeeId, DateTime workday)
```

- `HttpGet` metoda, která slouží k zobrazení formuláře pro model `AttendanceRecordModel.cs`, jehož prostřednictvím může uživatel editovat záznam pracovní docházky (`attendanceRecord`) pro sebe, případně další uživatele, což je dáno dle jemu přidělené role v rámci aplikace, viz metoda `GetUsersInScope(currentUserId)`. Metoda zobrazí formulář pomocí View `/Views/AttendanceRecords/Edit.cshtml`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(string employeeId, DateTime workday, byte[]
rowVersion)
```

- `HttpPost` metoda, jejímž úkolem je aktualizovat pracovní záznam docházky v databázi daný modelem `AttendanceRecordModel.cs`, pakliže jsou na vstupu poskytnuty validní hodnoty kompozitního primárního klíče (`employeeId`, `workday`). Metoda zároveň detekuje výjimku `DbUpdateConcurrencyException`, pokud by došlo v mezidobí k aktualizaci záznamu docházky jiným uživatelem. Po úspěšné aktualizaci záznamu v databázi je uživatel přesměrován na akci `Index`. Tato akční metoda také využívá metodu `GetUsersInScope(currentUserId)`.

```
public async Task<IActionResult> Delete(string employeeId, DateTime workday)
```

- `HttpGet` metoda, která slouží k zobrazení formuláře pro model `AttendanceRecordModel.cs`, jehož prostřednictvím může uživatel smazat jeden konkrétní záznam pracovní docházky (`attendanceRecord`) pro sebe, případně další uživatele, což je dáno dle jemu přidělené role v rámci aplikace, viz metoda `GetUsersInScope(currentUserId)`. Metoda zobrazí formulář pomocí View ze složky `/Views/AttendanceRecords/Delete.cshtml`.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(string employeeId, DateTime
workday)
```

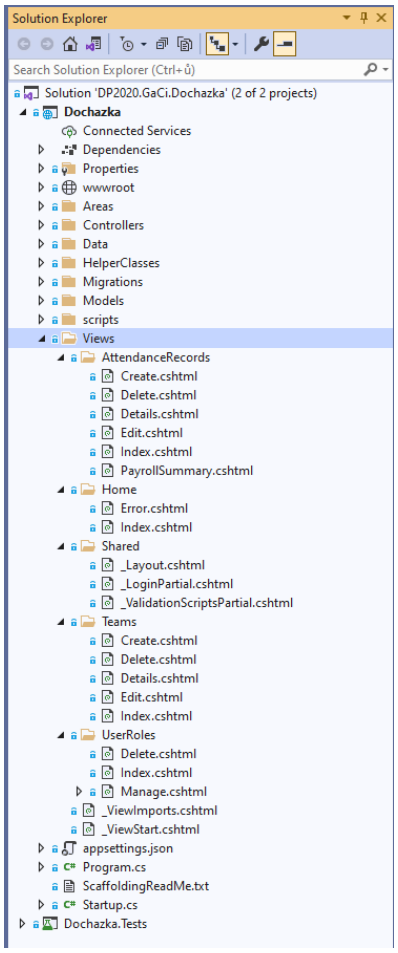
- `HttpPost` metoda, která smaže vybraný záznam pracovní docházky z databáze, pokud přihlášený uživatel je v roli `TeamAministratorRole`, nebo `TeamManagerRole`. Daný záznam musí existovat a přihlášený uživatel musí mít k němu odpovídající oprávnění, viz metoda `GetUsersInScope(currentUserId)`.

4.2.1.6 Složka Views

Ve složce jsou uloženy Razor *.cshtml soubory všech Views, jež jsou asociovány s akčními metodami Controllerů a generují odpovídající HTML kód stránek.

Jednotlivé View soubory, které ve složce existují, ukazuje Obrázek 42.

Obrázek 42 - Solution Explorer pohled na složku Views



Soubory `/Views/*.cshtml` a `Views/Shared/*.cshtml` byly vygenerovány jako součást skeletonu projektu, resp. s přidáním knihovny Asp.Net Core Identity, např. `_LoginPartial.cshtml`. Slouží k sestavení základní kostry HTML stránek včetně hlavního navigačního menu.

V dalších složkách jsou pak jednotlivé Razor soubory asociované s Controllery diskutovanými dříve:

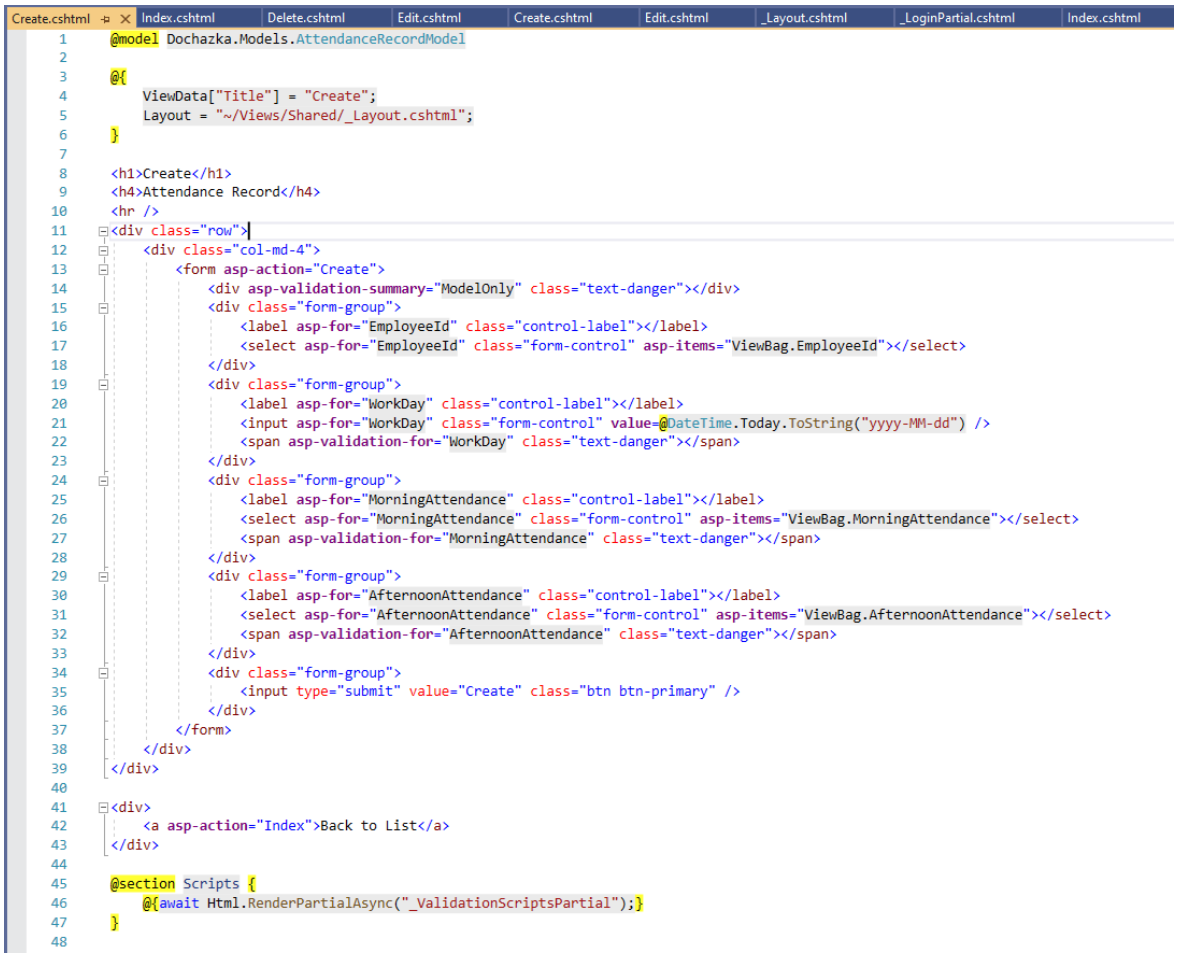
```
/Views/Home/*.cshtml
/Views/Teams/*.cshtml
/Views/UserRoles/*.cshtml
/Views/AttendanceRecords/*.cshtml
```

Jelikož není možné podrobně procházet všechny Razor soubory, které jsou navíc celkem srozumitelné a navzájem si podobné, budou z nich podrobněji analyzovány pouze dva vzorové soubory.

4.2.1.6.1 Razor soubor /Views/AttendanceRecords/Create.cshtml

View je použito pro vygenerování HTML stránky s formulářem, v němž může uživatel vytvořit nový záznam pracovní docházky. Je voláno akční metodou `HttpGet Create` z `AttendanceRecordsController.cs`. Zdrojový kód ukazuje Obrázek 43.

Obrázek 43 - Razor zdrojový kód /Views/AttendnaceRecords/Create.cshtml



- Na ř. 1 je pomocí C# kódu definován model tohoto View, tj. `Dochazka.Models.AttendanceRecordModel`.
- Na ř. 4-5 jsou opět pomocí C# kódu nastaveny další vlastnosti View, jako rozložení stránky odkazující na sdílený `_Layout.cshtml` a titulní název stránky.
- Na ř. 8-9 jsou vygenerovány nadpisy stránky pomocí standardních HTML tagů.
- Na ř. 11-39 je sekce, která definuje vstupní formulář.
    - Ř.13 otevírá úvodní značku formuláře a specifikuje pomocí ASP.NET Core Tagu `asp-action`, že vstup formuláře bude volat akční metodu `HttpPost Create`.

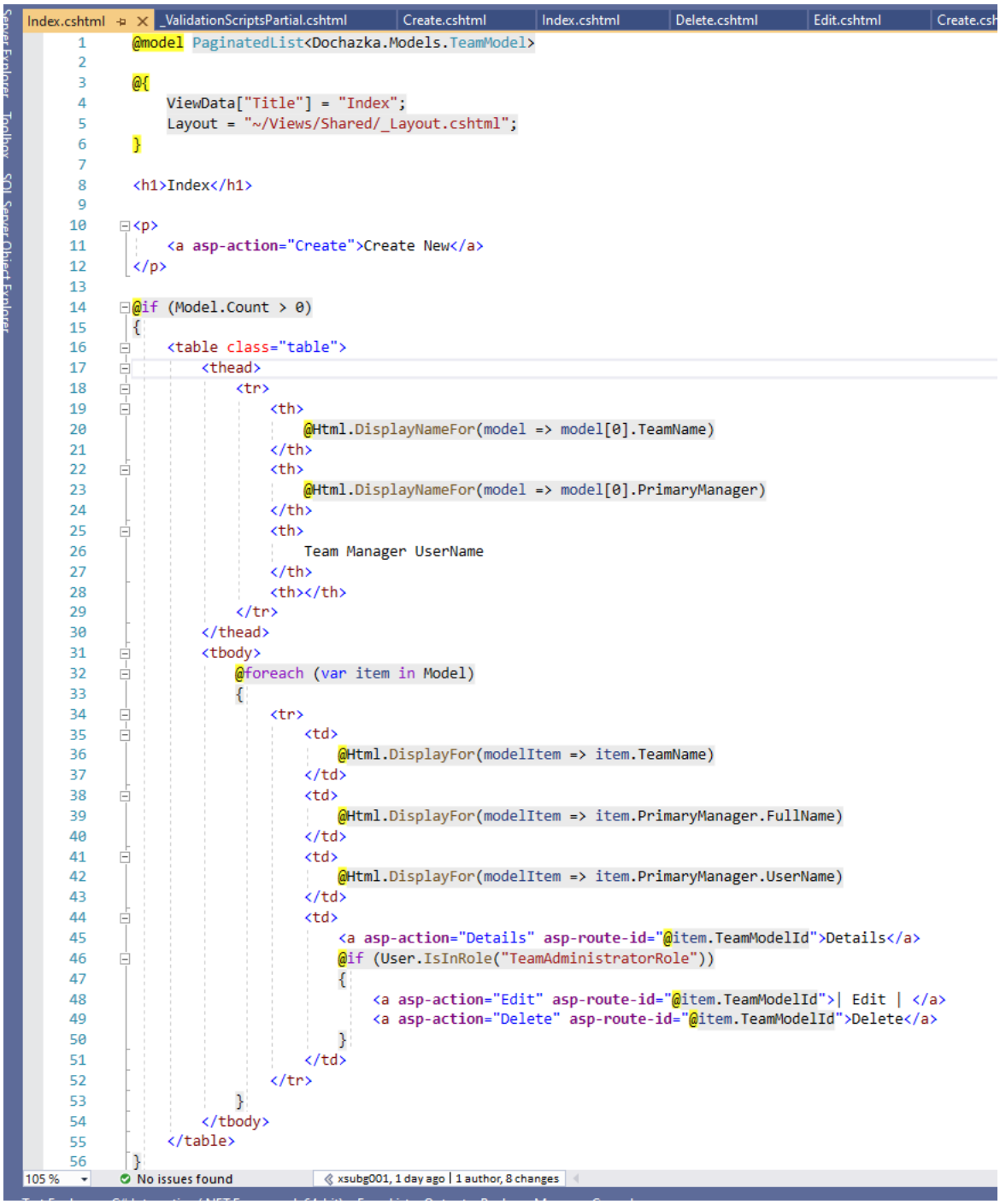
- Ř. 14, 22, 27 a 32 aktivují validaci vstupních dat formuláře a zobrazení případných validačních chyb.
- Ř. 15-18, 19-23, 24-28, 29-33 definují jednotlivá vstupní pole formuláře, která slouží k zadání potřebných dat pro model. Vstupy jsou většinou založeny na HTML elementu `Select`, s výjimkou pole `Workday`, kde je vstup automaticky vygenerován jako HTML kalendář díky atributu `[DataType(DataType.Date)]`, jenž je uveden u property `WorkDay` v definici modelu `AttendanceRecordModel.cs`.
- Ř. 35 pak vytvoří tlačítko formuláře, které formulář odešle na akční metodu `HttpPost Create`.

- Na ř. 41-44 je vytvořen odkaz na akci `Index` tohoto Controlleru
  - Na ř. 45-48 se do stránky vkládají JavaScript jQuery validační skripty prostřednictvím souboru dílčího View (Partial View) `/Views/Shared/_ValidationScriptsPartial.cshtml`, jež na ně odkazuje. Tato sekce je automaticky přidána průvodcem Visual Studia během generování skeletonu projektu pro všechny Views Create a Edit.

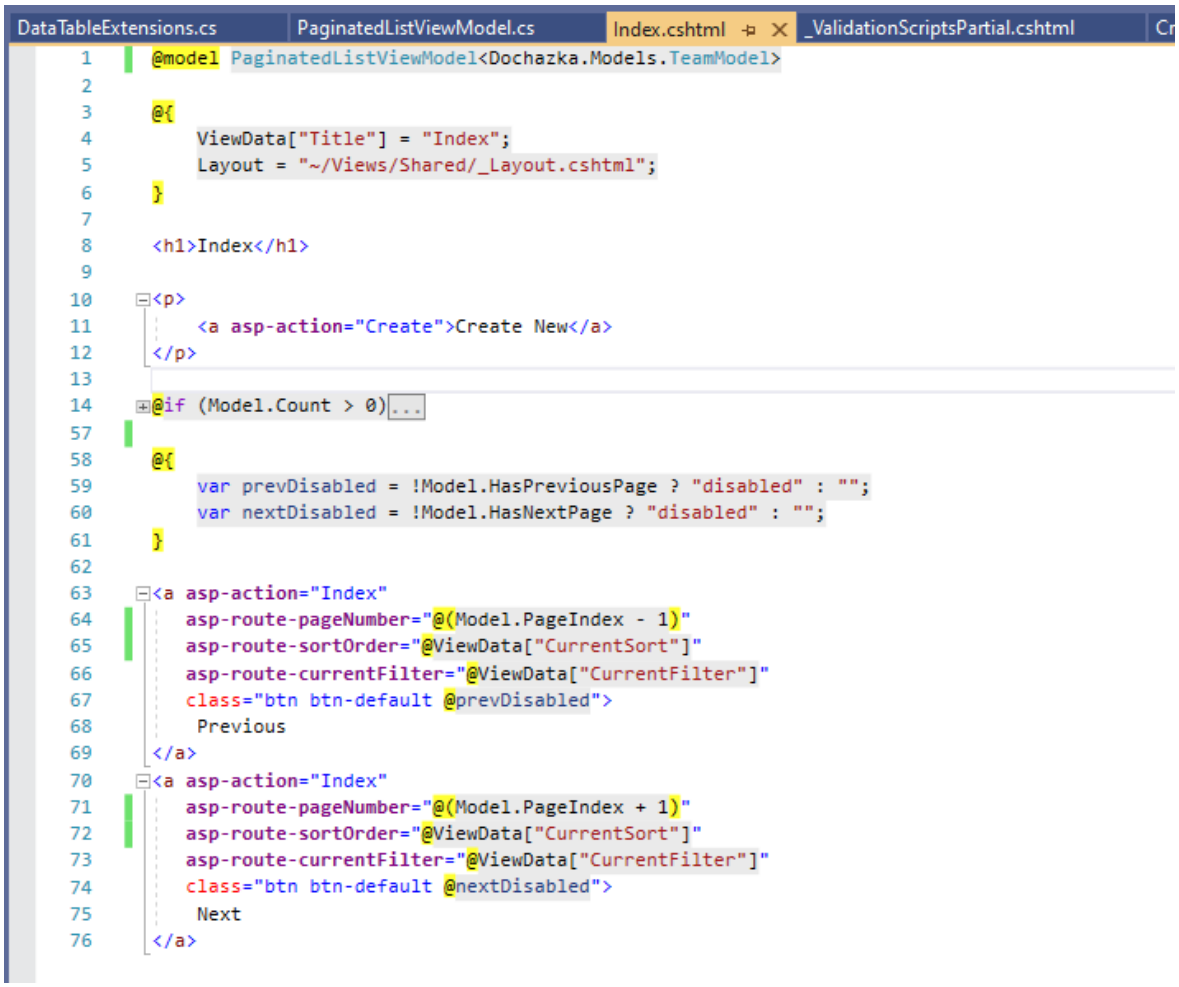
4.2.1.6.2 Razor soubor `/Views/Teams/Index.cshtml`

View je použito pro vygenerování HTML stránky, na níž je zobrazen stránkovaný výpis všech teamů aplikace. Zdrojový kód zachycují Obrázek 44 a Obrázek 45.

Obrázek 44 - Razor zdrojový kód /Views/Teams/Index.cshtml, část 1



Obrázek 45 - Razor zdrojový kód /Views/Teams/Index.cshtml, část 2



- Ř.1 pomocí C# kódu definuje ViewModel, který je použit pro toto View. V tomto případě jde o speciální generický model, `@model PaginatedList<Dochazka.Models.TeamModel>`, jenž je podrobněji vysvětlen v odstavci 4.2.1.4.11.
- Ř. 3-6 jsou rovněž podobné, jako v přechozím příkladě, a definují titulní název a hlavní rozložení stránky pomocí `_Layout.cshtml`.
- Ř. 14-56 jsou pěknou ukázkou propojení C# kódu, HTML a Razor značek Tag Helper. Tento kód vygeneruje HTML tabulku se čtyřmi sloupci, v jejímž záhlaví (značka `<thead></thead>`), je jeden řádek (značka `<tr></tr>`). Ten má ve dvou sloupcích (značka `<th></th>`) nastavenou hodnotu podle jména property modelu, nebo volitelného atributu `DisplayName`, jenž danou property může dekorovat. Např. `@Html.DisplayNameFor(model => model[0].PrimaryManager)` doplní

hodnotu „Team Manager“, viz třída modelu `TeamModel.cs`. Dva zbývající sloupce mají nastavenou fixní hodnotu přímo ve View.

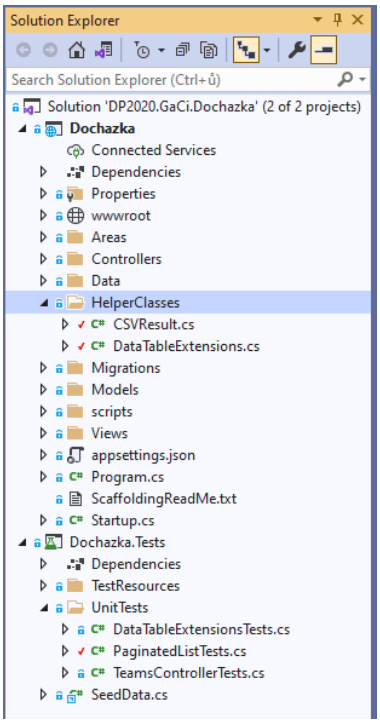
- Na ř. 32-54 je pak dynamicky C# cyklem `@foreach (var item in Model)` vygenerován takový počet řádků datové části tabulky (značka `<tbody></tbody>`), jaký je počet prvků ve vstupním modelu. Jako hodnoty v sloupcích tabulky jsou použity skutečné hodnoty properties v modelu, např. `@Html.DisplayFor(modelItem => item.TeamName).`
- V posledním sloupci datové části tabulky jsou vygenerovány odkazy na akce `Details`, `Edit`, `Delete` stejného Controlleru, s doplňkovým parametrem `id=@item.TeamModelId`, kdy odkazy pro akci `Edit` a `Delete` jsou podmíněny tím, že přihlášený uživatel musí mít roli `TeamAdministratorRole`, viz ř. 46.

- Na ř. 58-61 je opět C# kód, jenž nastaví dvě lokální proměnné `prevDisabled` a `nextDisabled` podle hodnot modelových properties `Model.HasPreviousPage` a `Model.HasNextPage`. Tyto lokální proměnné jsou dále použity níže pro aktivaci odkazů pro stránkování výpisu.
- Na ř. 63-69 je vygenerován odkaz „Previous“ pro přechod na předchozí stránku výpisu. Odkazuje tedy na tutéž akci `Index` téhož Controlleru a předává vstupní parametry indikující, jaké číslo stránky má být zobrazeno, viz ř. 64 a také, jaké je aktuální nastavení vstupních filtrů, viz ř. 65-66. Podle aktuální hodnoty lokální proměnné `prevDisabled` je tlačítko pak buď aktivní nebo deaktivované, viz ř. 74.
- Na ř. 70-76 je obdobný kód použit pro vytvoření HTML odkazu „Next“, jenž zobrazí následující stránku, pokud je k dispozici v závislosti na lokální proměnné `nextDisabled`, resp. modelové property `Model.HasNextPage`.

4.2.1.7 **Složka HelperClasses**

Ve složce jsou dvě pomocné třídy, viz Obrázek 46.

Obrázek 46 - Solution Explorer pohled na složku HelperClasses



4.2.1.7.1 Třída CSVResult.cs

Obrázek 47 ukazuje zdrojový kód třídy, jejímž účelem je naplnit objekt třídy `HttpResponse` aktuálního HTTP contextu (dotazu), ř. 26. Ten bude obsahovat vyexportovaný *.csv soubor jako bytové pole. Hlavní část HTTP odpovědi (`response.Body`) je vygenerována pomocnou třídou `DataTableExtensions.cs` a její metodou `WriteToCsvString()`, viz ř. 29-30, jež bude diskutována níže. Řešení bylo inspirováno diskusí na fóru stackoverflow.com [27].

Obrázek 47 - Zdrojový kód třídy CSVResult.cs

CSVResult.cs | Create.cshtml | AttendanceRecordsController.cs | UserRolesController.cs | TeamsController.cs | _Layout.cshtml | HomeController.cs

Dochazka | Dochazka.HelperClasses.CSVResult | Table

