Návrh interní aplikace pro hodnocení členů týmu ve firmě

Bc. Adam Michálek

Diplomová práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Adam Michálek**
Osobní číslo:	**A20133**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Návrh interní aplikace pro hodnocení členů týmu ve firmě**
Téma práce anglicky:	**Design of an Internal Company Application for Team Member Evaluation**

Zásady pro vypracování

1. Nastudujte potřebnou problematiku související s tématem.
2. Zvolte vhodné prostředky pro implementaci frontendové i backendové části aplikace.
3. Navrhněte aplikaci s ohledem na plné nasazení pomocí AZURE devOps technologií.
4. Implementujte samotnou aplikaci včetně všech potřebných součástí.
5. Výslednou aplikaci vhodně otestujte a vhodně prezentujte výsledky.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. MARTIN, Robert C. *Čistý kód*. Brno: Computer Press, 2009, 423 s. ISBN 9788025122853.
2. MARTIN, Robert C. *Clean architecture: a craftsman's guide to software structure and design*. Boston: Addison-Wesley, [2018], xxv, 404 s. Robert C. Martin series. ISBN 978-0-13-449416-6.
3. BUONANNO, Enrico. *Functional Programming in C#: How to Write Better C# Code*. August 12, 2017. Manning Publications, 2017. ISBN 9781617293955.
4. FREEMAN, Adam. *Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages*. 8th ed. 2020. Berkeley, CA: APress, 2020 ;, 1 online zdroj (XXIX, 1080 stran). ISBN 9781484254400. Dostupné také z: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&AN=2494147&authtype=ip,shib&custid=s393
5. MALIK, Amit, Sjoukje ZAAL a Stefano DEMILIANI. *Azure DevOps Explained: Get Started with Azure DevOps and Develop Your DevOps Practices*. December 11, 2020. PACKT PUBLISHING LIMITED, 2020. ISBN 9781800563513.
6. ERL, Thomas. *SOA: servisně orientovaná architektura : kompletní průvodce*. Brno: Computer Press, 2009, 671 s. Programování. ISBN 9788025118863.

Vedoucí diplomové práce: **Ing. Petr Žáček, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **3. prosince 2021**
Termín odevzdání diplomové práce: **23. května 2022**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Adam Michálek, v. r.
podpis studenta

ABSTRAKT

V práci je navržena interní aplikace s názvem ReviewMe pro hodnocení členů teamu ve firmě. Jednotlivé části aplikace jsou nasazeny samostatně a integrovány v rámci SOA architektury s použitím služby pro zasílání zpráv. Backendová část aplikace je vytvořena v .NET Core a zveřejňuje jednoduché API s požadovanou funkcionalitou. To je dále využíváno frontendovou částí (implementace v Blazor) pro zobrazení všech operací v přehledné formě. Rovněž je vytvořena jednoduchá mikro služba (EmployeeMicroservice), získávající data z externí služby, které posílá do zprostředkovatele zpráv (v tomto případě RabbitMQ). Ten musel být nastaven tak, aby splňoval požadavky na jednotnou komunikaci v rámci systému a zabezpečení dat. Tyto data ReviewMe odposlouchává, zpracovává a ukládá do své databáze. Aplikace je použitelná jako modul s možností ji jednoduše vložit do stávajícího systému, ale zároveň dodržuje principy Mikroservise architektury a může být použita samostatně. V případě výpadku zdroje dat tedy zůstane určitou dobu nedotečena. Proto jsou zde také navrženy prototypy pro části odposlouchávající a zpracovávající data (pub/sub). Ty mohou být později přidány k dalším aplikacím, které se stanou součástí systému. Teoretická část je věnována obecnému popisu zvolených technologií a principů čistého kódu a architektury, používané v projektu. V rámci organizace a plánování byly použity Devops technologie. Teoretické poznatky jsou poté aplikovány při implementaci a testování všech součástí práce.