```
1   using System.Data;
2   using Microsoft.AspNetCore.Mvc;
3   using Microsoft.AspNetCore.Http;
4   using System.Threading.Tasks;
5
6   namespace Dochazka.HelperClasses
7   {
        3 references | xsubg001, 95 days ago | 1 author, 1 change
8       public class CSVResult : ActionResult
9       {
10          /// <summary>
11          /// Converts the columns and rows from a data table into an Microsoft Excel compatible CSV file using custom DataTable Extension methods.
12          /// </summary>
13          /// <param name="dataTable"></param>
14          /// <param name="fileName">The full file name including the extension.</param>
            2 references | xsubg001, 95 days ago | 1 author, 1 change
15          public CSVResult(DataTable dataTable, string fileName)
16          {
17              Table = dataTable;
18              FileName = fileName;
19          }
20
            2 references | xsubg001, 95 days ago | 1 author, 1 change
21          public string FileName { get; protected set; }
            2 references | xsubg001, 95 days ago | 1 author, 1 change
22          public DataTable Table { get; protected set; }
23
            0 references | xsubg001, 95 days ago | 1 author, 1 change
24          public override Task ExecuteResultAsync(ActionContext context)
25          {
26              HttpResponse response = context.HttpContext.Response;
27              response.ContentType = "text/csv";
28              response.Headers.Add("Content-Disposition", "attachment;filename=" + this.FileName);
29              byte[] data = System.Text.Encoding.UTF8.GetBytes(Table.WriteToCsvString());
30              response.Body.WriteAsync(data, 0, data.Length);
31              return Task.CompletedTask;
32          }
33      }
34  }
35
```

4.2.1.7.2 Třída DataTableExtensions.cs

Tato třída rozšiřuje standardní metody třídy `DataTable` a zavádí novou statickou metodu třídy `DataTable.WriteToCsvString`. Ta umožňuje transformovat data z objektu typu `DataTable` na objekt typu `string`, který reprezentuje mnohořádkový řetězec, jehož jednotlivé řádky mají strukturu jako *.csv soubor, tzn. že sloupce jsou odděleny čárkou a hodnoty uzavřeny v uvozovkách.

Zdrojový kód této třídy zachycuje Obrázek 48.

Obrázek 48 - Zdrojový kód třídy DataTableExtensions

DataTableExtensions.cs | DataTableExtensionsTests.cs | WriteToCsvStringTest.csv | TeamsControllerTests.cs | PaginatedListTests.cs*

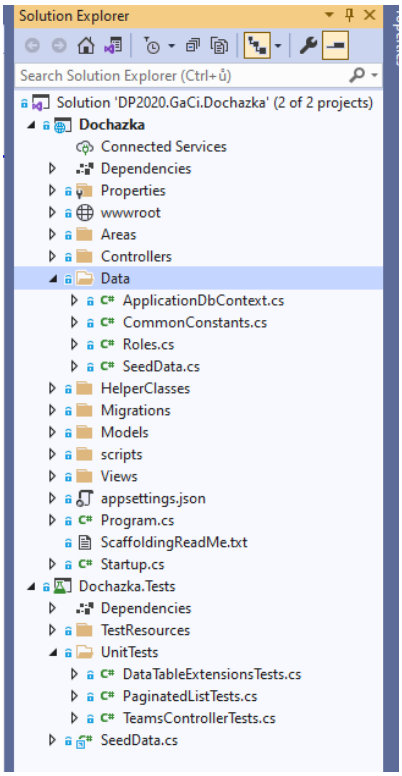
Dochazka | Dochazka.HelperClasses.DataTableExtensions

```
using System.Data;
using System.Text;


namespace Dochazka.HelperClasses
{
    0 references | xsubg001, 19 hours ago | 1 author, 3 changes
    public static class DataTableExtensions
    {
        2 references | 1/1 passing | xsubg001, 59 days ago | 1 author, 2 changes
        public static string WriteToCsvString(this DataTable dataTable)
        {
            StringBuilder sbData = new StringBuilder();

            // Only return Null if there is no structure.
            if (dataTable.Columns.Count == 0)
                return string.Empty;

            foreach (var col in dataTable.Columns)
            {
                if (col == null)
                    sbData.Append(",");
                else
                    sbData.Append("\"" + col.ToString().Replace("\"", "\"\"") + "\",");
            }

            sbData.Replace(",", System.Environment.NewLine, sbData.Length - 1, 1);

            foreach (DataRow dr in dataTable.Rows)
            {
                foreach (var column in dr.ItemArray)
                {
                    if (column == null)
                        sbData.Append(",");
                    else
                        sbData.Append("\"" + column.ToString().Replace("\"", "\"\"") + "\",");
                }
                sbData.Replace(",", System.Environment.NewLine, sbData.Length - 1, 1);
            }

            return sbData.ToString();
        }
    }
}

```

4.2.1.8 Složka Data

Ve složce nalezneme třídy, jež souvisí s inicializací datové vrstvy aplikace, tj. objektů sloužících pro přístup k databázi a pro její inicializaci. Obrázek 49 ukazuje všechny třídy, které se ve složce nachází.

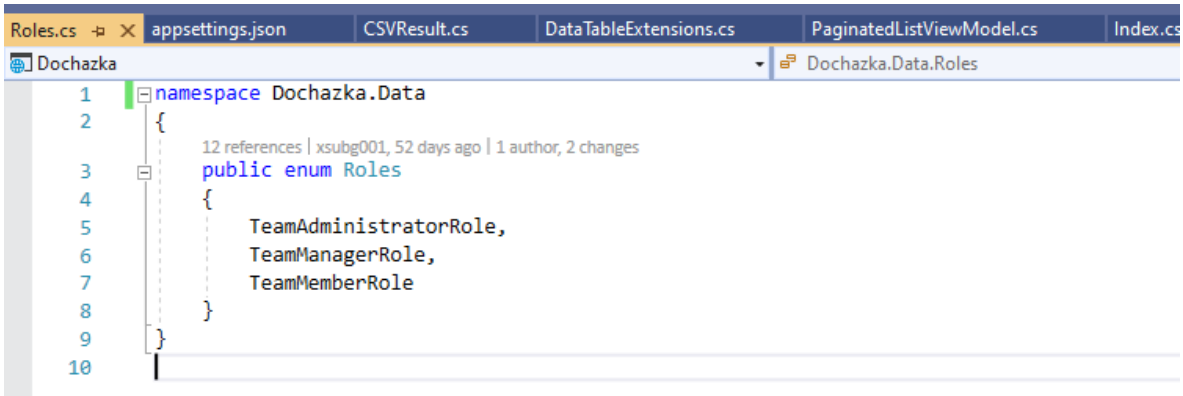
Obrázek 49 - Solution Explorer pohled na složku Data



4.2.1.8.1 Enum Roles.cs

Jde o jednoduchý `enum` typ, který definuje jména rolí, s nimiž bude aplikace pracovat. Zdrojový kód zachycuje Obrázek 50.

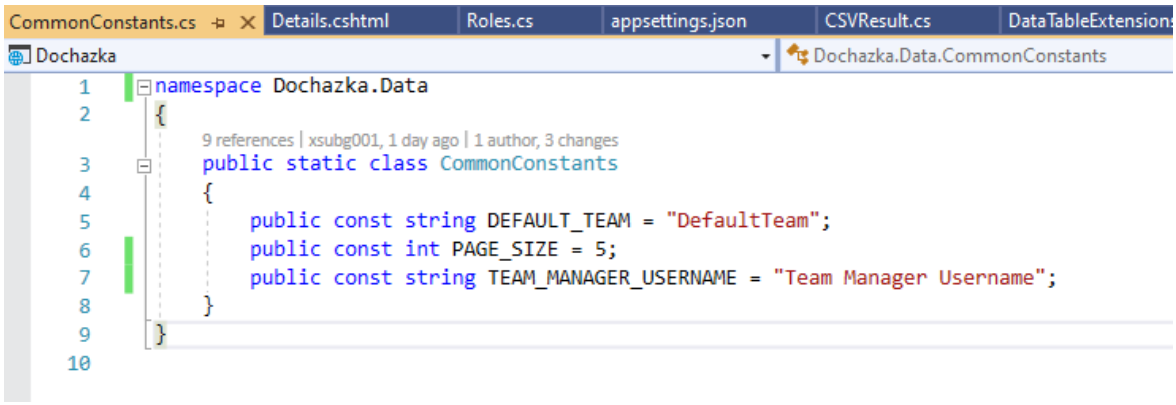
Obrázek 50 - Zdrojový kód enum typu Roles.cs



4.2.1.8.2 Enum CommonConstants.cs

Jde o jednoduchou třídu, která definuje tři konstanty, které se používají napříč aplikací. Zdrojový kód ukazuje Obrázek 51.

Obrázek 51 - Zdrojový kód třídy CommonConstants.cs



4.2.1.8.3 Třída ApplicationDbContext.cs

Jde o třídu, jejíž skeleton je na začátku projektu automaticky vygenerován průvodcem Visual Studia a dále aktualizován, tak jak jsou do projektu přidávaný nové entitní modely. Třída přidává (registruje) do Entity Framework Core databázového contextu entitní modelové třídy, tj. `AttendanceRecordModel` a `TeamModel`. Přístup k entitám modelu je pak zajištěn prostřednictvím dvou properties tohoto contextu:

```
public DbSet<AttendanceRecordModel> AttendanceRecords { get; set; }
public DbSet<TeamModel> Teams { get; set; }
```

Třída dědí z rodičovské třídy `IdentityDbContext<ApplicationUser>`, jež poskytuje Entity Framework Core databázový context pro knihovnu Identity.

Za zmínku stojí také nastavení kompozitního klíče pro entity modelu `AttendanceRecordModel`, viz ř. 22, a vytvoření Indexu podle property `TeamName` pro entity modelu `TeamModel`, viz ř. 25-26.

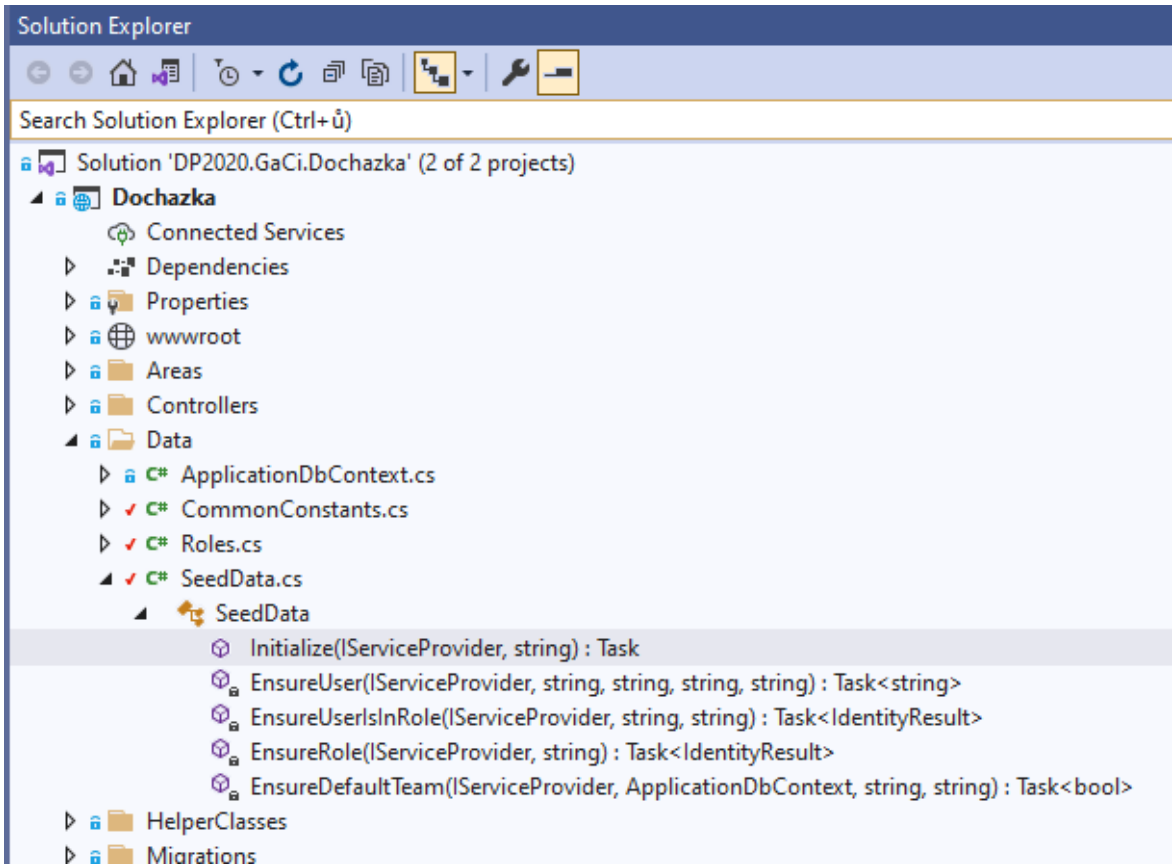
Celý zdrojový kód třídy zachycuje Obrázek 52.

Obrázek 52 - Zdrojový kód třídy ApplicationDbContext.cs

4.2.1.8.4 Třída SeedData.cs

Účelem této třídy je zajistit, že je databáze naplněna nezbytnými daty pro správnou funkci aplikace. Pro tento účel třída definuje několik metod, které se o to starají při každém spuštění aplikace. Pohled na třídu a její metody ukazuje Obrázek 53.

Obrázek 53 - Solution Explorer pohled na třídu SeedData.cs



Nyní blíže k jednotlivým metodám.

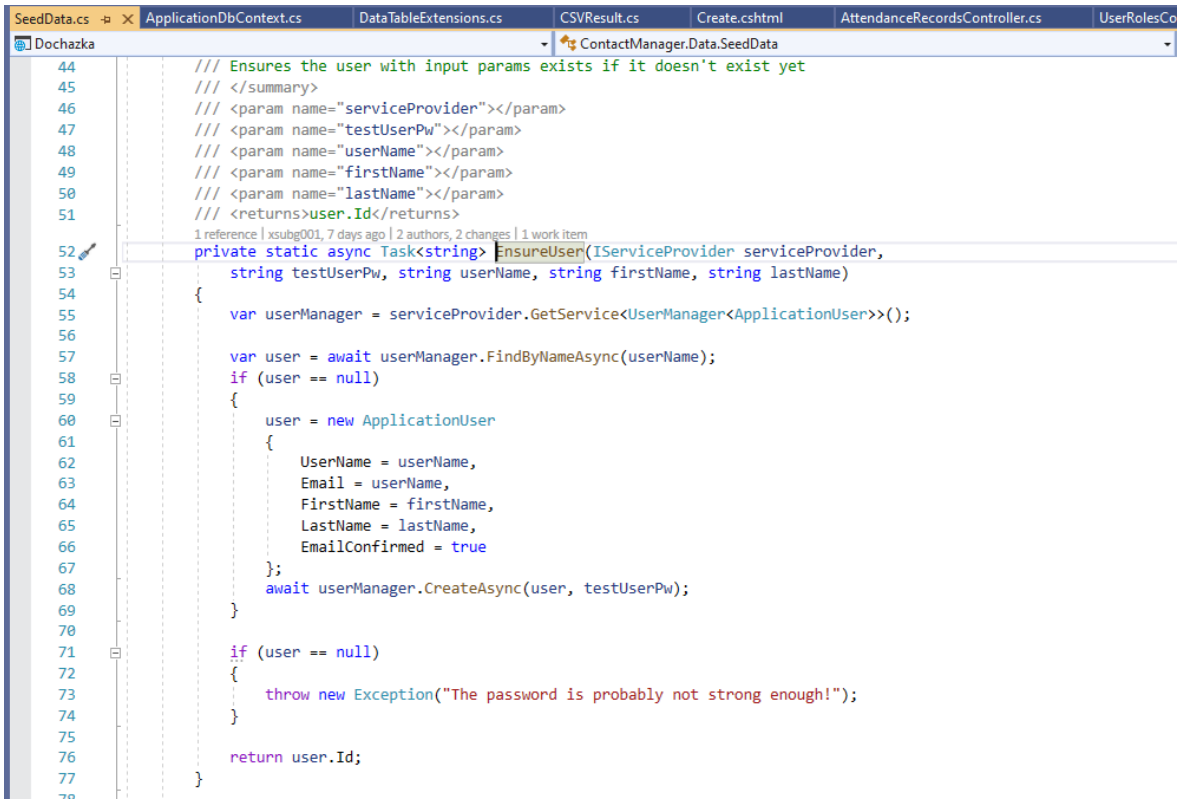
```
private static async Task<string> EnsureUser(IServiceProvider serviceProvider,
string testUserPw, string userName, string firstName, string lastName)
```

- Účelem metody je zajistit, že existuje uživatel podle daných vstupních parametrů.

  Metoda operuje s objektem typu:

  `Microsoft.AspNetCore.Identity.UserManager<ApplicationUser>`, jenž poskytuje API pro práci s uživateli. Zdrojový kód metody zachycuje Obrázek 54.

Obrázek 54 - Zdrojový kód metody EnsureUser ve třídě SeedData.cs



`private static async Task<IdentityResult> EnsureRole(IServiceProvider serviceProvider, string role)`

- Podobná metoda jako výše. Jejím účelem je zajistit, že existuje role s daným jménem. Metoda operuje s objektem typu: `Microsoft.AspNetCore.Identity.RoleManager< IdentityRole>`, jenž poskytuje API pro práci s rolemi.

`private static async Task<IdentityResult> EnsureUserIsInRole(IServiceProvider serviceProvider, string uid, string role)`

- Také podobná metoda jako výše, která má za úkol zajistit přidělení role uživateli podle vstupních parametrů `string uid` (id uživatele) a `string role` (název přidělené role)

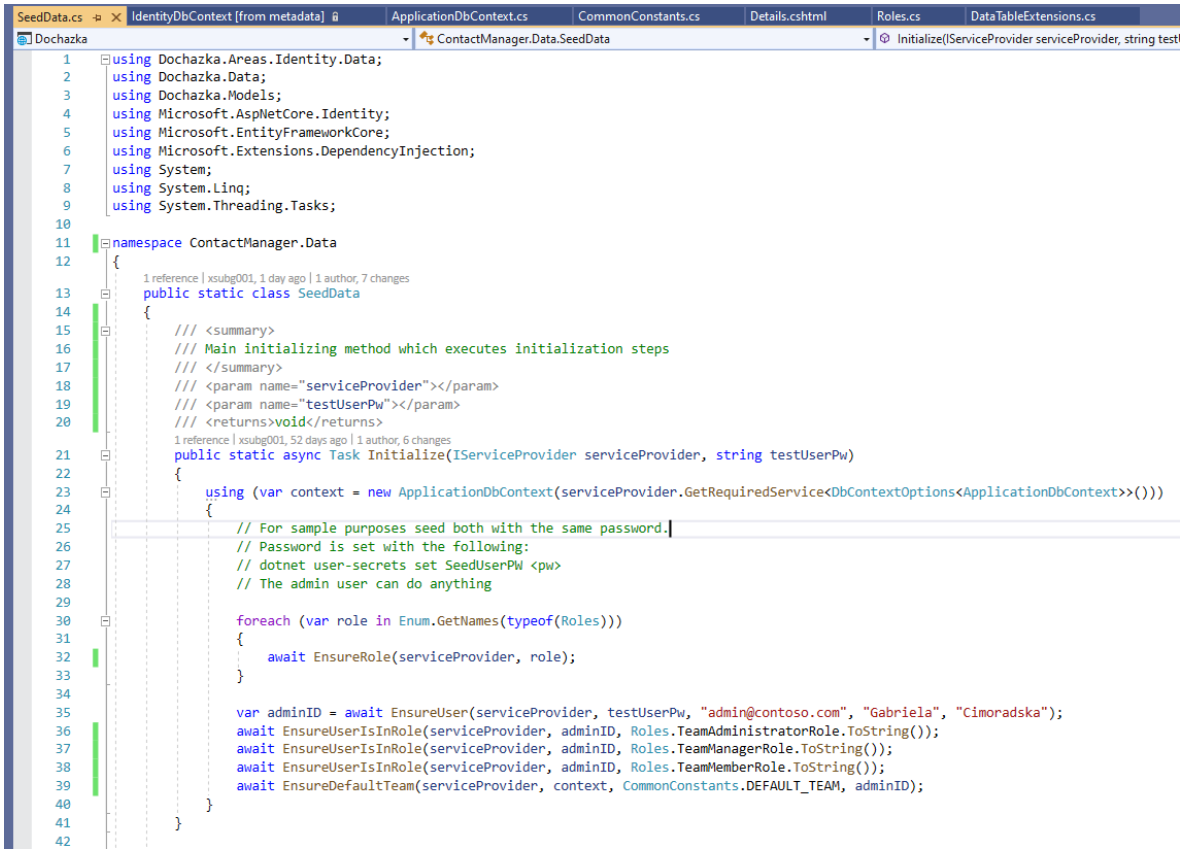
`private static async Task<bool> EnsureDefaultTeam(IServiceProvider serviceProvider, ApplicationDbContext context, string teamName, string primaryManagerId)`

- Úkolem metody je zajistit, že je vytvořen výchozí team aplikace a je mu přiřazen manažer podle vstupních parametrů metody `string teamName` a `string primaryManagerId`

`public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)`

- Tato metoda koordinuje všechny výše uvedené metody a realizuje úvodní plán inicializace databáze při spuštění aplikace, jelikož je volána z metody `Main` ve třídě `Program.cs`. Zdrojový kód metody ukazuje Obrázek 55.

Obrázek 55 - Zdrojový kód metody Initialize ve třídě SeedData.cs



4.2.1.9 Složka Migrations

V této složce jsou automaticky vygenerované tzv. migrační soubory (*.cs), které jsou použity pro inicializaci a konfiguraci databáze prostřednictvím Entity Frameworku Core. Migrační soubory jsou v podstatě jakési SQL Server automatizační skripty, které vytváří databázové schéma, tj. tabulky, nastavují klíče, indexy, omezení apod.

Tyto soubory jsou výsledkem příkazů dostupných z Visual Studio Package Manager Console, kterými se řídí akce Entity Frameworku Core aplikované na databázi, jež jsou transformovány do těchto automaticky vygenerovaných migračních souborů. Ty se generují zejména podle aktuálních modelových tříd projektu.

Během projektu byly nejčastěji používány tyto příkazy:

- `Add-Migration`
- `Update-Database`
- `Drop-Database`

Jejich podrobné vysvětlení vychází z této dokumentace [28].

Složku je možné zcela smazat a nechat vygenerovat nové migrační soubory od začátku, což se obvykle dělá, pokud je potřeba začít pracovat s novou databází.

4.2.1.10 Složka Dochazka

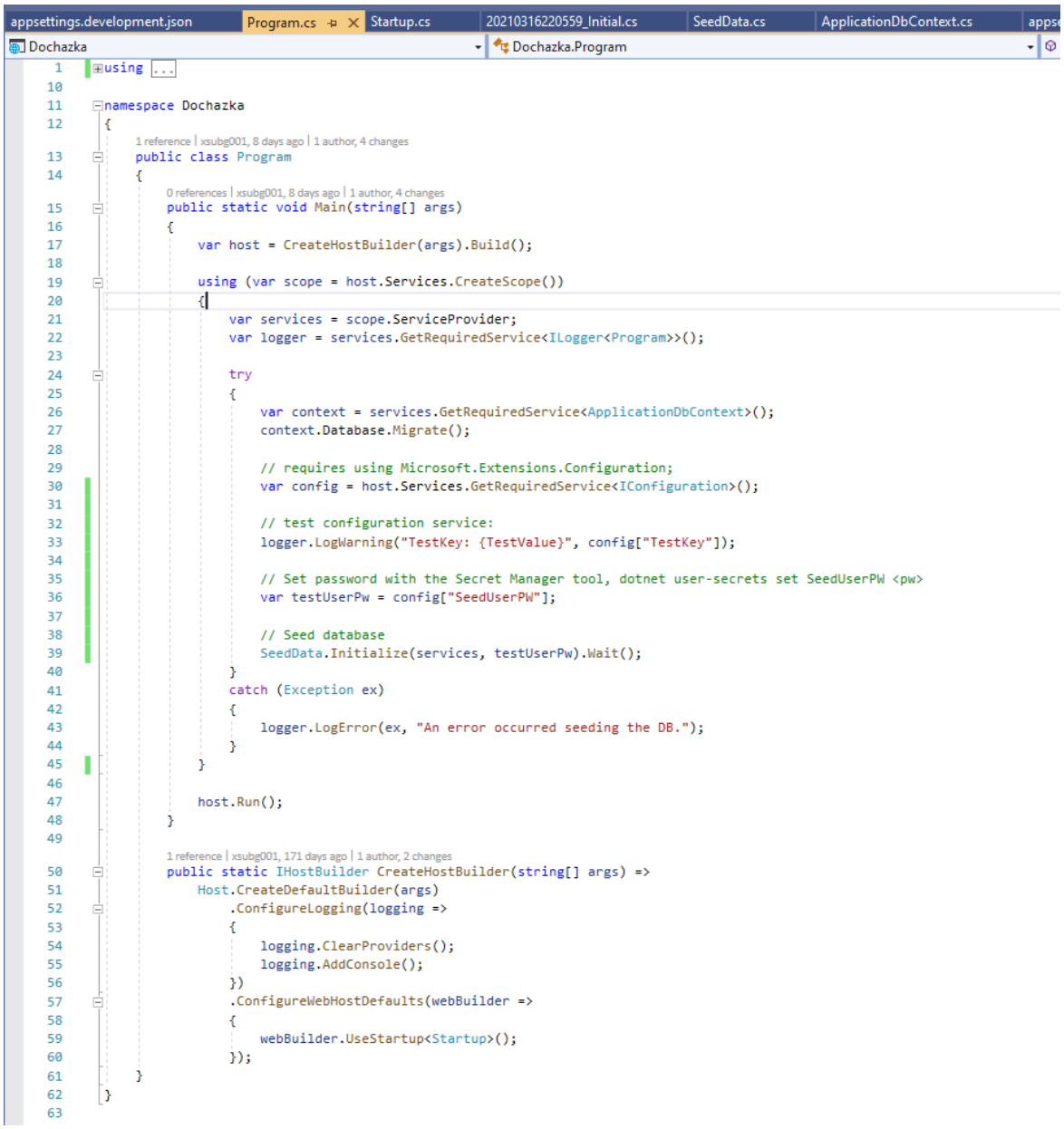
Tato složka obsahuje startovací a konfigurační třídy projektu včetně konfiguračního souboru projektu, jež konfigurují ASP.NET Core runtime a webový server, který bude hostit aplikační kód. Většina kódu je zde automaticky vygenerována v rámci skeletonu projektu, avšak bylo v nich provedeno několik drobných změn pro správnou funkci aplikace.

4.2.1.10.1 Třída Program.cs

Tato třída představuje startovací bod aplikace. Obsahuje metodu `Main`, což je první metoda, kterou se spouští celý program. Zdrojový kód třídy ukazuje Obrázek 56.

Kromě metody `Main` obsahuje také metodu `CreateHostBuilder`, ř. 50-60, jež vrací objekt typu `IHostBuilder`. S pomocí tohoto objektu dochází k inicializaci webového serveru, což se děje v metodě `Main` na ř. 17. Zajímavostí je, že metoda `CreateHostBuilder` konzumuje třídu `Startup.cs`, o níž bude pojednáno níže.

Obrázek 56 - Zdrojový kód třídy Program.cs



Metoda `Main` dále aplikuje nezbytné migrační skripty a zajistí vytvoření databáze, pokud ta ještě neexistuje, viz ř. 27.

Na ř. 30 dochází k načtení aplikační konfigurace, jež je hierarchicky tvořena z několika různých zdrojů, jako jsou např. proměnné prostředí, lokální user-secret a konfigurační soubory `appsettings.json`, `appsettings.development.json`.

Celá problematika aplikační konfigurace v .NET Core je poměrně rozsáhlá a je podrobněji vysvětlena v tomto článku [29]. Pro účely tohoto projektu se z konfigurace načítá jen několik málo hodnot, jako např. `ConnectionString` pro sestavení spojení do databáze,

nebo úvodní uživatelské heslo `SeedUserPW` pro prvního výchozího administrátora aplikace. Toto heslo je uloženo do konfiguračního klíče typu `user-secret` v lokálním secret-store projektu pro lokální aplikaci, případně se čte z proměnné prostředí se jménem `SeedUserPW` pro aplikaci, která je již nasazena do Azure App Service.

Lokální `user-secret` byl vytvořen ve Visual Studio Package Manager Console pomocí následujících příkazů a dokumentace [30]:

```
PM> dotnet user-secrets init
PM> dotnet user-secrets set SeedUserPW "tajneheslo"
Successfully saved SeedUserPW = tajneheslo to the secret store.
PM>
```

Díky tomu je pak možné heslo bezpečně načíst z konfigurace, viz ř. 36, ačkoliv není viditelně součástí žádného konfiguračního souboru `appsettings.*.json`, jak by se mohlo zdát.

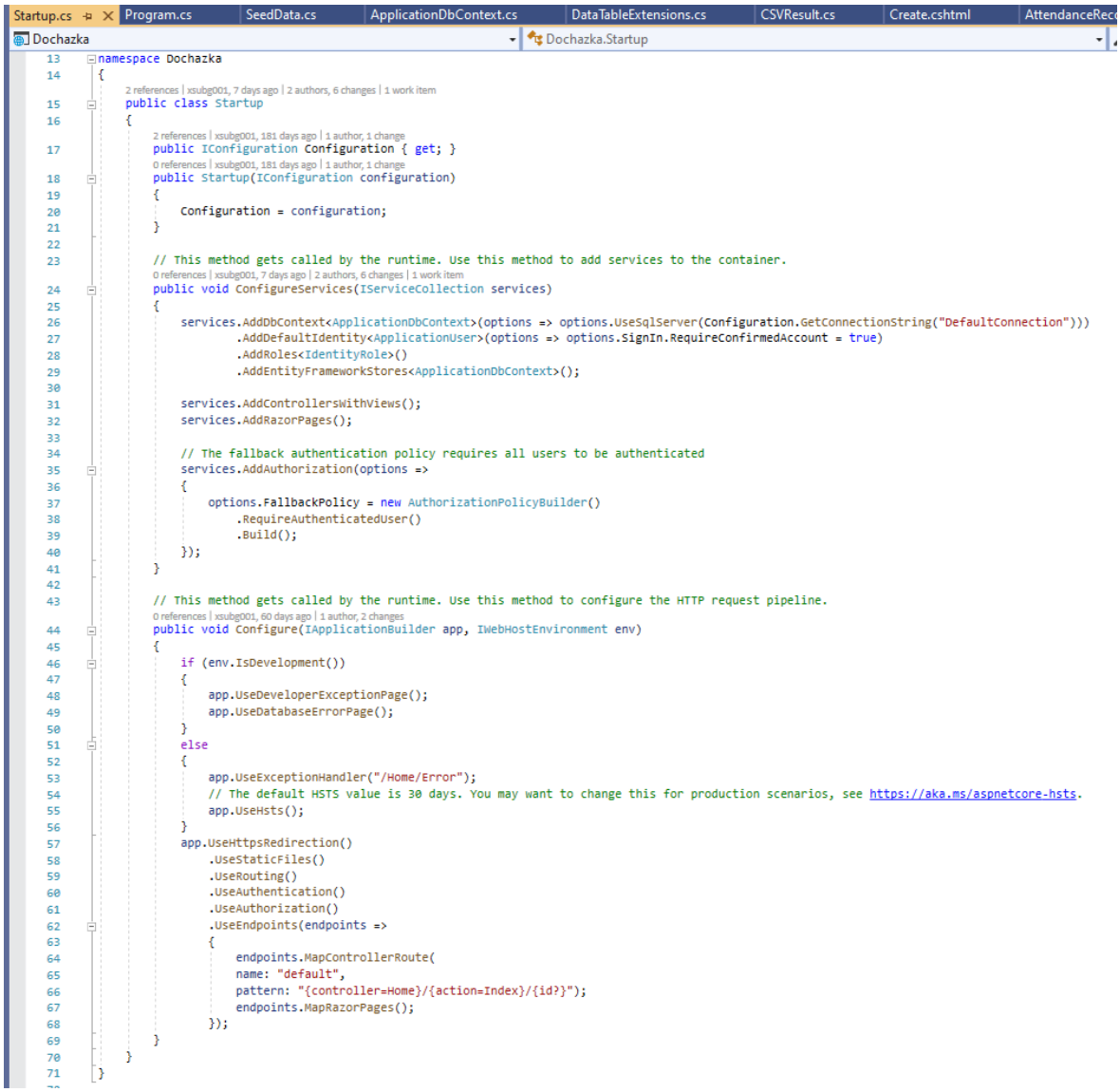
Na ř. 39 pak dochází ke spuštění metody `SeedData.Initialize`, která provádí inicializační kroky, jimiž se zajistní naplnění databáze nezbytnými daty potřebnými pro správný běh aplikace.

Webová aplikace se definitivně spouští jako poslední na ř. 47.

4.2.1.10.2 Třída Startup.cs

Skeleton třídy je automaticky vygenerován a třída je dále upravována tak, jak dochází k modifikaci projektu pomocí průvodce Visual Studia, nebo je případně doplněna manuálně. Zdrojový kód zachycuje Obrázek 57.

Obrázek 57 - Zdrojový kód třídy Startup.cs



Třída v podstatě definuje konfiguraci služeb ASP.NET Runtime a webového serveru pomocí dvou metod, které jsou implicitně volány runtime prostředím:

```
public void ConfigureServices(IServiceCollection services)
```

- Metoda nastavuje Dependency Injection servisní kontejner typu: `Microsoft.Extensions.DependencyInjection.IServiceCollection`, do něhož jsou přidány všechny potřebné služby, resp. objekty, které se v aplikaci používají. Tento servisní kontejner je pak jejich zdrojem prostřednictvím mechanismu Dependency Injection popsaného v 4.2.1.5.1.1 a zajišťuje injekci požadovaných objektů např. do Controlleru. Do kontejneru je tak např. přidán objekt

`ApplicationDbContext`, který zajišťuje komunikaci s databází, nebo objekty knihovny `Microsoft.AspNetCore.Identity`, jež se používají napříč aplikací.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```

- Tato metoda konfiguruje webový server a způsob, jakým je řízena HTTP komunikace. Konfigurují se zde vlastnosti serveru, např. zda je povolena autentizace a autorizace, ř. 60-61, a jaké je schéma Controller endpointu, včetně určení výchozího Controlleru a jeho akce, ř. 62-68.

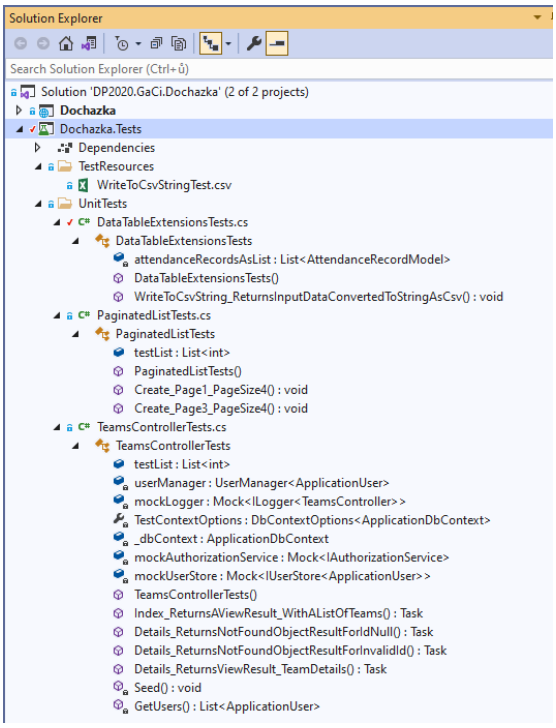
4.2.2 **Projekt Dochazka.Tests.csproj**

V tomto jednoduchém projektu jsou uloženy unit testy, jimiž se testují některé části aplikace. Solution Explorer pohled na tento projekt ukazuje Obrázek 58.

Kromě vlastních testovacích tříd ve složce `UnitTests`, obsahuje projekt soubor `/TestResources/WriteToCsvStringTest.csv`, který obsahuje očekávaný výsledek testované metody v unit testu:

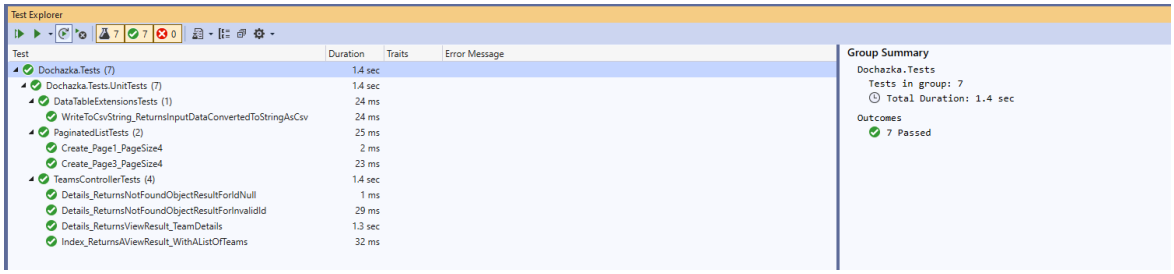
`DataTableExtensionsTests.WriteToCsvString_ReturnsInputDataConvertedToStringAsCsv`, jež testuje konverzi `DataTable` na `string` s CSV strukturou. Tento soubor pak slouží k porovnání aktuálního a očekávaného výsledku testované metody.

Obrázek 58 - Solution Explorer pohled na projekt Dochazka.Tests.csproj



Testy jsou napsané pomocí knihovny Xunit (https://xunit.net) a je možné je spustit pomocí nástroje Visual Studio Test Explorer. Výsledek jednoho z běhů všech testů ukazuje Obrázek 59.

Obrázek 59 - Výsledek běhu Xunit testů z projektu Dochazka.Tests



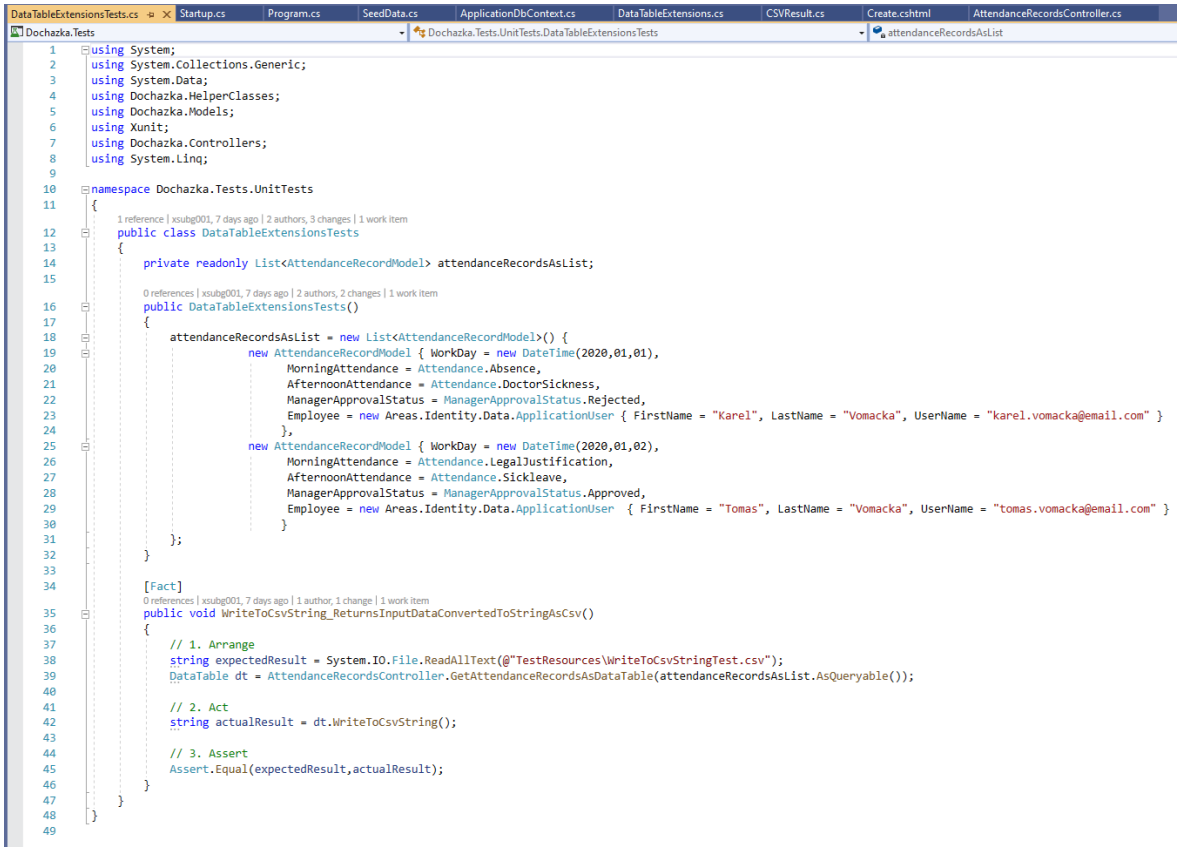
4.2.2.1 Třída DataTableExtensionsTests.cs

Tato třída obsahuje unit test, který testuje novou doplňkovou metodu `WriteToCsvString` třídy `DataTable` definovanou v rozšiřující třídě `DataTableExtensions`, viz 4.2.1.7.2.

Zdrojový kód třídy ukazuje Obrázek 60. V konstruktoru třídy je inicializován lokální člen `private readonly List<AttendanceRecordModel> attendanceRecordsAsList`, který

obsahuje vstupní testovací data, což je seznam mající dva prvky třídy `AttendanceRecordModel`.

Obrázek 60 - Zdrojový kód třídy DataTableExtensionsTests.cs



```
DataTableExtensionsTests.cs | Startup.cs | Program.cs | SeedData.cs | ApplicationDbContext.cs | DataTableExtensions.cs | CSVResult.cs | Create.cshtml | AttendanceRecordsController.cs
Dochazka.Tests | Dochazka.Tests.UnitTests.DataTableExtensionsTests | attendanceRecordsAsList

1  using System;
2  using System.Collections.Generic;
3  using System.Data;
4  using Dochazka.HelperClasses;
5  using Dochazka.Models;
6  using Xunit;
7  using Dochazka.Controllers;
8  using System.Linq;
9
10 namespace Dochazka.Tests.UnitTests
11 {
       1 reference | xsubg001, 7 days ago | 2 authors, 3 changes | 1 work item
12     public class DataTableExtensionsTests
13     {
14         private readonly List<AttendanceRecordModel> attendanceRecordsAsList;
15
           0 references | xsubg001, 7 days ago | 2 authors, 2 changes | 1 work item
16         public DataTableExtensionsTests()
17         {
18             attendanceRecordsAsList = new List<AttendanceRecordModel>() {
19                         new AttendanceRecordModel { WorkDay = new DateTime(2020,01,01),
20                             MorningAttendance = Attendance.Absence,
21                             AfternoonAttendance = Attendance.DoctorSickness,
22                             ManagerApprovalStatus = ManagerApprovalStatus.Rejected,
23                             Employee = new Areas.Identity.Data.ApplicationUser { FirstName = "Karel", LastName = "Vomacka", UserName = "karel.vomacka@email.com" }
24                             },
25                         new AttendanceRecordModel { WorkDay = new DateTime(2020,01,02),
26                             MorningAttendance = Attendance.LegalJustification,
27                             AfternoonAttendance = Attendance.Sickleave,
28                             ManagerApprovalStatus = ManagerApprovalStatus.Approved,
29                             Employee = new Areas.Identity.Data.ApplicationUser  { FirstName = "Tomas", LastName = "Vomacka", UserName = "tomas.vomacka@email.com" }
30                             }
31             };
32         }
33
34         [Fact]
           0 references | xsubg001, 7 days ago | 1 author, 1 change | 1 work item
35         public void WriteToCsvString_ReturnsInputDataConvertedToStringAsCsv()
36         {
37             // 1. Arrange
38             string expectedResult = System.IO.File.ReadAllText(@"TestResources\WriteToCsvStringTest.csv");
39             DataTable dt = AttendanceRecordsController.GetAttendanceRecordsAsDataTable(attendanceRecordsAsList.AsQueryable());
40
41             // 2. Act
42             string actualResult = dt.WriteToCsvString();
43
44             // 3. Assert
45             Assert.Equal(expectedResult,actualResult);
46         }
47     }
48 }
49
```

Testovací metoda:

`public void WriteToCsvString_ReturnsInputDataConvertedToStringAsCsv()` ,

je dekorována `Xunit` atributem `[Fact]` a má tři logické části:

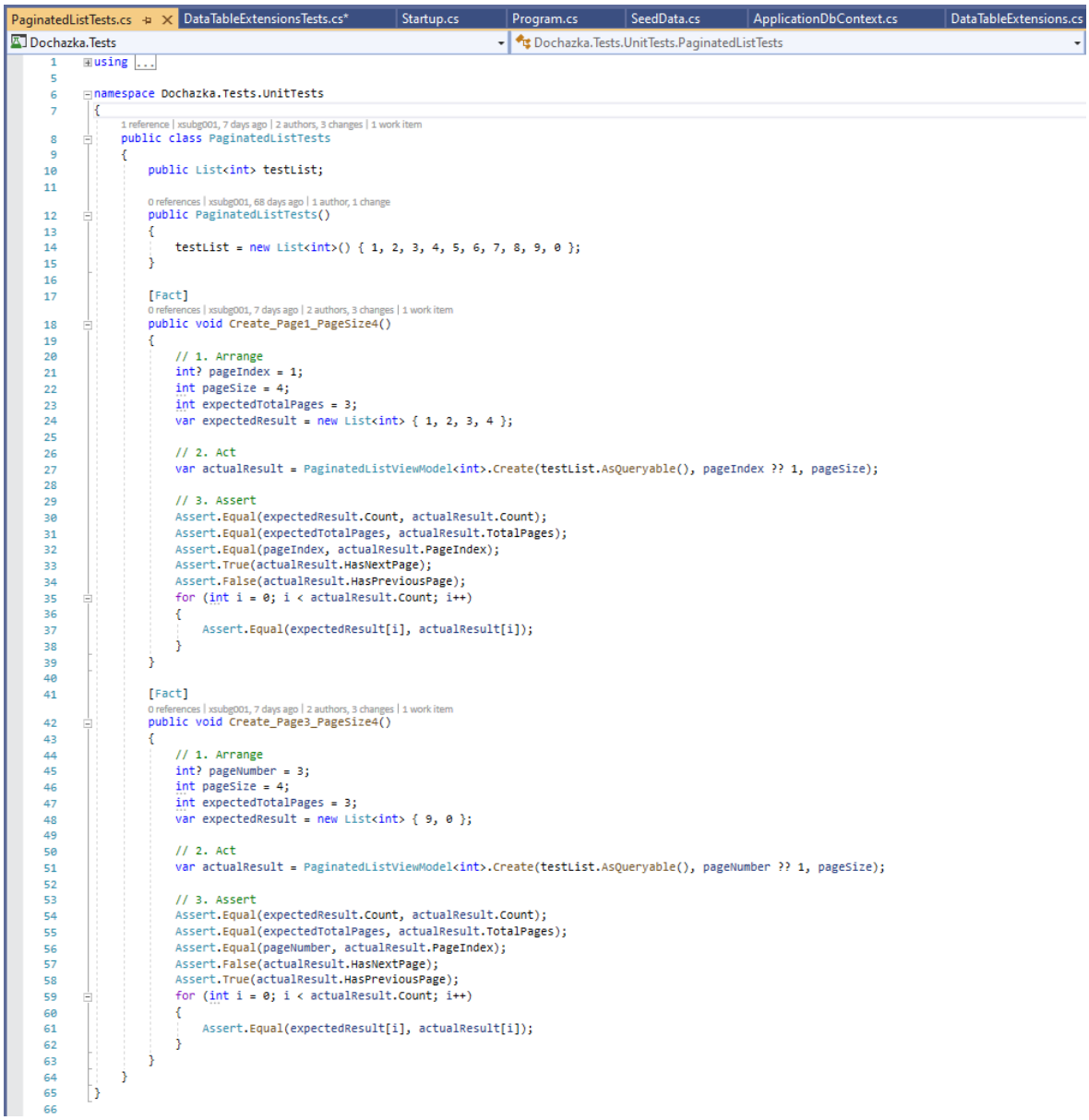
- V první části, označené jako Arrange, jsou provedeny kroky pro přípravu testu. Do lokální proměnné `expectedResult` se načte obsah souboru `/TestResources/WriteToCsvStringTest.csv`, který obsahuje očekávaný výsledek testované metody (ř. 38). Dále pak dojde ke konverzi vstupního seznamu s testovacími daty na objekt třídy `DataTable`, ř. 39.
- V druhé části, označené jako Act, je zavolána testovaná metoda `DataTableExtensions.WriteToCsvString`, jež konvertuje objekt třídy `DataTable` na `string` s CSV strukturou a výsledek se uloží do lokální proměnné `actualResult`.

- V třetí části, označené jako Assert, jsou pomocí metody `Xunit.Assert.Equal` porovnány `expectedResult` a `actualResult`. Výsledek tohoto porovnání určí výsledek celého testu.

4.2.2.2 **Třída PaginatedListTests.cs**

Tato testovací třída testuje metodu `Create` ve ViewModelu třídy `PaginatedListViewModel.cs`. Třída obsahuje dvě metody, které testují funkci metody `Create` s různými vstupy. Zdrojový kód zachycuje Obrázek 61.

Obrázek 61 -Zdrojový kód třídy PaginatedListTests.csv



```
using ...

namespace Dochazka.Tests.UnitTests
{
    1 reference | xsubg001, 7 days ago | 2 authors, 3 changes | 1 work item
    public class PaginatedListTests
    {
        public List<int> testList;

        0 references | xsubg001, 68 days ago | 1 author, 1 change
        public PaginatedListTests()
        {
            testList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
        }

        [Fact]
        0 references | xsubg001, 7 days ago | 2 authors, 3 changes | 1 work item
        public void Create_Page1_PageSize4()
        {
            // 1. Arrange
            int? pageIndex = 1;
            int pageSize = 4;
            int expectedTotalPages = 3;
            var expectedResult = new List<int> { 1, 2, 3, 4 };

            // 2. Act
            var actualResult = PaginatedListViewModel<int>.Create(testList.AsQueryable(), pageIndex ?? 1, pageSize);

            // 3. Assert
            Assert.Equal(expectedResult.Count, actualResult.Count);
            Assert.Equal(expectedTotalPages, actualResult.TotalPages);
            Assert.Equal(pageIndex, actualResult.PageIndex);
            Assert.True(actualResult.HasNextPage);
            Assert.False(actualResult.HasPreviousPage);
            for (int i = 0; i < actualResult.Count; i++)
            {
                Assert.Equal(expectedResult[i], actualResult[i]);
            }
        }

        [Fact]
        0 references | xsubg001, 7 days ago | 2 authors, 3 changes | 1 work item
        public void Create_Page3_PageSize4()
        {
            // 1. Arrange
            int? pageNumber = 3;
            int pageSize = 4;
            int expectedTotalPages = 3;
            var expectedResult = new List<int> { 9, 0 };

            // 2. Act
            var actualResult = PaginatedListViewModel<int>.Create(testList.AsQueryable(), pageNumber ?? 1, pageSize);

            // 3. Assert
            Assert.Equal(expectedResult.Count, actualResult.Count);
            Assert.Equal(expectedTotalPages, actualResult.TotalPages);
            Assert.Equal(pageNumber, actualResult.PageIndex);
            Assert.False(actualResult.HasNextPage);
            Assert.True(actualResult.HasPreviousPage);
            for (int i = 0; i < actualResult.Count; i++)
            {
                Assert.Equal(expectedResult[i], actualResult[i]);
            }
        }
    }
}
```

V konstruktoru třídy je inicializována lokální proměnná obsahující testovací data, což je seznam čísel:

```
testList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
```

Dále jsou pak vytvořeny dvě testovací metody dekorované `Xunit [Fact]` atributem:

```
public void Create_Page1_PageSize4()
public void Create_Page3_PageSize4()
```

Jelikož jsou analogické, blíže pouze k první z nich:

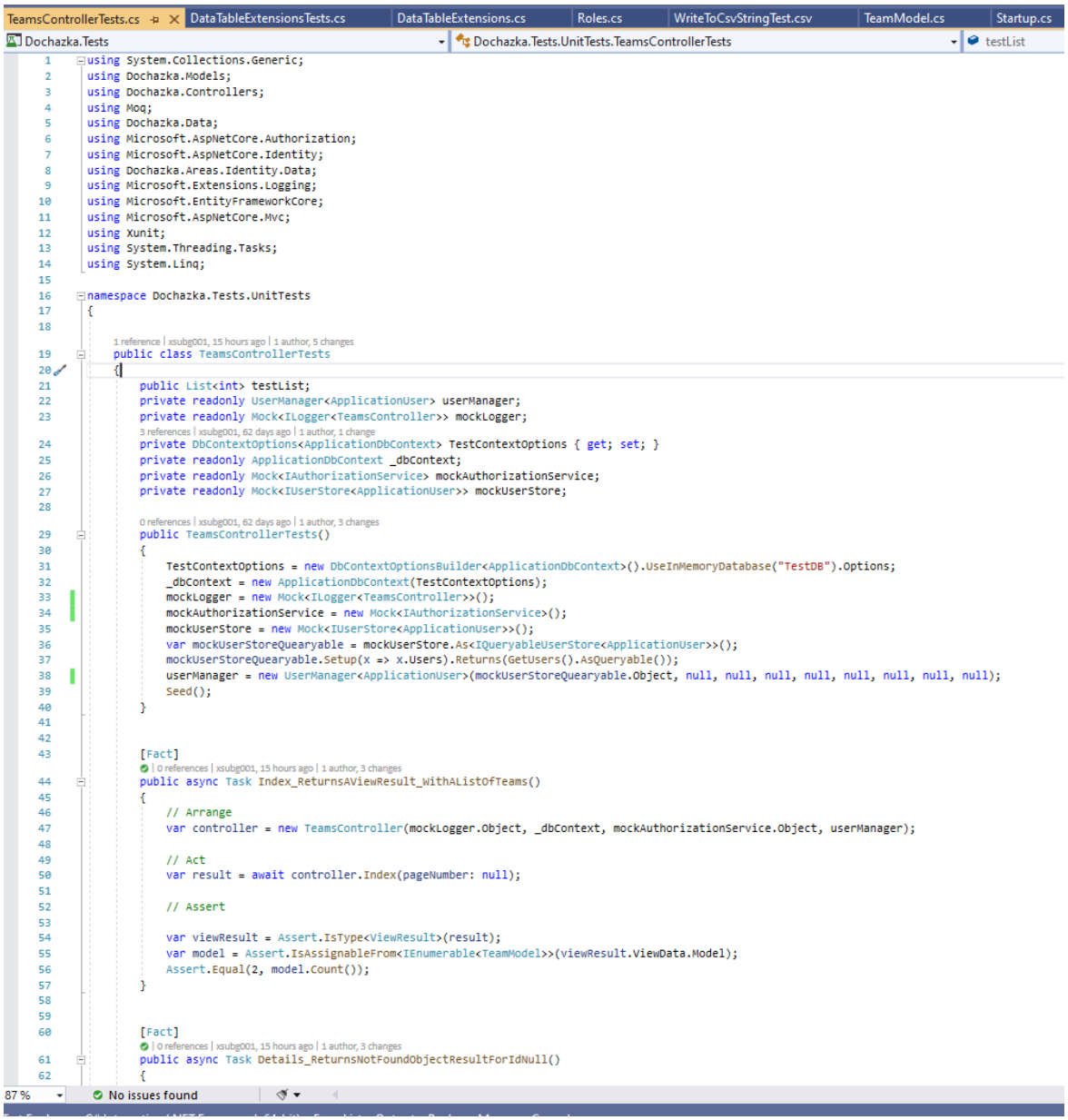
- V sekci Arrange jsou nastaveny lokální proměnné, které jednak připraví hodnoty vstupních parametrů testované metody `PaginatedListViewModel<T>.Create` (ř. 21-23). Také se zde nastaví lokální proměnné `expectedTotalPages` a `expectedResult`, které obsahují očekávané výsledky testované metody (ř. 24).
- V sekci Act pak dojde k zavolání testované metody `PaginatedListViewModel<int>.Create` (ř. 27), jež vrací objekt třídy `PaginatedListViewModel<int>`, který je uložen do lokální proměnné `actualResult.`
- V sekci Assert pak jsou pomocí metod třídy `Xunit.Assert` porovnány aktuální a očekávané výsledky. Ověřuje se počet prvků ve vráceném seznamu (`actualResult.Count`) na ř. 30, počet stránek, které lze z originálního seznamu vytvořit (`actualResult.TotalPage`) na ř. 31, vrácené číslo stránky (`actualResult.PageIndex`) na ř. 32, správná hodnota properties `HasNextPage` a `HasPreviousPage` na ř. 33-34, a nakonec také jednotlivé prvky ve vráceném seznamu na ř. 35-38.

4.2.2.3 **Třída TeamsControllerTests.cs**

Tato třída testuje třídu `TeamsController.cs` a v ní dvě akční metody `Index` a `Details`. Třída využívá také knihovnu `Moq` (https://github.com/Moq/moq4/wiki/Quickstart) pro simulaci některých objektů na nichž testovaný kód Controlleru závisí, aby bylo možné zavolat konstruktor a vytvořit instanci třídy `TeamsController`, aniž bychom měli k dispozici reálné prostředí webového serveru a databázi. Úvodní část třídy s definicí konstruktoru zachycuje Obrázek 62.

Tento kód byl inspirována průvodcem pro testování ASP.NET Controlleru [31].

Obrázek 62 - Zdrojový kód TeamsControllerTests.cs, úvodní část s konstruktorem



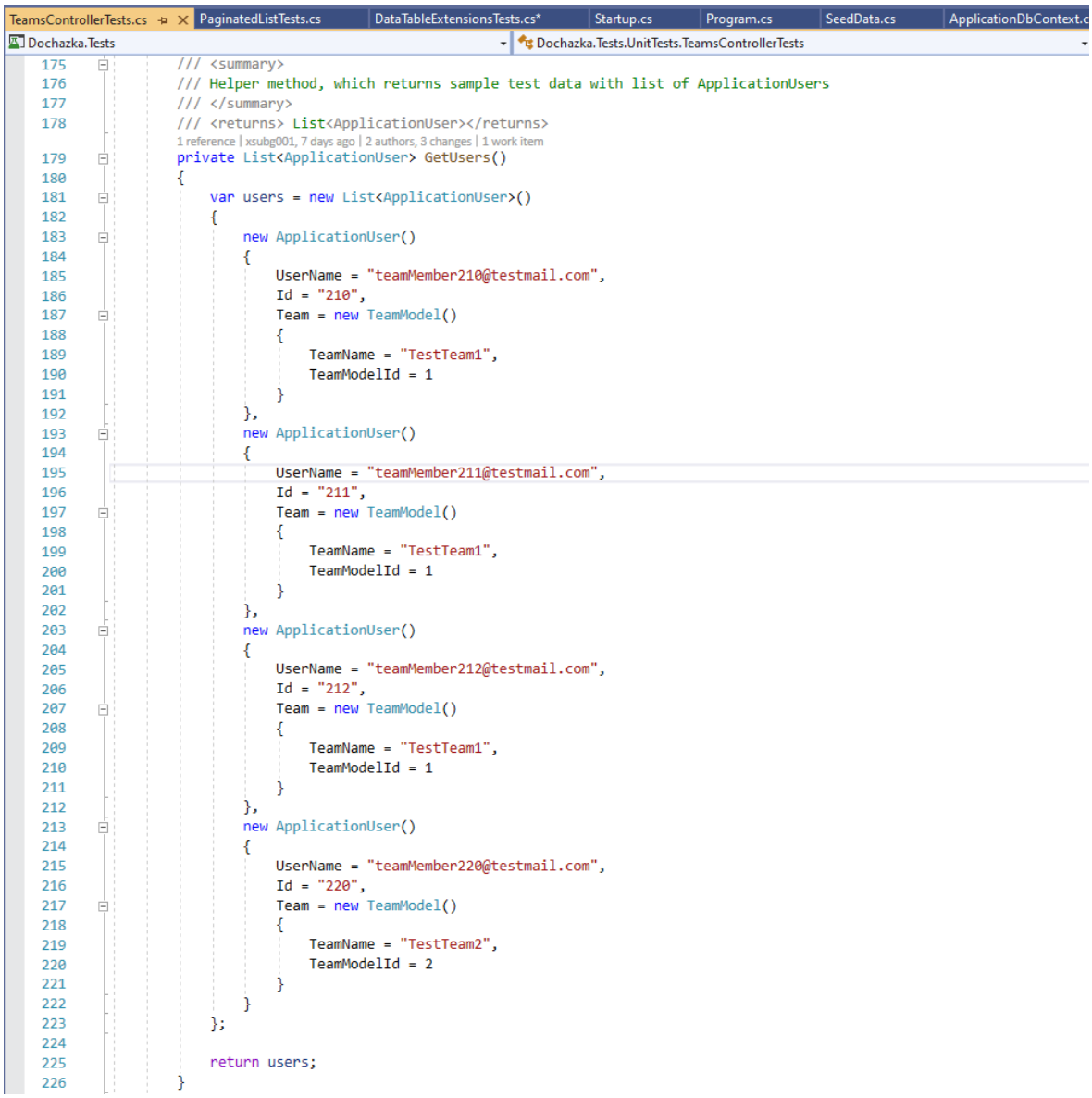
Obrázek 63 - Zdrojový kód TeamsControllerTests.cs, metoda Seed()

TeamsControllerTests.cs | PaginatedListTests.cs | DataTableExtensions.cs | DataTableExtensionsTests.cs | WriteToCsvStringTest.csv

Dochazka.Tests | Dochazka.Tests.UnitTests.TeamsControllerTests

```
109         /// <summary>
110         /// Helper method which seeds the InMemoryDatabase with sample test data, i.e. it creates test entries of TeamModel
111         /// </summary>
            1 reference | xsubg001, 2 days ago | 1 author, 2 changes
112         private void Seed()
113         {
114             using (var context = new ApplicationDbContext(TestContextOptions))
115             {
116                 context.Database.EnsureDeleted();
117                 context.Database.EnsureCreated();
118
119                 var team1 = new TeamModel()
120                 {
121                     TeamName = "TestTeam1",
122                     TeamModelId = 1,
123                     PrimaryManager = new ApplicationUser()
124                     {
125                         UserName = "testmgr021@testmail.com",
126                         Id = "021"
127                     },
128                     PrimaryManagerId = "021",
129                     TeamMembers = new List<ApplicationUser>
130                     {
131                         new ApplicationUser()
132                         {
133                             UserName = "teamMember210@testmail.com",
134                             Id = "210"
135                         },
136                         new ApplicationUser()
137                         {
138                             UserName = "teamMember211@testmail.com",
139                             Id = "211"
140                         }
141                     }
142
143                 };
144
145                 var team2 = new TeamModel()
146                 {
147                     TeamName = "TestTeam2",
148                     TeamModelId = 2,
149                     PrimaryManager = new ApplicationUser()
150                     {
151                         UserName = "testmgr022@testmail.com",
152                         Id = "022"
153                     },
154                     PrimaryManagerId = "022",
155                     TeamMembers = new List<ApplicationUser>
156                     {
157                         new ApplicationUser()
158                         {
159                             UserName = "teamMember220@testmail.com",
160                             Id = "220"
161                         },
162                         new ApplicationUser()
163                         {
164                             UserName = "teamMember221@testmail.com",
165                             Id = "221"
166                         }
167                     }
168                 };
169
170                 context.AddRange(team1, team2);
171                 context.SaveChanges();
172             }
173         }
```

Obrázek 64 - Zdrojový kód TeamsControllerTests.cs, metoda GetUsers()



Nyní blíže k jednotlivým metodám třídy a konstruktoru.

`private List<ApplicationUser> GetUsers()`

- Jde o pomocnou metodu, která připravuje testovací data, tj. vrací seznam uživatelů, tedy objektů třídy `ApplicationUser`.

`private void Seed()`

- Jde o pomocnou metodu, která nejprve smaže a znovu vytvoří testovací databázi (ř. 116-117). Poté vytvoří dva testovací objekty třídy `TeamModel` (ř. 119-168). Ty jsou následně přidány do aktuálního aplikačního databázového contextu (ř. 170)

používaného v rámci této třídy a pak zapsány do paměťové (InMemory) databáze (ř. 171), jež se používá během unit testu namísto skutečné SQL databáze.

`public TeamsControllerTests()`

- Konstruktor třídy, který inicializuje potřebné členy a volá také metodu `Seed()`.
    - Pro běh testu se v této třídě využívá tzv. paměťová (InMemory) databáze, což je v podstatě jakýsi simulátor SQL databáze, běžící pouze v paměti v průběhu testů. Jakmile testy doběhnou, jsou veškerá data v této databázi ztracena. Databáze je dostupná prostřednictvím členu `_dbContext`, který je inicializován na ř. 31-32.
    - Dále jsou pomocí knihovny `Moq` vytvořeny simulované objekty (stub) pro členy `mockLogger`, `mockAuthorizationService`, `mockUserStore` a `mockUserStoreQuearyable`, kde pro objekt `mockUserStoreQuearyable` je také simulována (mockována) hodnota property `Users`, pomocí metody `GetUsers`() diskutované výše (ř. 36).
    - Na ř. 37 je vytvořen objekt `userManager` typu `UserManager<ApplicationUser>` pomocí konstruktoru třídy.
    - Nakonec ř. 39 volá metodu `Seed()`, která vloží do paměťově databáze testovací data.

`public async Task Index_ReturnsAViewResult_WithAListOfTeams()`

- Jde o testovací metodu dekorovanou `Xunit` atributem `[Fact]`.
- V sekci Arrange je inicializován objekt `TeamsController` buď pomocí simulovaných objektů (stub), nebo reálných objektů inicializovaných v konstruktoru této třídy.
- V sekci Act se volá testovaná akční metoda Controlleru `Index` a návratová hodnota je uložena do lokální proměnné `result`.
- V sekci Assert dochází k porovnání aktuálních a očekávaných výsledků:
    - Ř.54 ověřuje, že lokální proměnná `result` je typu `ViewResult`, což je očekávaný výsledek. Pokud ano, přiřadí se tento objekt do lokální proměnné `viewResult`.

- Ř.55 ověří, že model v objektu `viewResult.ViewData.Model` je typu `IEnumerable<TeamModel>`, což je opět očekávaný typ výsledku, a pokud ano, přiřadí ho do lokální proměnné `model`.
- Ř.56 ověří, že počet objektů v seznamu `model` je roven dvěma, což je počet je očekávaný počet objektů třídy `TeamsModel`, jelikož právě tolik entit vytváří metoda `Seed()` v databázi prostřednictvím databázového contextu `_dBcontext`.

```
public async Task Details_ReturnsNotFoundObjectResultForIdNull()
```

- Testovací metoda, která testuje akční metodu Controlleru `Details`. Konkrétně se v testu ověřuje případ, kdy vstupní parameter `id` metody `Details` obsahuje hodnotu `null`, což vede k vrácení HTTP odpovědi `NotFoundResult(404)`.

```
public async Task Details_ReturnsNotFoundObjectResultForInvalidId()
```

- Podobná testovací metoda jako výše. Testuje případ, kdy `id` sice obsahuje hodnotu, ale ta neukazuje na žádnou validní entitu, proto se opět v takovém případě vrátí HTTP odpověď `NotFoundResult (404)`.

```
public async Task Details_ReturnsViewResult_TeamDetails()
```

- Podobná metoda jako výše, která testuje pozitivní scénář, kdy `id` je platné a odkazuje na existující entitu `TeamModel`. Zdrojový kód metody ukazuje Obrázek 65.
- V Assert části se vyhodnotí, že metoda Controlleru `Details` vrátí výsledek očekáváného typu `ViewResult` (ř. 99), dále, že objekt modelu uvnitř výsledku `ViewResult` je očekávaného typu `TeamModel` (ř. 100). Na závěr se ověří vlastní data v modelu (ř. 101-105). Ta musí přesně odpovídat testovacím datům založeným metodami `GetUsers()` a `Seed()`.

Obrázek 65 - Zdrojový kód testovací metody Details_Returns ViewResult_TeamDetails()