Klíčová slova: Mikro služba, Devops, ReviewMe, EmployeeMicroservice, RabbitMq, Blazor

ABSTRACT

An internal application called ReviewMe is designed for the evaluation of team members in the company. The individual parts of the application are deployed separately and integrated within the SOA architecture using a messaging service. The backend part of the application is created in .NET Core and publishes a simple API with the required functionality. This is further used by the frontend part (implementation in Blazor) to display all operations in a clear form. A simple micro service (EmployeeMicroservice) is also created, retrieving data from an external service that it sends to a message provider (in this case, RabbitMQ). It had to be set up to meet the requirements of unified system communication and data security. ReviewMe will listen to, process and store this data in its database. The application can be used as a module with the possibility to easily insert it into the existing system, but at the same time it follows the principles of the Microservice architecture and can be used separately. Therefore, in the event of a data source failure, it will remain intact for some time. Therefore, a prototype for parts listening and processing data (pub / sub) was designed here. These can later be added to other applications that become part of the system. The theoretical part is devoted to a general description of the technologies used and the principles of clean code and architecture used in the project. Devops technologies were used in the organization and planning. Theoretical knowledge is then applied in the implementation and testing of all parts of the work.

Keywords: Microservice, Devops, ReviewMe, EmployeeMicroservice, RabbitMq, Blazor

Poděkování

Tímto bych chtěl poděkovat vedoucímu své diplomové práce Ing. Petr Žáček, Ph.D. a lidem z firmy CN Group za jejich odbornou pomoc při práci na tomto projektu. Děkuji rovněž svým rodičům, kteří mě při psaní práce podrželi a poradili s následnou opravou.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD ... **11**
I TEORETICKÁ ČÁST ... **13**
1 .NET ... **14**
1.1 .NET CORE ... 14
1.1.1 ASP.NET Core ... 15
1.1.2 .NET Core 6 ... 15
1.2 BLAZOR ... 16
1.2.1 Komponenty ... 16
1.2.2 Client side ... 17
1.2.3 Server side ... 18
1.3 BEZPEČNOST V .NET ... 18
1.3.1 Autentizace ... 19
1.3.2 Autorizace ... 20
1.3.3 Identity API ... 20
2 ČISTÁ ARCHITEKTURA A KÓD ... **22**
2.1 ČISTÁ ARCHITEKTURA ... 22
2.1.1 Popis jednotlivých vrstev ... 23
2.1.2 Výhody a nevýhody ... 24
2.2 ATOMICKÝ DESIGN ... 25
2.3 FUNKCIONÁLNÍ PROGRAMOVÁNÍ V C# ... 26
2.3.1 Základní principy FP ... 26
2.3.2 Jazykové konstrukty podporující FP ... 27
2.3.3 LINQ knihovna ... 30
2.4 UNIT TESTY ... 31
3 SOA ARCHITEKTURA ... **32**
3.1 SOA PRINCIPY ... 33
3.2 MONOLITICKÁ ARCHITEKTURA ... 34
3.3 MICROSERVICE ARCHITEKTURA ... 35
3.3.1 SOA vs Microservice architektura ... 35
3.3.2 Porovnání jednotlivých architektur ... 36
4 ZPROSTŘEDKOVATEL ZPRÁV (MB) ... **37**
4.1 MODELY MB ... 37
4.2 MB VS API ... 38
4.3 ESB VS MB ... 38
4.4 RABBITMQ ... 39
4.4.1 Základní pojmy a principy ... 40
4.4.2 Směrování a klíče ... 41

5 DEVOPS 42
5.1 ZÁKLADNÍ PRINCIPY A NÁSTROJE 43
5.2 PRŮBĚŽNÁ INTEGRACE (CI) 44
5.3 PRŮBĚŽNÁ DORUČOVÁNÍ (CD) 45
5.4 AZURE DEVOPS 46
II PRAKTICKÁ ČÁST 47
6 POŽADAVKY NA APLIKACI 48
6.1 FUNKCIONÁLNÍ A NEFUNKCIONÁLNÍ POŽADAVKY 48
6.1.1 Zaměstnanec 48
6.1.2 Administrátor 49
6.1.3 Aplikace v systému 50
6.2 REŠERŠE PODOBNÝCH APLIKACÍ 51
7 NÁVRH ARCHITEKTURY 52
7.1 SCHÉMA ARCHITEKTURY 52
7.2 VÝBĚR TECHNOLOGIÍ 54
7.3 KONVENCE 55
8 DEVOPS A RABBITMQ NASTAVENÍ 56
8.1 DEVOPS 56
8.1.1 Backlog a plánování 56
8.1.2 CI 57
8.1.3 CD 58
8.1.4 Nastavení serveru 58
8.2 RABBITMQ 59
8.2.1 RabbitMQ UI Management 60
8.2.2 Topic permissions 62
9 REVIEWME BE ČÁST 63
9.1 VRSTVY ARCHITEKTURY A JEJICH POPIS 63
9.1.1 ReviewME.API 64
9.1.2 ReviewME.CORE a Tests 66
9.1.3 ReviewME.Infrastructure.DbStorage 68
9.1.4 ReviewME.Infrastructure.EmailSender a Razor templates 69
9.1.5 ReviewME.Infrastructure.RabbitMQConsumer 70
9.2 BUSINESS LOGIKA 71
9.2.1 Výpis všech zaměstnanců 71
9.2.2 Otevření a práce s ohodnoceními 72
9.2.3 Provedení ohodnocení a další operace 74
9.2.4 Další služby 75
9.3 SYSTÉM ZASÍLNÍ NOTIFIKACÍ 76
9.3.1 Šablony, tvorba a poslání emailu 76