```
89      [Fact]
        0 references | xsubg001, 2 days ago | 1 author, 2 changes
90      public async Task Details_ReturnsViewResult_TeamDetails()
91      {
92          // Arrange
93          var controller = new TeamsController(mockLogger.Object, _dbContext, mockAuthorizationService.Object, userManager);
94
95          // Act
96          var result = await controller.Details(id: 1);
97
98          // Assert
99          var viewResult = Assert.IsType<ViewResult>(result);
100         var model = Assert.IsAssignableFrom<TeamModel>(viewResult.ViewData.Model);
101         Assert.Equal("TestTeam1", model.TeamName);
102         Assert.Equal("testmgr021@testmail.com", model.PrimaryManager.UserName);
103         Assert.True(viewResult.ViewData.ContainsKey("teamMembers"));
104         var teamMembers = Assert.IsType<List<ApplicationUser>>(viewResult.ViewData["teamMembers"]);
105         Assert.Equal(3, teamMembers.Count);
106     }
107
```

4.3 Nasazení aplikace do cloudu

V této části bude vysvětlen postup, jakým byla aplikace nasazena do veřejného cloudu Microsoft Azure pomocí služeb Azure App Service, viz odstavec 3.10 a Azure SQL Database, viz odstavec 3.11.

K nasazení byl využit příkazový interpret PowerShell a PowerShell Az.Rm modul, jež jsou diskutovány v 3.6 a 3.6.1. PowerShell skript byl vytvořen pomocí Visual Studio Code, viz 3.12.

Skript diskutovaný v této části se nachází v projektu `Dochazka.csproj` ve složce `/scripts/AzurePublishCommands.ps1`.

4.3.1 Vytvoření služby Azure SQL Server Database

Tato část realizuje:

- User Story 22: Vytvoření služby Azure SQL Server Database,

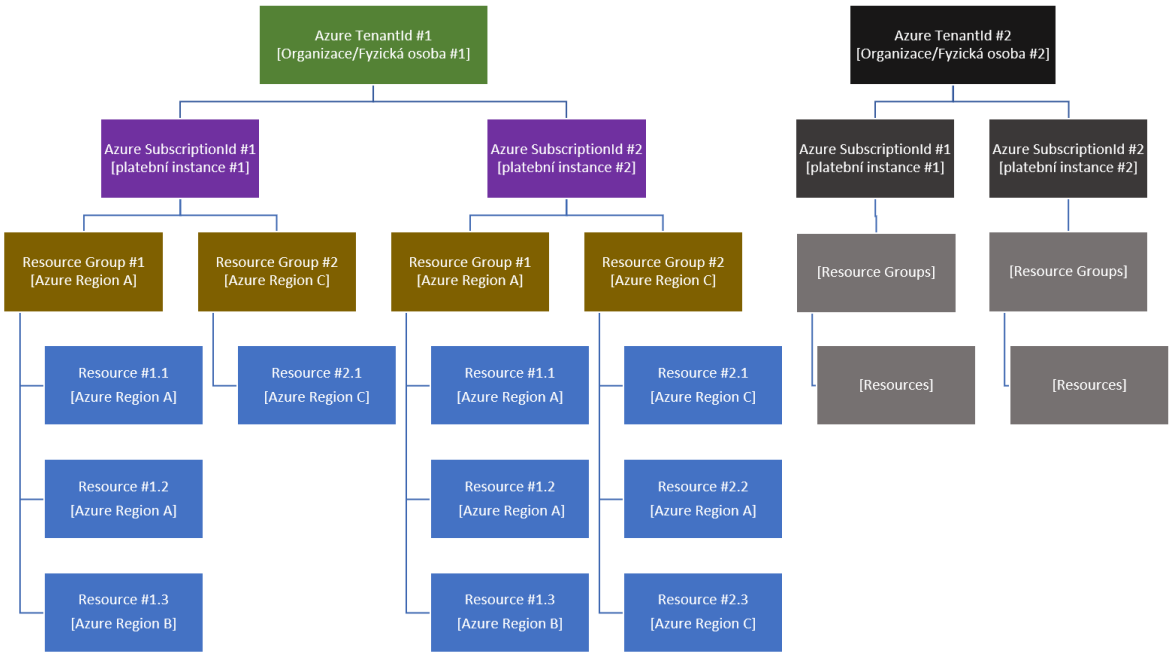
viz odstavec 4.1.2.

Použitý PowerShell kód ukazuje Obrázek 66, jenž je inspirován průvodcem [32].

Obrázek 66 - PowerShell skript, část vytvoření instance služby Azure SQL Database

- V úvodní části, (ř. 1-3), jsou inicializovány lokální proměnné, které specifikují, jméno Azure regionu, kde budeme vytvářet Azure zdroje a jméno Azure Resource Group (skupina zdrojů). Azure Resource Group si lze představit podobně, jako složku v adresáři, která obsahuje Azure resources (zdroje), což jsou pak konkrétní instance jednotlivých Azure služeb. Tyto dvě proměnné jsou společné pro oba zdroje, tedy instance služby Azure SQL Database a Azure App Service, jelikož obě budou existovat ve stejné Resource Group a ve stejném Azure regionu.
- Ř.4 vytvoří připojení ke službě Azure pomocí konkrétní `SubscriptionId` a `TenantId`. Subscription (předplatné) si je možné vytvořit zdarma, např. pomocí [33]. `TenantId` si lze představit jako identifikátor organizace nebo fyzické osoby. V rámci jednoho `TenantId` pak existuje jedno nebo více Subscriptions s různými `SubscriptionId`. Subscription je pak jedna platební instance Azure služby, k níž se vztahují platby za spotřebované Azure zdroje a služby technické podpory. Azure resources, jež jsou vytvořeny v rámci dané Subscription jsou organizovány do alespoň jedné nebo více Azure Resource Group. V rámci jedné Azure Subscription, a dokonce v rámci jedné Azure Resource Group, mohou existovat Azure resources vytvořené v různých Azure Regionech. Tuto hierarchii znázorňuje Obrázek 67 a vychází z [34] a [35].

Obrázek 67 - Subscription model služby Azure



- Ř. 7-9 inicializují další lokální proměnné:
  - Ř.7 přiřadí do lokální proměnné `$serverName` jméno instance služby Azure SQL database. Je třeba, aby to bylo jméno globálně unikátní v rámci DNS systému spravujícího Azure doménu `*.database.windows.net`, jelikož toto jméno bude součástí hostname SQL serveru a tedy i součástí `$connectionString` (řetězec připojení).
  - Ř. 8 přiřadí do lokální proměnné `$dbName` jméno SQL databáze, která bude vytvořena na SQL serveru.
  - Ř. 9 přiřadí do lokální proměnné `$myIPaddress` konkrétní IP adresu, která se bude moci k SQL serveru připojit nad rámec rozsahu IP adres přiřazených Azure datovým centrům.
- Ř. 10 vyzve uživatele, aby zadal admin uživatelské jméno pro SQL server.
- Ř. 11 vyzve uživatele, aby zadal heslo pro admin uživatele SQL serveru.
- Ř. 12 provede konverzi `stringu` obsahující admin heslo na typ `SecureString`, což je zabezpečený typ pro uložení hesla a používá se v dalších příkazech níže.
- Ř. 13 vytvoří objekt třídy `PSCredential` pomocí dříve připravených proměnných obsahujících admin uživatelské jméno a zabezpečené heslo.
- Ř. 15 vytvoří Resource Group mající jméno dle proměnné `$rgName` a v Azure regionu daného proměnnou `$location`. Tato Resource Group i instance služeb

vytvářených níže budou vytvořeny v Azure Subscription, kam jsme se připojili příkazem na ř. 4 a jehož vlastnosti jsou uloženy v lokálním contextu aktivní PowerShell seance.

- Ř. 16 vytvoří instanci služby Azure SQL serveru s parametry podle hodnot argumentů proměnných.
- Ř. 17 vytvoří firewall pravidlo pro instanci služby Azure SQL server vytvořenou v předchozím kroku, která povoluje přístup k SQL serveru pouze z veřejných IP adres datových center Azure.
- Ř. 18 vytvoří firewall pravidlo pro instanci služby Azure SQL server vytvořenou v předchozím kroku, která také povoluje přístup k SQL serveru pouze z konkrétní IP adresy (adresa mého domácího internetového připojení).
- Ř. 19 vytvoří SQL databázi v rámci serveru, která má tarif `Basic`. Ta určuje výkonové parametry databáze, její funkce, vlastnosti a také cenu služby [36]. Ta je podle aktuálního vytížení databáze přibližně 4 € za měsíc a garantuje SLA (Service Level Agreement) 99.99 % [37].
- Ř. 21, nastaví lokální proměnnou `$connectionString`, která obsahuje nezbytné informace pro připojení k SQL Serveru ve standardním formátu a používá se pro konfiguraci aplikace.
- Ř. 22 pro kontrolu vypíše hodnotu proměnné `$connectionString` do terminálu.

4.3.2 Vytvoření služby Azure App Service a její integrace se službou GitHub

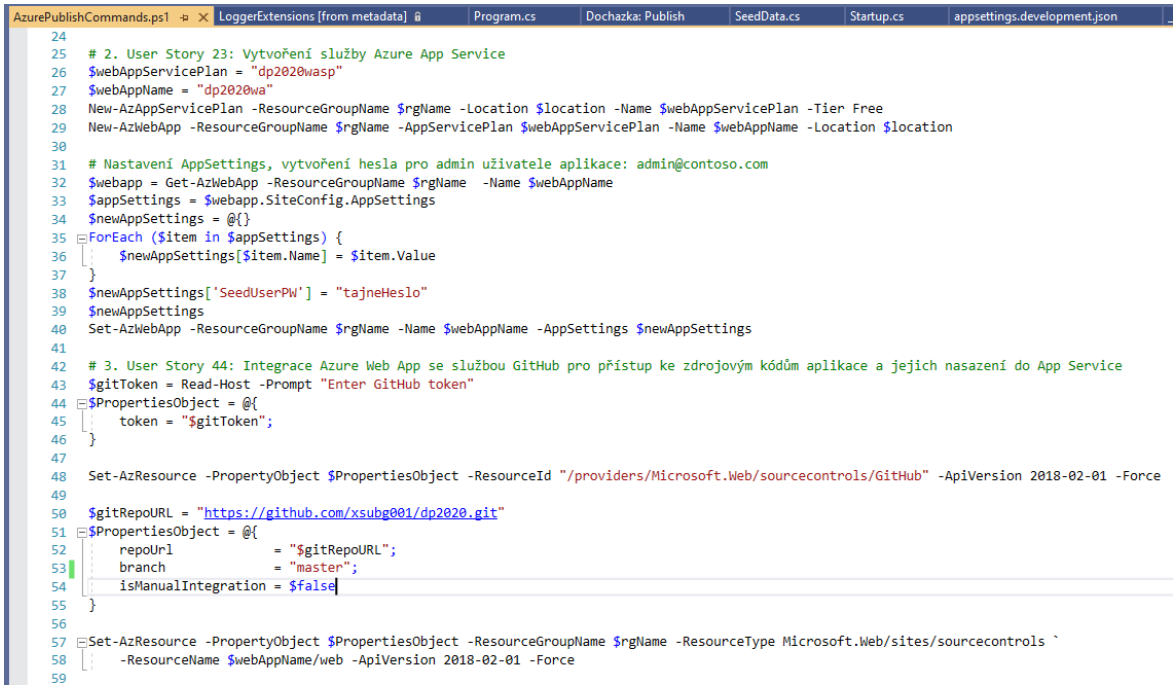
Tato část realizuje:

- User Story 23: Vytvoření služby Azure App Service
- User Story 44: Integrace Azure Web App se službou GitHub pro přístup ke zdrojovým kódům aplikace a jejich nasazení do App Service,

viz odstavec 4.1.2.

Použitý PowerShell kód ukazuje Obrázek 68 a je inspirován průvodcem [38].

Obrázek 68 - PowerShell skript, část vytvoření instance služby Azure App Service a její integrace se službou Git



- Ř.26 přiřadí do lokální proměnné `$webAppServicePlan` jméno instance služby Azure App Service Plan, což je kontejner pro jednu nebo více instancí služeb Azure App Service.
- Ř.27 přiřadí do lokální proměnné `$webAppName` jméno instance služby Azure App Service a je třeba, aby to bylo jméno globálně unikátní v rámci DNS systému spravujícího Azure doménu `*.azurewebsites.net`, jelikož toto jméno bude součástí hostname této instance Azure App Service a tedy součástí URL, na kterém poběží naše aplikace.
- Ř. 28 podle vstupních argumentů daných lokálními proměnnými vytvoří instanci služby Azure App Service Plan, tedy zmíněný kontejner, jenž bude hostovat jednotlivé instance služby Azure App Service. Tarifní plán služby je `Free`, tedy bezplatný, a určuje výkonové parametry i dostupné funkce a kvalitativní vlastnosti služby. V této tarifní kategorii zdarma není garantováno žádné SLA, které však s vyšším tarifem může být až 99.95 % [39].
- Ř. 29 vytvoří instanci služby Azure App Service podle hodnot vstupních argumentů.
- Ř.32 načte objekt Azure App Service do lokální proměnné.

- Ř.33 přiřadí objekt `$webapp.SiteConfig.AppSettings` do lokální proměnné `$appSettings`, se kterou se bude dál pracovat. Objekt je typu `HashTable`.
- Ř.34 inicializuje novou lokální proměnnou `$newAppSettings` typu `HashTable`.
- Ř. 35-37 tento blok kódu překopíruje data z původního `$appSettings` do `$newAppSettings`.
- Ř. 38, vytvoří v objektu `$newAppSettings` nový klíč `SeedUserPW` s hodnotou tajného admin hesla pro výchozího admin uživatele aplikace, viz odstavec 4.2.1.10.1.
- Ř. 39 pro kontrolu vypíše hodnotu aktualizovaného `$newAppSettings` na konzoli.
- Ř. 40 aplikuje `$newAppSettings`, což je v podstatě původní `$appSettings` rozšířené o nový klíč s heslem, na dříve vytvořenou instanci Azure App Service, kde je z tohoto klíče následně vytvořena příslušná proměnná prostředí (environment variable) na lokálním hostovi. Jméno nové proměnné prostředí bude `SeedUserPW`, čímž se heslo stane viditelné pro aplikační konfiguraci, viz [29].

Následuje část integrace Azure App Service se službou GitHub, která vychází také z průvodce [38]:

- Ř. 43 načte od uživatele GitHub bezpečnostní token, což je v podstatě časově omezené heslo, jemuž uživatel při jeho vytvoření nastaví jistá oprávněními ke svému GitHub profilu, a tudíž opravňuje držitele tokenu přistupovat k datům v rámci jeho uživatelského profilu podle přidělených oprávnění. Token byl vygenerován ve službě GitHub [40].
- Ř. 44-46 vytvoří objekt typu `HashTable`, který v sobě drží hodnotu tokenu klíčovanou jako `token`. Tento objekt je konzumován příkazy níže.
- Ř. 48 vytvoří Azure zdroj, který je potřebný pro integraci Azure Web App a GitHub služeb a v podstatě obsahuje jen objekt s GitHub tokenem. Tento zdroj je závislý na dříve vytvořené instanci Azure App Service, což je jeho rodič, a v podstatě složí jako forma konfigurace služby.
- Ř. 50-55 vytvoří objekt typu `HashTable`, jenž prakticky odkazuje na konkrétní repositář služby GitHub a konkrétní git větev (branch), v němž je uložen zdrojový kód aplikace Docházka.

- Ř. 57 vytvoří obdobný Azure zdroj jako na ř. 48, který je též potřebný pro integraci Azure Web App a GitHub služeb. V podstatě obsahuje objekt vytvořený v předchozím kroku, jenž odkazuje na konkrétní GitHub repositář se zdrojovým kódem aplikace, která bude nasazena do této instance Azure Web App Service.
- Oba zdroje vytvořené na ř. 48 a 57 pak poskytují nezbytnou konfiguraci k tomu, aby instance služby Azure App Service získala přístup do GitHub repositáře a mohla si stáhnout kód aplikace, kdykoliv je kód v repositáři aktualizován (známo jako Continous Deployment).

4.3.3 Integrace služeb Azure App Service a Azure SQL Database

Tato část realizuje:

- User Story 45: Integrace služeb Azure App Service a Azure SQL Database,

viz odstavec 4.1.2.

Použitý PowerShell kód ukazuje Obrázek 69 a je inspirován průvodcem [32].

Obrázek 69 - PowerShell skript, část integrace služeb Azure App Service a Azure SQL Database

AzurePublishCommands.ps1 | LoggerExtensions [from metadata] | Program.cs | Dochazka: Publish | SeedData.cs | Startup.cs