9.3.2 Notifikační služba ... 78
9.4 RabbitMQ subscriber ... 80
9.4.1 Registrace RabbitMq knihovny a hostovaná služba ... 80
9.4.2 Komponenty pro zpracování zpráv ... 83
9.4.3 Synchronizační služba ... 84
10 REVIEWME FE ČÁST ... 85
10.1 Vrstvy a jejich popis ... 85
10.1.1 ReviewME.Models ... 85
10.1.2 ReviewME.Frontend ... 85
10.1.3 ReviewME.Components ... 86
10.2 Implementace stránky pro ohodnocení ... 87
10.2.1 Získání dat a vytvoření stránky ... 87
11 EMPLOYEEMICROSERVICE ČÁST ... 90
11.1 Vrstvy a jejich popis ... 90
11.2 Synchronizace dat ... 91
11.3 RabbitMq publisher ... 92
12 TESTOVÁNÍ, VYHODNOCENÍ A POUŽITÍ APLIKACE ... 94
12.1 ReviewMe BE ... 94
12.2 ReviewMe FE ... 97
12.2.1 List zaměstnanců ... 97
12.2.2 Vytváření ohodnocení ... 97
12.2.3 Zaslané notifikační emaily ... 99
12.2.4 List úkolů pro hodnocení ... 100
12.2.5 Vyplnění zpětné vazby ... 100
12.2.6 Zobrazení ohodnocení administrátorem ... 101
12.3 Testování distribuce dat ... 101
12.4 Použitelnost aplikace v praxi ... 103
ZÁVĚR ... 105
SEZNAM POUŽITÉ LITERATURY ... 107
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 112
SEZNAM OBRÁZKŮ ... 113
SEZNAM PŘÍLOH ... 116

ÚVOD

Nacházíme se v době, kdy distribuované systémy nabývají více a více na významu. Kryptoměny, internet 2.0, banky, velké společnosti jako Amazon či Netflix, čím dál více využívají podobných systému pro jejich aplikace, ať už interní, nebo ty které poskytují koncovým zákazníkům. Vypadá to, že tento trend bude v budoucnosti pokračovat, protože operovat s větším množstvím menších služeb, nebo mikro služeb, které nejsou pevně svázány, má velké množství výhod podrobněji rozebraných v teoretické části. Aplikace, která je předmětem této práce, do určité míry zachovává tento styl architektury, tedy nezávislost na dalších službách v systému a udržování dat, které nezbytně potřebuje k provozu.

Webová aplikace ReviewMe je primárně určena pro firemní interní potřeby hodnocení jednotlivých členů teamu, a to hlavně před pravidelnou schůzí o produktivitě zaměstnance (PR – performance review). Hodnocení jsou sbírána od ostatních členů teamu, popřípadě od dodatečných osob přidaných HR či administrátorem. Hodnocení je v podobě písemné odpovědi s předpřipravenými sekcemi. Aplikace dále obsahuje administrátorskou sekci s možností vytvoření nového posouzení (assessment) pro konkrétního zaměstnance. Zde je možné vybrat množství lidí, kteří budou daného zaměstnance hodnotit, do kdy bude probíhat sběr relevantních dat a kdy daný jedinec bude mít PR.

Teoretická část práce se převážně zabývá technologiemi a principy, které byly použity při návrhu aplikace a dalších komponent či mikro služeb. První část (Kapitola 1) se věnuje frameworkům, pomocí kterých byla vyvinuta BE a FE část aplikace včetně zajištění dostatečné míry zabezpečení. Následující kapitola (Kapitola 2) se zabývá čistou architekturou a kódem. Tyto principy byla snaha udržet v každé části projektu. Budou zde zmíněny konstrukty z LINQ knihovny vycházející z principů funkcionálního programování, dále práce s Entity frameworkem a databází, návrh unit testů pro jednotlivé vrstvy architektury a atomický design na FE části. Další kapitola (Kapitola 3) se věnuje SOA architektuře a jejímu porovnání s Mikroservise architekturou, protože obě jsou v projektu využity. SOA architektura často pro komunikaci potřebuje nějakého zprostředkovatele zpráv. V následující kapitole (Kapitola 4) bude vysvětlen jeho princip, proč byl zvolen zrovna RabbitMQ a jaké jsou jeho výhody. Poslední kapitola (Kapitola 5) se věnuje plánování, organizaci v projektu, průběžnému doručování, nasazení a dalším operacím, které zjednodušily vytváření tohoto projektu a pomohly udržet jeho kvalitu.

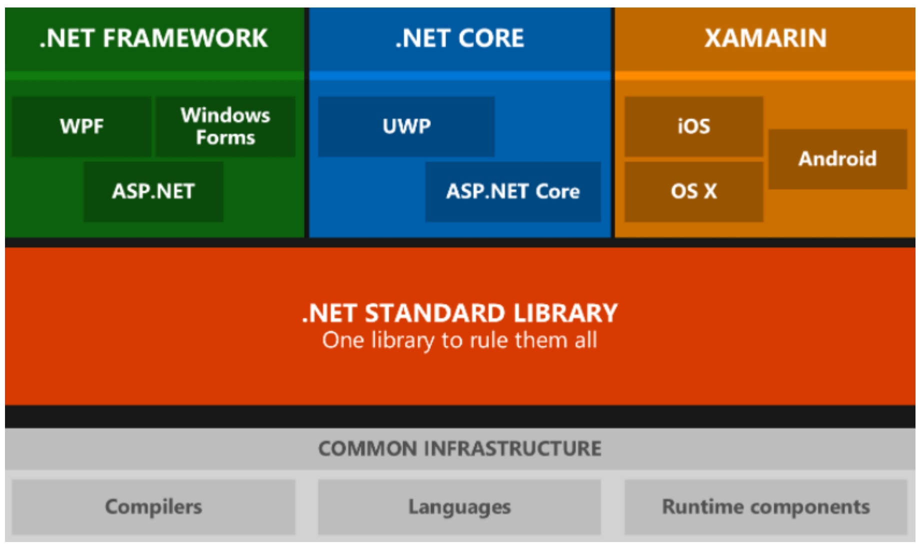
Praktická část práce se zabývá návrhem, implementací a testování samotné aplikace a dalších části v rámci tohoto projektu. V první části (Kapitola 6) budou definovány požadavky na aplikaci, které bylo potřeba zohlednit při architektonickém návrhu. Tím se poté zabývá další (Kapitola 7) část. Bude zde představen návrh BE a FE části, mikro služby pro zpracování dat a rovněž vytvořeno schéma zobrazující celkovou podobu architektury se všemi komponentami. Organizace, verzování, průběžná integrace a nasazení (CI a CD) za pomocí Azure devops budou předmětem další kapitoly (Kapitola 8). Následující kapitola (Kapitola 9) se věnuje implementaci BE části aplikace. Zde budou popsány hlavní vrstvy a principy v rámci business logiky, včetně všech externích závislostí, nezbytných pro chod aplikace. Jednou z nich je například RabbitMQ subscriber pro zpracování dat přicházejících z externích aplikací. Dále se zaměříme na FE část (Kapitola 10), která využívá BE API a zobrazuje přehledně vykonané akce. Poslední implementační část (Kapitola 11) se věnuje mikro službě, která bude mít za úkol průběžně odposlouchávat data ze zaměstnaneckého portálu a posílat je dále do RabbitMQ. Ten musí být samozřejmě nastaven tak, aby data dokázal zpracovat a poslat na příslušné místo (například na BE ReviewMe, který zprávu zpracuje). Poslední kapitola (Kapitola 12) se věnuje testování, vyhodnocení a použití aplikace. Budou zde rozebrány jednotlivé části z uživatelského pohledu, popsán způsob testování a potenciál v praxi.