```
60  # 4. User Story 45: Integrace služeb Azure App Service a Azure SQL Database
61  $webAppConnectionStrings = @{
62      DefaultConnection = @{
63          Type  = "SQLAzure";
64          Value = $connectionString
65      }
66  }
67
68  Set-AzWebApp -ResourceGroupName $rgName -Name $webAppName -ConnectionStrings $webAppConnectionStrings
69
70  # zde končí část nastavení Azure
71  exit;
72
73  # 5. User Story 46: Migrace modelu databáze do instance služby Azure SQL Database
74  # nutno provést v lokálním package manageru Visual Studia
75  $connectionString = "<hodnota z původní lokální proměnné $connectionString z předchozí PowerShell seance>"
76  $env:ConnectionStrings:DefaultConnection = $connectionString
77  rm -r Migrations
78  Add-Migration initialcreate
79  Update-Database
80  # $env:ConnectionStrings:DefaultConnection = "" # volat pouze podle potřeby k resetování Connection Stringu
81
82  # 6. Nutno provést v původní seanci PowerShellu, kde jsme připojení k Azure
83  Restart-AzWebApp -ResourceGroupName $rgName -Name $webAppName
84
```

- Ř. 61-66 vytvoří objekt typu `HashTable`, který reprezentuje connection string pro připojení k instanci Azure SQL serveru, jenž byl vytvořen v předchozích krocích

a jeho hodnota je uložena v lokální proměnné `$connectionString`, viz ř. 21, Obrázek 66.

- Ř. 68 přiřadí hodnotu objektu z předchozího kroku do Azure App Service konfigurační property `ConnectionStrings`. Podle této property je dále automaticky vytvořena lokální proměnná prostředí (environment variable) `SQLAZURECONNSTR_DefaultConnection` na hostovi, kde běží naše aplikace projektu `Dochazka.csproj`. Jde o obdobný mechanismus jako byl popsán v 4.3.2.
- Tím je dokončena konfigurace instance Azure App Service, kdy aplikace v ní běžící je od tohoto okamžiku schopná se připojit k SQL serveru pomocí admin connection stringu.

4.3.4 Migrace modelu databáze do instance služby Azure SQL Database

Tato část realizuje:

- User Story 46: Migrace modelu databáze do instance služby Azure SQL Database,

viz odstavec 4.1.2.

Použitý PowerShell ukazuje Obrázek 69 a je inspirován průvodcem [32].

Tyto kroky je nutné provést v Package Manageru Visual Studia s otevřeným projektem `Dochazka.csproj`. Zajistí nám inicializaci instance Azure SQL Database, tj. vytvoření potřebných tabulek pro entitní modely, indexů, nastavení klíčů, omezení apod.

- Ř. 75 má za úkol inicializovat lokální proměnnou na hodnotu původní lokální proměnné `$connectionString` z předchozí PowerShell seance, v níž jsme vytvářeli Azure zdroje, a kde je uložený connection string k Azure SQL serveru, viz ř. 21, Obrázek 66.
- Ř. 76 nastaví lokální proměnou prostředí na hodnotu lokální proměnné z předchozího kroku. Tato lokální proměnná prostředí (environment variable) bude určovat, proti jakému SQL serveru se budou provádět následující migrační kroky Entity Frameworku Core.
- Ř. 77, smaže obsah složky `Migrations` v adresáři projektu `Dochazka.csproj`.
- Ř. 78, příkaz Entity Frameworku Core [28] vytvoří novou verzi migračních skriptů ve složce `Migrations`.

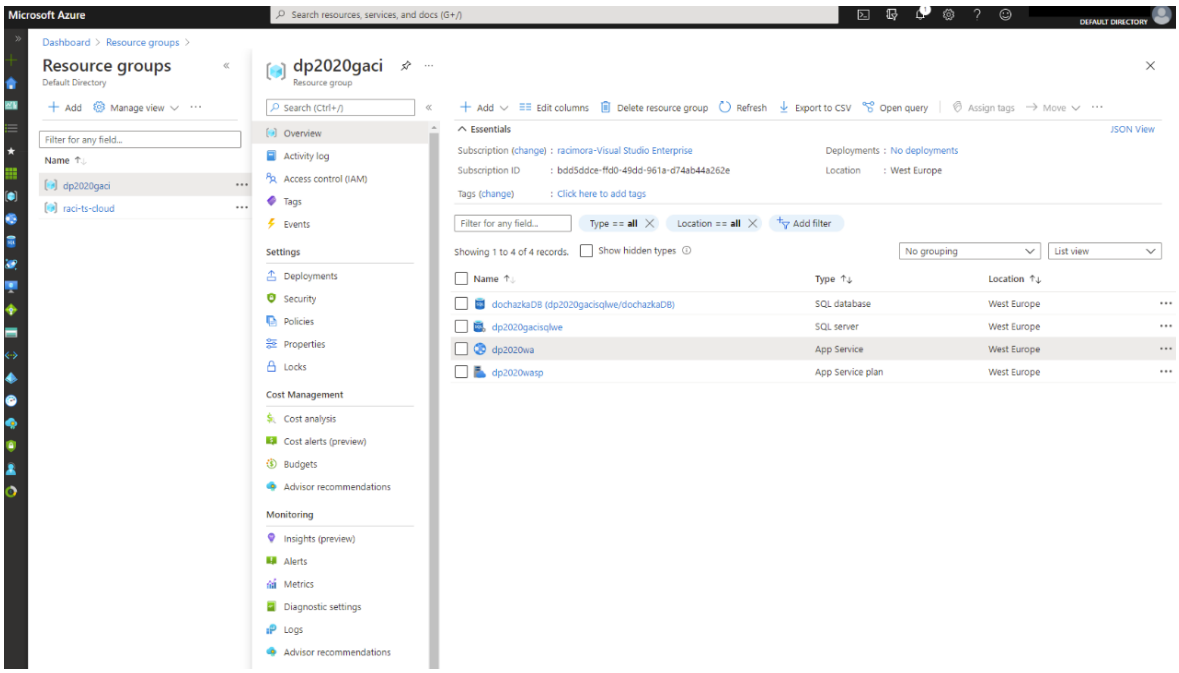
- Ř. 79, příkaz Entity Frameworku Core [28], který aplikuje migrační skripty proti Azure SQL serveru, jehož connection string je uložen v lokální proměnné prostředí.
- Ř. 83, v posledním kroku je ještě vhodné instanci Azure App Service restartovat. Tento krok je nutné provést v původní seanci PowerShellu, kde jsme připojeni k Azure. Tím se dokončí nasazení aplikace do Azure.

4.3.5 Prohlídka nasazené aplikace

Po dokončení předchozí části je aplikace nasazena do Azure cloudu a je připravena k použití.

Vytvořené Azure zdroje si lze mj. prohlédnout v Azure Portálu, což je uživatelsky přívětivé webové rozhraní služby Azure, viz Obrázek 70. Portál nabízí obrovské množství možností pro tvorbu, konfiguraci, správu, výkonové škálování, monitoring a administraci služeb Azure. Ty však v případě této práce byly vytvořeny pomocí PowerShell modulu Azure, což je jedna z mnoha dalších možností a nástrojů, jak lze spravovat Azure služby.

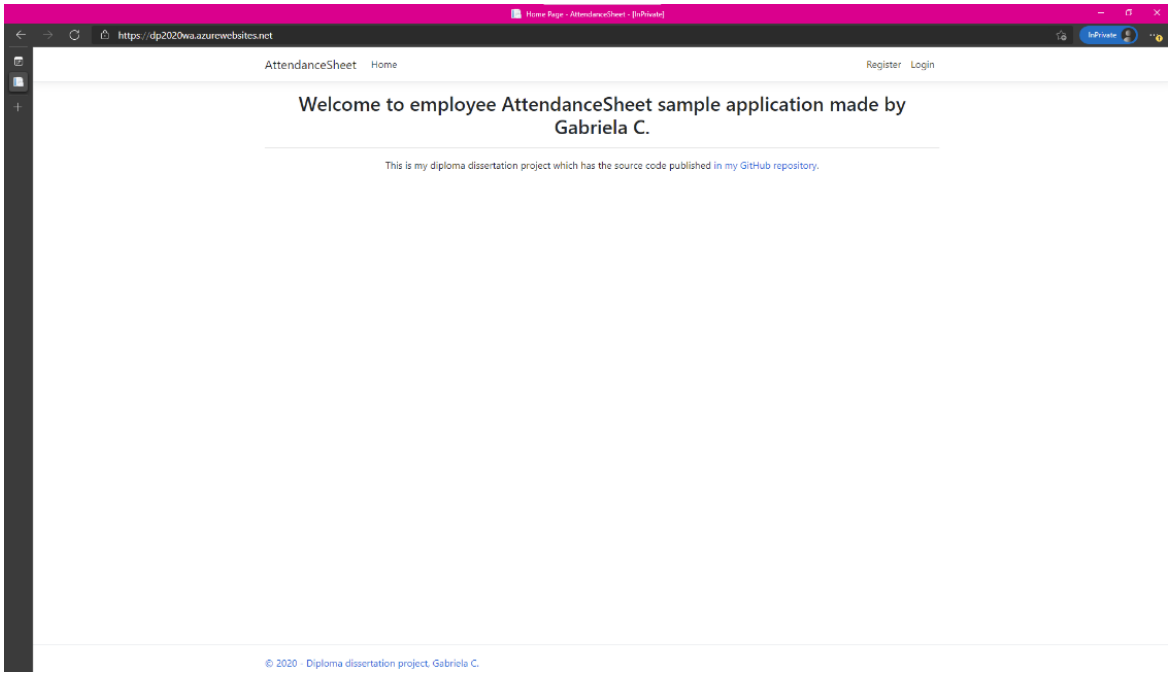
Obrázek 70 - Náhled na vytvořené Azure zdroje ve webovém portálu Azure



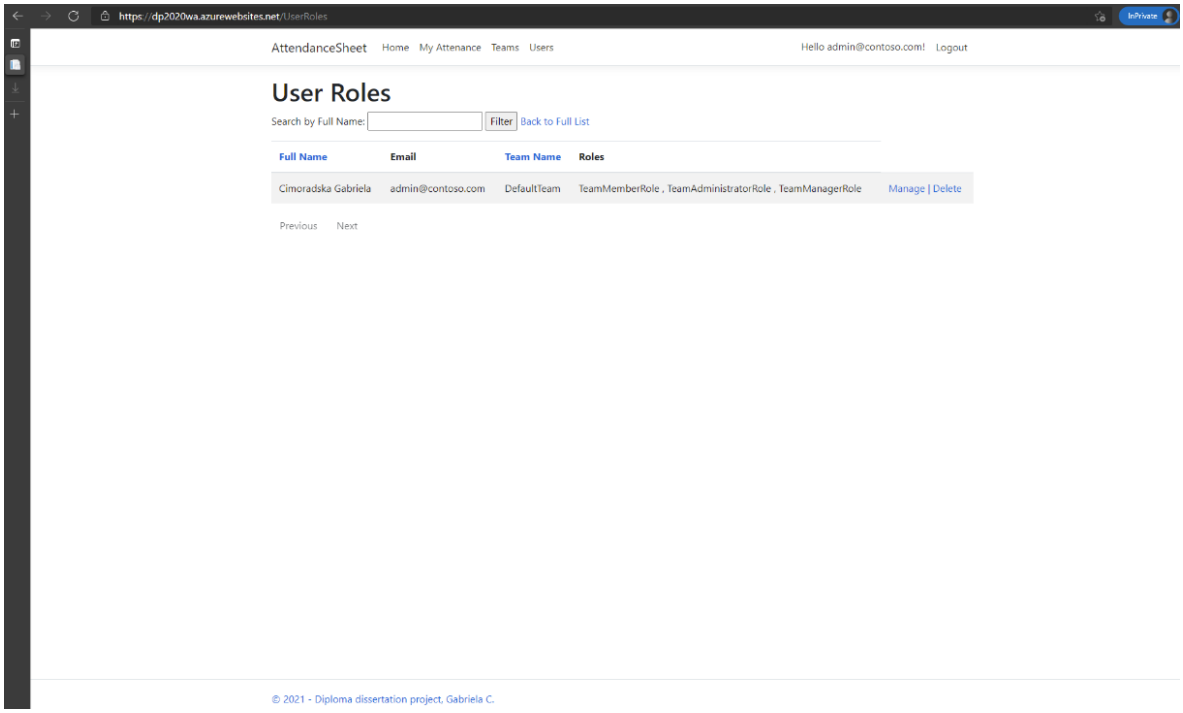
Po úspěšném nasazení aplikace je možné v prohlížeči navigovat na její adresu: https://dp2020wa.azurewebsites.net/

a přihlásit se jako uživatel `admin@contoso.com` a heslem nastaveným v odstavci `4.3.2`, viz Obrázek 71 a Obrázek 72. Následně lze aplikaci začít plně používat. Heslo je možné samozřejmě změnit v uživatelském profilu.

Obrázek 71 - Domácí stránka aplikace Docházka nasazená do Azure a otevřená v prohlížeči



Obrázek 72 - Stránka aplikace Docházka hostovaná v Azure, po přihlášení admin uživatele



5 Výsledky a diskuse

5.1 Uživatelský test

Aplikace byla podrobena uživatelskému testu od několika nezávislých uživatelů. Pro testování jim byl předán popis aplikace se zadáním testovacích úkolů, viz Příloha č.1 této práce, které měli provést a následně ohodnotit obtížnost jejich splnění.

5.2 Vyhodnocení uživatelského testu

Výsledky uživatelských testů jsem obdržela od 5 respondentů a jejich agregovaný výsledek s autentickými komentáři uvádí Tabulka 2. V prostředním sloupci je tučně vyznačeno průměrné hodnocení testovacích kroků.

Tabulka 2 - Agregovaný výsledek uživatelských testů

Test/Úkol	Známka [1-4]	Komentář/Chyba aplikace
TeamMemberRole/1	1,1,1,1,1=>**1**	
TeamMemberRole/2	1,1,1,1,1=>**1**	
TeamMemberRole/3	2,1,1,2,1=>**1.4**	1. Při vytvoření záznamu by mohl být přednastavený datum/kalendář podle vyfiltrovaného měsíce. 2. V políčku „Filter by Month“ není uvedený výchozí datum/měsíc, ale jen tečky. Je to matoucí. 3. V menu je uveden název „My Attenance“ místo „My Attendance“. Uživateli bych umožnil editovat více dní současně. Nyní musí řešit každý den zvlášť. 4. Dále bych v tabulce upravil název sloupce UserName na Email, když se zde poté uvádí emailová adresa uživatele. 5. Filtrování by mohlo být řešeno formou nějakého kalendáře. 6. Počet zobrazených záznamů by mohl být vyšší nebo volitelný.
TeamMemberRole/4	1,1,1,2,1=>**1.2**	7. Data v souboru souhlasí, ale nejsou rozdělena do jednotlivých sloupců. Všechna data jsou ve sloupci A.
TeamMamanagerRole/1	1,1,1,1,1=>**1**	
TeamMamanagerRole/2	2,1,1,2,1=>**1.4**	8. Po návratu z editace záznamu do seznamu by mohl být zachován filtr. 9. Intuitivně jsem hledal v záložce Teams, až potom My Attendance. Očekával bych schválení všech najednou, což však

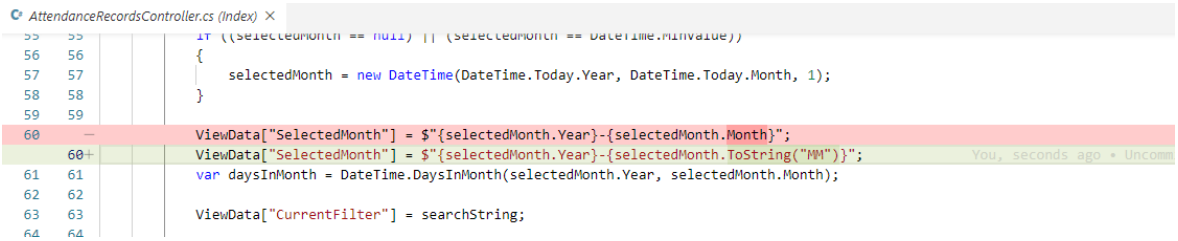
		fungovalo jen v rámci viditelných záznamů. Další stránku bylo potřebné schválit znova. 10. Asi by bylo vhodné rozdělit docházku Managera a docházku podřízených. Řešení v podobě filtru mi nepřijde příliš přívětivé. 11. U „Filter by Full Name“ bych dala možnost rozkliknutí a vybrání uživatele, podobné jako u „Filter by Month“. Nyní se musí uživatel psát ručně. Takové filtry usnadňují čas, nešetřila bych s nimi. 12. V rámci schvalování docházky jsem narazil na to, že se ve výběru z možností zobrazuje aktuálně vybraná položka vždy dvakrát.
TeamMamanagerRole/3	1,1,4,3,4=>**2.6**	13. Neodpovídají počty UnpaidVacation na AttendanceSheet a na Payroll Summary viz přílohy. Neodpovídají počty UnpaidVacation na Payroll Summary a v CSV viz přílohy. 14. Data v souboru neodpovídají datům v aplikaci. V aplikaci chybí údaj o UnpaidVacation. Z nějakého důvodu není uveden měsíci 2/2021, ale je uveden až ve 3/2021. V souboru jsou naopak data v pořádku, ale nesouhlasí pořadí jednotlivých sloupců (jsou přeházené oproti aplikaci). 15. Data v souboru opět nejsou rozdělena do jednotlivých sloupců. Vše je uvedeno ve sloupci A. 16. Chybí „Full name“ ve výpisu, ale filtr tuto možnost nabízí, pak tedy dle toho nelze filtrovat. 17. V obrazovce výpisu docházky mám nastavený filtr, ale po přechodu na obrazovku „payroll“ se mi tento filtr stejně zruší.
TeamAdministratorRole/1	1,1,1,1,1=>**1**	
TeamAdministratorRole/2	1,1,1,1,1=>**1**	18. Při návratu z editace uživatele na seznam uživatelů by mohl být zachován filtr. 19. Rozšířil bych seznam zobrazených uživatelů z 5 alespoň na 10.
TeamAdministratorRole/3	1,1,1,1,1=>**1**	20. I zde bych rozšířil seznam zobrazených teamů. Na stránce je spousta volného místa. 21. Dále bych opravil název sloupce „Team Manager UserName“ na „Team Manager Email“. Napříč aplikací se tyto termíny liší. Někde je uvedeno UserName a jinde naopak Email, tak tedy zvážit, zda to nesjednotit.
TeamAdministratorRole/4	1,1,1,1,1=>**1**	

Nyní k jednotlivým výsledkům a komentářům podrobněji.

Komentáře 1, 2

- Identifikováno jako chyba, jenž byla reprodukována a nalezena v třídě Controlleru `/Controllers/AttendanceRecords/Index.cs`, kde bylo třeba upravit vstupní hodnoty pro filtr kalendářního měsíce tak, aby správně interpretoval výchozí nebo vybraný kalendářní měsíc a nezobrazovaly se v něm jen tečky. Chyba byla následně opravena v akční metodě `Index a PayrollSummary` pomocí kódu, který ukazuje Obrázek 73. Zde se nastavuje hodnota, která bude v kalendáři přednastavena jako výchozí.

Obrázek 73 - Oprava chyby zobrazení filtru kalendář, akce Index a PayrollSummary



Komentář 3

- Tato chyba byla zjevně „přepis“ (typo) v kódu šablony View `_Layout.cshtml`, a byla odhalena již dříve a následně opravena.

Komentáře 4, 19

- Zde se jedná o nedorozumění, zda ve sloupcích s označením `*UserName*` má být kontaktní email nebo platné `UserName`. Ve skutečnosti tyto sloupce opravdu vypisují platné `UserName` uživatele, protože email zadaný při registraci uživatele se stává současně i jeho `UserName`. Ačkoliv si emailový kontakt uživatel může později změnit, tak `UserName` nikoliv.

Komentář 5

- Filtr s kalendářem je již na stránce přítomen, takže tento komentář není zřejmý.

Komentáře 6, 17, 18

- Stránkování s velikostí stránky 5 položek seznamu je evidentně příliš málo. U pokročilejších aplikací se často používá ovládací prvek Dropdown list (Selection list), který umožňuje uživateli vybrat si velikost stránky podle vlastní preference.

Zde by se zřejmě hodilo něco podobného. V současné verzi lze změnit velikost stránky pro celou aplikaci ve třídě `CommonConstants.cs`, pomocí konstanty `public const int PAGE_SIZE`. Na základě zpětné vazby byla velikost stránky navýšena globálně v celé aplikaci pomocí uvedené konstanty na 10 položek seznamu.

Komentáře 7, 15

- Formát CSV souborů byl ověřen a je validní v obou použitých případech. Uživatel zřejmě narazil na problém, kdy Microsoft Excel a podobné programy obvykle standardně načtou CSV soubor tak, že celý řádek ze souboru uloží do jednoho sloupce. Tomu lze předejít, pokud je CSV soubor správně naimportován a čárky jsou použity jako oddělovače sloupců.

Komentář 8, 16

- Ano, nastavení filtru pro stránku `Index` je po jejím opuštění obvykle ztraceno a nutí uživatele filtr znovu a znovu nastavit. Zde je jistě prostor pro vylepšení logiky filtrování. Řešení by mohlo spočívat ve vytvoření objektu, který by si pamatoval nastavení filtru pro jednotlivé stránky a předával se mezi stránkami.

Komentáře 9, 10

- Poznámky se vztahují ke způsobu schválení docházky manažerem. Zde se nabízí, jestli by nebylo vhodné oddělit práci s osobní docházkou do jednoho Controlleru a práci s docházkou podřízených do druhého, a to i na úrovni položek menu. Tím by se zřejmě otevřel také prostor pro rozšíření možností, jak rychleji schválit či editovat velké množství záznamů docházky, které může být za určitých okolností poměrně pracné.

Komentář 11

- Jedná se o validní poznámku. Dropdown list (Selection list) pro filtr podle jména, by se dal jistě přidat jako další filtr a mohlo by se tak usnadnit hledání záznamů konkrétního uživatele.

Komentář 12

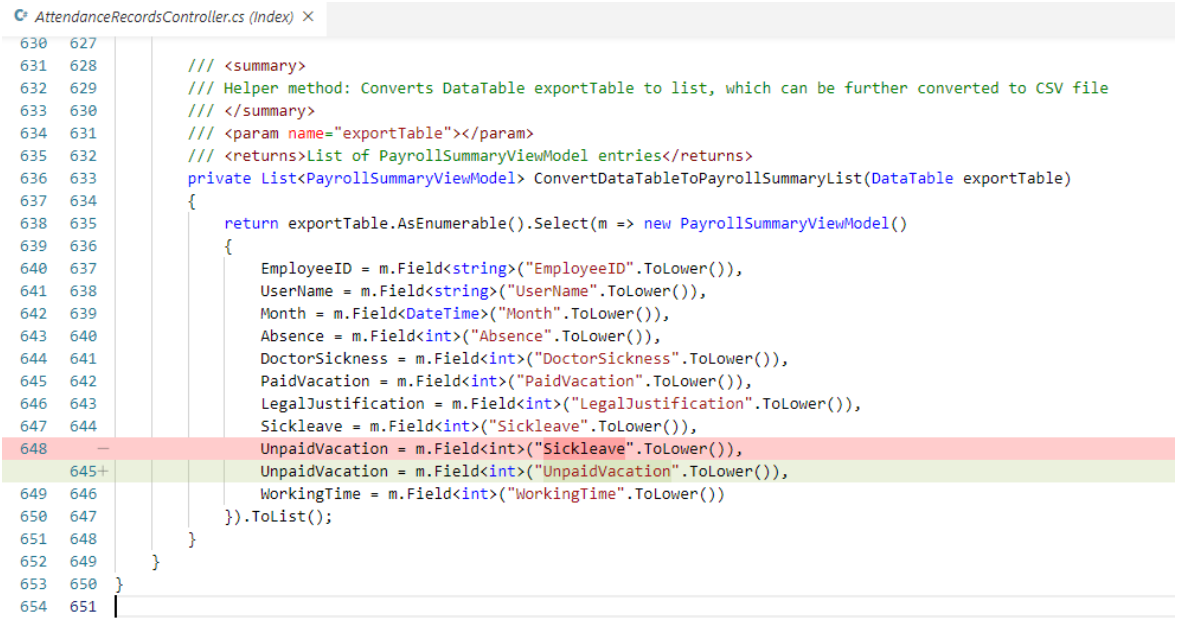
- Identifikováno jako chyba, jenž byla reprodukována a nalezena v kódu pro Selection list na stránce `/Views/AttendanceRecords/Index.cshtml`. Při schválení docházky se skutečně zobrazovala aktuální hodnota pole dvakrát. Chyba byla opravena, viz Obrázek 74.

Obrázek 74 - Oprava Selection listu ve /Views/AttendanceRecords/Index.cshtml

Komentáře 13, 14

- Tyto komentáře byly také identifikovány jako validní a úspěšně reprodukovány. Díky nim byla nalezena podstatná chyba v aplikační logice. Chybný kód byl odhalen v Controlleru `/Controllers/AttendanceRecordsController/Index.cs`, v metodě `ConvertDataTableToPayrollSummaryList`. Opravený kód ukazuje Obrázek 75.

Obrázek 75 – Opravený kód metody ConvertDataTableToPayrollSummaryList



Na místě by bylo ještě pokrýt akční metodu Controlleru `HttpGet PayrollSummary` unit testem, aby se podobným chybám v této kritické části aplikační logiky předešlo.

Komentář 16

- Popsaný nedostatek byl opět reprodukován a bylo nalezeno několik míst v kódu, které vyžadovaly opravu. Nejprve bylo nutné rozšířit model `PayrollSummaryViewModel.cs` o novou property `public string FullName { get; set; }`, aby s ní bylo možné pracovat ve výpisu agregované docházky. Dále bylo zapotřebí aktualizovat pomocné metody v Controlleru `AttendanceRecordsController.cs`, které souvisí s přípravou agregovaných dat pro tento model, viz Obrázek 76. V neposlední řadě zbývalo doplnit nový sloupec s hodnotou property `FullName` do tabulky výpisu záznamů ve View `/Views/AttendanceRecords/PayrollSummary.cshtml`, viz Obrázek 78, ř. 86-88 (na obrázku ještě chybí obdobná změna v záhlaví tabulky).

Obrázek 76 - Oprava pomocných metod v AttendanceRecordsController.cs

AttendanceRecordsController.cs (Working Tree)