I. TEORETICKÁ ČÁST

1 .NET

.NET je platforma pro vývoj softwaru, skládající se z nástrojů a knihoven pro různé typy aplikací. Dostupných je i více implementací pro další platformy (Linux, MacOS). Základní komponentou je modul CLR (Common language runtime), poskytující správu paměti (garbage collector), práci s vlákny, výjimky, typovou bezpečnost (type safety) apod. Napsaný kód (v libovolném podporovaném jazyku, například C#, nebo F#) je vždy v prvním kroku zkompilován do CIL (Common intermediate language) a uložen v assembly (.dll soubor). Když je aplikace poté spuštěna, CLR pomocí JIT (Just in time) překladače vytvoří strojový kód pro konkrétní architekturu procesoru [1, 2].

.NET můžeme rozdělit na Framework a Core. Framework je vice orientovaný na nativní aplikace pro Windows. Core je multiplatformní implementace pro webové stránky, služby a konzolové aplikace. Poté máme ještě .NET Standard, což je formální specifikace sloužící ke sjednocení jeho ekosystému. Pokud chceme sdílet kód mezi jednotlivými implementacemi, naše knihovna by měla být ve verzi standard [2, 3].



Obrázek č.1 .NET ekosystém [4]

Na obrázku č.1 vidíme jednotlivé části ekosystému a jejich technologie.

1.1 .NET Core

Jak už bylo nastíněno výše, .NET Core je open-source framework pro vytváření aplikací na více platforem. Jedná se v podstatě o běhové prostředí pro aplikace typu ASP.NET Core a další [5].

1.1.1 ASP.NET Core

Jedná se o multiplatformní, výkonný open source framework pro vytváření moderních webových aplikací, služeb, BE pro mobily a mnoho dalších.

Poskytuje například **Razor pages**, které zjednodušují programovací model (odstraněním velkého množství MVC ceremonií) při vytváření webových aplikací s jednoduchými stránkami (například pouze pro čtení). Dalším současným trendem je **Blazor**, umožňující psát FE část aplikace v C# (společně s Javascriptem). O něm více v kapitole 1.2. Dále za zmínku stojí podpora **gRPC**, zabudovaná DI, jednoduchá a modulární http pipeline požadavků a mnoho dalších [6].

ASP.NET Core nám dále poskytuje předpřipravené šablony a návrhové vzory pro zavedení struktury v projektu a zvýšení přehlednosti a čitelnosti. Patří sem primárně:

1. MVC (Model–View-Controller) vzor, zvyšující přehlednost a testovatelnost aplikace
2. Razor stránky, zjednodušující vytváření UI
3. Tag pomocníky
4. Vazebný model (binding) pro mapování dat
5. Validace modelu [6]

Podporuje rovněž client-side vývoj, kdy ASP.NET můžeme jednoduše integrovat s populárními frameworky jako Angular, React a samozřejmě Blazor.

1.1.2 .NET Core 6

Nejnovější verze .NET, která vyšla na konci roku 2021, slouží hlavně ke sjednocení SDK a základních knihoven napříč mobilními, desktopovými a cloudovými aplikace. Kromě zvýšení výkonu (nový **FileStream**, **PGO**, **Crossgen2**) a produktivity při práci jsme se v nové verzi dočkali *Hot Reload* funkcionality, kdy při běhu aplikace můžeme upravit zdrojový kód a okamžitě vidět změnu bez nutnosti opětovného spuštění. Můžeme ji spustit pomocí tlačítka ve Visual Studiu, popřípadě pomocí příkazu [7]:

dotnet watch

Současně Core vyšla i nová verze jazyku C# - 10. Ta dále redukuje množství kódu, který je potřeba psát.

Mezi hlavní novinky patří:

1. global using – je možné nastavit společný using pro jednotlivé odkazované knihovny
2. file-scoped namespace – nová syntax pro jmenné prostory redukující množství kódu
3. vylepšené rozpoznávání vzoru – pro vlastnosti a další datové typy
4. Vylepšení Lambda výrazů
5. Kontrola nulových parametrů
6. Rozšíření záznam typu (Record) pro struktury [8]

1.2 Blazor

Jedná se o framework pro vytváření interaktivních client-side aplikací za pomocí .NET. Charakteristickou vlastností je použití C#, místo Javascriptu (ne vždy je to samozřejmě možné) na klientské straně a dále sdílení logiky mezi klientem a serverem (pomocí Shared modulu). Rovněž dostáváme přístup ke všem možným knihovnám a ekosystému v rámci .NET, včetně slušné úrovně zabezpečení, spolehlivosti a výkonu [9].