```
585 591     /// <summary>
586 592     /// Helper method: Generates DataTable with aggregated PayrollSummaryResults based on byEmployeeIDResults and selectedMonth.
587 593     /// This is aggregated view of attendance per employeeID and selectedMonth. Can be used for sallary calculation.
588 594     /// </summary>
589 595     /// <param name="byEmployeeIDResults"></param>
590 596     /// <param name="selectedMonth"></param>
591 597     /// <returns>DataTable with PayrollSummary entries</returns>
592 598     private async Task<DataTable> GetSummaryResultsAsDataTable(
593 599         Dictionary<string, Dictionary<string, int>> byEmployeeIDResults, DateTime selectedMonth)
594 600     {
595 601         DataTable table = new DataTable("ExportAsCsv");
596 602         DataColumn[] columns =
597 603         {
598 604             new DataColumn("employeeid", typeof(String)),
    605+             new DataColumn("fullname", typeof(String)),
599 606             new DataColumn("username", typeof(String)),
600 607             new DataColumn("month", typeof(DateTime))
601 608         };
602 609
603 610         table.Columns.AddRange(columns);
604 611
605 612         foreach (string attendanceValue in Enum.GetNames(typeof(Attendance)))
606 613         {
607 614             table.Columns.Add(new DataColumn(attendanceValue.ToLower(), typeof(int)));   xsubg001, 9 days ago • Gabika/develop (#7)
608 615         }
609 616
610 617         foreach (var employeeID in byEmployeeIDResults.Keys)
611 618         {
612 619             DataRow row = table.NewRow();
    620+             var user = await _userManager.FindByIdAsync(employeeID);
613 621             row["employeeid"] = employeeID;
614  —              row["username"] = await _userManager.FindByIdAsync(employeeID);
    622+             row["fullname"] = user.FullName;
    623+             row["username"] = user.UserName;
615 624             row["month"] = selectedMonth;
616 625             foreach (string attendanceValue in byEmployeeIDResults[employeeID].Keys)
617 626             {
618 627                 row[attendanceValue] = byEmployeeIDResults[employeeID][attendanceValue];
619 628             }
620 629
621 630             table.Rows.Add(row);
622 631         }
623 632
624 633         return table;
625 634     }
626 635
627 636
628 637     /// <summary>
629 638     /// Helper method: Converts DataTable exportTable to list, which can be further converted to CSV file
630 639     /// </summary>
631 640     /// <param name="exportTable"></param>
632 641     /// <returns>List of PayrollSummaryViewModel entries</returns>
633 642     private List<PayrollSummaryViewModel> ConvertDataTableToPayrollSummaryList(DataTable exportTable)
634 643     {
635 644         return exportTable.AsEnumerable().Select(m => new PayrollSummaryViewModel()
636 645         {
637 646             EmployeeID = m.Field<string>("EmployeeID".ToLower()),
    647+             FullName = m.Field<string>("FullName".ToLower()),
638 648             UserName = m.Field<string>("UserName".ToLower()),
```

Komentář 17

- Pro zlepšení zachování nastaveného filtru při přechodu mezi stránkami `Index` a `PayrollSummary` v rámci Controlleru `AttendanceRecordsController.cs` byla provedena malá změna v akční metodě `HttpGet PayrollSummary` a úprava odpovídajícího View, která zachovává nastavení pro filtry „Full Name search“ a „Filter by Month“ viz Obrázek 77 a Obrázek 78.

Obrázek 77 - Rozšíření metody PayrollSummary v AttendanceRecordsController.cs

AttendanceRecordsController.cs (Working Tree)

```
188 188     // GET: Payroll summary for selected employees
189         public async Task<IActionResult> PayrollSummary(string searchString, DateTime selectedMonth, bool getAsCsv)
    189+    public async Task<IActionResult> PayrollSummary(string searchString, DateTime selectedMonth, string currentFilter, bool getAsCsv)
190 190     {
191 191         _logger.LogInformation($"Request month value: {selectedMonth}");
192 192         if ((selectedMonth == null) || (selectedMonth == DateTime.MinValue))
193 193         {
194 194             selectedMonth = new DateTime(DateTime.Today.Year, DateTime.Today.Month, 1);
195 195         }
196 196
197 197         ViewData["SelectedMonth"] = $"{selectedMonth.Year}-{selectedMonth.ToString("MM")}";
198 198         var daysInMonth = DateTime.DaysInMonth(selectedMonth.Year, selectedMonth.Month);
199 199
    200+
    201+         if (searchString == null)
    202+         {
    203+             searchString = currentFilter;
    204+         }
    205+
200 206         ViewData["CurrentFilter"] = searchString;
201 207
202 208         var currentUserId = _userManager.GetUserId(User);
```

Obrázek 78 - Oprava ve View souboru /Views/AttendanceRecords/PayrollSummary.cshtml

PayrollSummary.cshtml (Working Tree)

```
79  82                      <td>
80  83                          @Html.DisplayFor(modelItem => item.EmployeeID)
81  84                      </td>
82  85                      <td>
    86+                         @Html.DisplayFor(modelItem => item.FullName)
    87+                     </td>
    88+                     <td>
83  89                          @Html.DisplayFor(modelItem => item.UserName)
84  90                      </td>
85  91                      <td>
86  92                          @Html.DisplayFor(modelItem => item.Month)
87  93                      </td>
88  94                      <td>
89  95                          @Html.DisplayFor(modelItem => item.WorkingTime)
90  96                      </td>
91  97                      <td>
92  98                          @Html.DisplayFor(modelItem => item.PaidVacation)
93  99                      </td>
94 100                      <td>
95 101                          @Html.DisplayFor(modelItem => item.UnpaidVacation)
96 102                      </td>
97 103                      <td>
98 104                          @Html.DisplayFor(modelItem => item.DoctorSickness)
99 105                      </td>
100 106                     <td>
101 107                         @Html.DisplayFor(modelItem => item.Sickleave)
102 108                     </td>
103 109                     <td>
104 110                         @Html.DisplayFor(modelItem => item.Absence)
105 111                     </td>
106 112                     <td>
107 113                         @Html.DisplayFor(modelItem => item.LegalJustification)
108 114                     </td>
109 115                 </tr>
110 116             }
111 117         </tbody>
112 118     </table>
113 119 }
114 120
115 121 <div>
116         <a asp-action="Index">Back to List</a>
    122+     <a asp-action="Index"
    123+        asp-route-selectedMonth="@ViewData["SelectedMonth"]"
    124+        asp-route-currentFilter="@ViewData["CurrentFilter"]">
    125+         Back to List
    126+     </a>
117 127 </div>
```

Z uvedeného vyhodnocení vyplývá, že uživatelský test byl velmi užitečný a přínosný i na tak malém vzorku testerů. Ačkoliv uživatelé zvládli zadané úkoly vesměs bez problémů, s pomocí jejich komentářů bylo nalezeno několik chyb, které bylo vhodné odstranit neprodleně. Ostatní komentáře jsou pak většinou spíše validní návrhy k dalšímu vylepšení aplikace a mohly by být řešeny v další etapě jejího vývoje.

6 Závěr

Podle mého úsudku, tato práce úspěšně splnila všechny hlavní cíle zadání.

V teoretické části práce byly popsány a odůvodněny zvolené technologie, které byly použity k realizaci řešení.

Následně, pomocí platformy Microsoft ASP.NET Core, byl vyvinut plně funkční prototyp webové aplikace pro evidenci pracovní docházky zaměstnanců, kterou by zřejmě bylo možné začít okamžitě používat v menší firmě.

Podle požadavků zadání, byla aplikace nasazena do veřejného cloudu Microsoft Azure, který umožňuje podle zakoupeného tarifu využít vysokou dostupnost pro služby App Service a SQL database, stejně jako průběžně škálovat výkon služeb bez omezení dostupnosti aplikace. Aktuálně, obě služby vykazují průměrný celkový náklad cca 4 € za měsíc a dodržení SLA pro databázi 99.99 %. Z hlediska infrastruktury, obě služby jsou takřka bezúdržbové, protože se jedná o služby typu PaaS.

Na základě uživatelských testů byla také získána cenná zpětná vazba pro další vývoj a objeveny některé funkční nedostatky, které byly okamžitě opraveny v nejnovější verzi aplikace a poté automaticky publikovány do cloudu.

Diplomová práce jako taková, měla pro mě samotnou velký význam z hlediska seznámení se s novými Microsoft technologiemi a moderním agilním procesem vývoje softwaru. Celkově ji považuji za velmi přínosnou pro svůj profesní rozvoj a získání přehledu o práci softwarového vývojáře.

7 Seznam použitých zdrojů

[1] Microsoft, "Introduction to .NET," Microsoft, 16 November 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/introduction. [Accessed 27 February 2021].

[2] Wikipedia, "NET Framework version history," 16 February 2021. [Online]. Available: https://en.wikipedia.org/wiki/.NET_Framework_version_history#.NET_Framework_4.7.2. [Accessed 28 February 2021].

[3] Microsoft, "NET Standard," Microsoft, 5 October 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/net-standard. [Accessed 27 Frebruary 2021].

[4] Microsoft, "NET architectural components," Microsoft, 5 October 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/components. [Accessed 28 February 2021].

[5] Microsoft, "Releases and support for .NET Core and .NET 5," Microsoft, 7 October 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/releases-and-support. [Accessed 28 February 2021].

[6] Microsoft, "Introduction to ASP.NET Core," Microsoft, 17 April 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0. [Accessed 28 February 2021].

[7] Microsoft, "Overview of ASP.NET Core MVC," 12 February 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1. [Accessed 28 February 2021].

[8] Microsoft, "Introduction to Identity on ASP.NET Core," 15 July 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-3.1&tabs=visual-studio. [Accessed 28 February 2021].

[9] Microsoft, "Entity Framework Core," 09 September 2020. [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Accessed 28 February 2021].

[10] Microsoft, "Compare EF Core & EF6," 23 January 2019. [Online]. Available: https://docs.microsoft.com/en-us/ef/efcore-and-ef6/. [Accessed 28 February 2021].

[11] R. R. Kumar, "A Basic Introduction To C# Unit Test For Beginners," 06 June 2019. [Online]. Available: https://www.c-sharpcorner.com/article/a-basic-introduction-of-unit-test-for-beginners/. [Accessed 28 February 2021].

[12] Microsoft, "Unit testing best practices with .NET Core and .NET Standard," 28 July 2018. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices. [Accessed 28 February 2021].

[13] Microsoft, "What is PowerShell?," 22 May 2020. [Online]. Available: https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.1. [Accessed 28 February 2021].

[14] Microsoft, "PowerShell Support Lifecycle," 11 November 2020. [Online]. Available: https://docs.microsoft.com/en-us/powershell/scripting/powershell-support-lifecycle?view=powershell-5.1. [Accessed 28 February 2021].

[15] Microsoft, "Introducing the Azure Az PowerShell module," 21 February 2021. [Online]. Available: https://docs.microsoft.com/en-us/powershell/azure/new-azureps-module-az?view=azps-5.6.0. [Accessed 28 February 2021].

[16] Wikipedia, "Git," 03 March 2021. [Online]. Available: https://en.wikipedia.org/wiki/Git. [Accessed 05 March 2021].

[17] Microsoft, "What is Azure DevOps?," 22 January 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops. [Accessed 01 March 2021].

[18] Microsoft, "Products available by region," [Online]. Available: https://azure.microsoft.com/en-us/global-infrastructure/services/?products=sql-database,app-service. [Accessed 1 March 2021].

[19] Microsoft, "App Service overview," 07 June 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/app-service/overview. [Accessed 01 March 2021].

[20] Microsoft, "What is Azure SQL Database?," 21 September 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview. [Accessed 1 March 2021].

[21] Microsoft, "Tutorial: Get started with EF Core in an ASP.NET MVC web app," 06 November 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-3.1. [Accessed 01 09 2020].

[22] Microsoft, "Scaffold Identity in ASP.NET Core projects," 01 May 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?view=aspnetcore-3.1&tabs=visual-studio#scaffold-identity-into-an-mvc-project-without-existing-authorization. [Accessed 01 September 2020].

[23] Microsoft, "Tutorial: Add sorting, filtering, and paging - ASP.NET MVC with EF Core," 27 March 2019. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/sort-filter-page?view=aspnetcore-3.1#add-paging-to-index-method. [Accessed 10 September 2021].

[24] Microsoft, "Dependency injection in ASP.NET Core," 21 July 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1. [Accessed 01 March 2021].

[25] "SOLID In C# – Dependency Inversion," 19 November 2019. [Online]. Available: https://dotnetcoretutorials.com/2019/11/10/solid-in-c-dependency-inversion/#:~:text=Broadly%20speaking,%20Dependency%20Injection%20is%20a%20way%20to,Dependency%20Injection%20is%20a%20way%20to%20achieve. [Accessed 01 March 2021].

[26] Microsoft, "Architectural principles," 01 December 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion. [Accessed 01 March 2021].

[27] stackoverflowcom, "Write to CSV file and export it?," StackOverFlow, [Online]. Available: https://stackoverflow.com/questions/3777874/write-to-csv-file-and-export-it. [Accessed 10 October 2021].

[28] Microsoft, "Entity Framework Core tools reference - Package Manager Console in Visual Studio," 27 October 2020. [Online]. Available: https://docs.microsoft.com/en-us/ef/core/cli/powershell. [Accessed 10 October 2021].

[29] Microsoft, "Configuration in ASP.NET Core," 01 January 2021. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-3.1. [Accessed 01 September 2020].

[30] Microsoft, "Safe storage of app secrets in development in ASP.NET Core," 24 November 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-3.1&tabs=windows. [Accessed 01 September 2021].

[31] Microsoft, "Unit test controller logic in ASP.NET Core," 22 July 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing?view=aspnetcore-3.1. [Accessed 10 November 2021].

[32] Microsoft, "Connect an App Service app to SQL Database," Microsoft, 03 March 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/app-service/scripts/powershell-connect-to-sql?toc=/powershell/module/toc.json. [Accessed 01 October 2021].

[33] Microsoft, "Create your Azure free account today," [Online]. Available: https://azure.microsoft.com/en-us/free/. [Accessed 01 September 2020].

[34] Microsoft, "Subscriptions, licenses, accounts, and tenants for Microsoft's cloud offerings," 17 March 2021. [Online]. Available: https://docs.microsoft.com/en-us/microsoft-365/enterprise/subscriptions-licenses-accounts-and-tenants-for-microsoft-cloud-offerings?view=o365-worldwide. [Accessed 20 March 2021].

[35] Microsoft, "Manage Azure Resource Manager resource groups by using Azure PowerShell," Microsoft, 01 September 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-powershell. [Accessed 01 October 2020].

[36] Microsoft, "Azure SQL Database pricing," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/sql-database/single/. [Accessed 20 March 2021].

[37] Microsoft, "SLA for Azure SQL Database," July 2019. [Online]. Available: https://azure.microsoft.com/en-us/support/legal/sla/app-service/v1_4/. [Accessed 28 March 2021].

[38] Microsoft, "Create a web app with continuous deployment from GitHub," 20 March 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/app-service/scripts/powershell-continuous-deployment-github?toc=/powershell/module/toc.json. [Accessed 20 October 2020].

[39] Microsoft, "SLA for App Service," July 2016. [Online]. Available: https://azure.microsoft.com/en-us/support/legal/sla/app-service/v1_4/. [Accessed 26 March 2021].

[40] GitHub, "Creating a personal access token," [Online]. Available: https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token. [Accessed 01 October 2021].

[41] J. CILIBERTI, ASP.NET Core Recipes, New Yourk: APress, 2015.

[42] J. CHAMBERS, P. David and T. Simon, ASP. NET Core Application Development: Building an application in four sprints, Microsoft Press, 2017.

[43] J. MURACH and M. DELAMATER, Murach´s ASP.NET Core MVC, Fresno: Mike Murach & Associates Inc., 2020.

[44] R. SIMON, C# Programujeme profesionálně, Brno: Computer Press, 2003.

8 Přílohy

Nedílnou součástí této práce jsou tyto přílohy:

Příloha č. 1	Popis aplikace se zadáním testovacích úkolů pro uživatelské testy
Příloha č. 2	CD se zdrojovými kódy aplikace