1.2.1 Komponenty

Základem Blazor aplikací jsou komponenty, elementy UI jako stránky, dialogy, tlačítka, formuláře a další. Jsou to C# třídy zabudované do .NET Assemblies, které zpracovávají uživatelské události a definují vykreslovací logiku pro UI. Tato třída je většinou ve formě Razor markup page (.razor), což je syntaxe kombinující HTML a C# kód [9].

Na obrázku níže můžeme vidět jednoduchou komponentu counter:

```
1.  @page "/counter"
2.
3.  <h1>Counter</h1>
4.  <p>Current count: @counter</p>
5.  <button class="btn btn-primary" @onclick="AddToCounter">Click here</button>
6.
7.  @code {
8.      private int counter = 0;
9.
10.     private void AddToCounter()
11.     {
12.         counter++;
13.     }
14. }
```

Obrázek č.2 Counter – Blazor komponenta

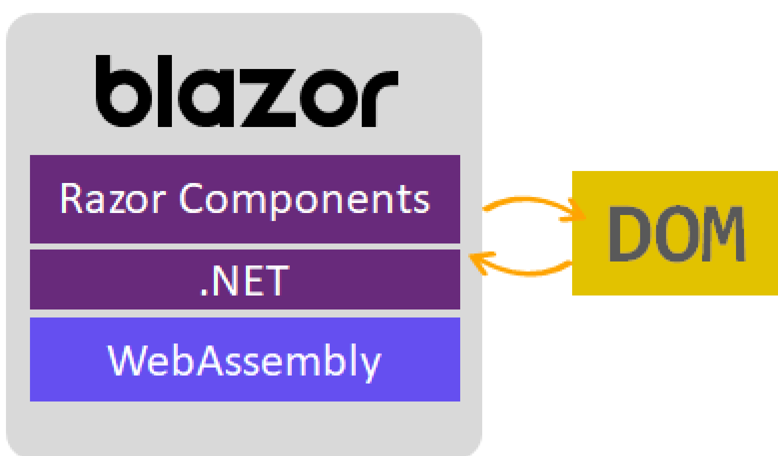
Vidíme, že kód se v podstatě dělí na sekci s html (obsahující c# proměnné pomocí Razor syntaxe - @). Zde zobrazujeme proměnnou *counter* obsahující počet aktuálních kliků na

tlačítko '*Click here*'. Jako jeho vlastnost připojíme obsluhu událostí *onClick* a přidáme delegát na naši funkci s názvem '*AddToCounter*'. Veškerou logiku poté píšeme do sekce *@code*, ve které deklarujeme proměnnou a vytvoříme tělo naší funkce. Tímto způsobem v Blazoru konstruujeme jednotlivé komponenty, které do sebe následně vnořujeme až například skončíme u takové, která reprezentuje celou jednu stránku. V další kapitole (č. 2.2) bude zmíněn jeden ze způsobů, kterým Blazor komponenty strukturujeme pro zvýšení čitelnosti (Atomic design).

Abychom mohli jednoduše vnořovat komponenty do sebe, musíme nějakým způsobem označit proměnnou, která bude sloužit jako parametr při volání dané komponenty jinou. V Blazoru používáme atribut **[Parameter]** umístěný nad proměnnou. *Templated* komponenty používáme, když komponenta přijímá jinou pro renderování – využívá se proměnná typu **RenderFragment** se jménem **ChildContent** [9]

1.2.2 Client side

Běh .NET kódu uvnitř webového prohlížeče je možný pomocí *WebAssembly*, která je v současnosti standardem a je podporována většinou moderních prohlížečů. Její pomocí můžeme vytvářet interaktivní client-side webové aplikace pomocí Blazoru. Rovněž má přístup ke všem funkcionalitám prohlížeče pomocí Javascript *interlop*. Na obrázku můžeme vidět zjednodušené schéma komunikace Blazoru s DOM pomocí *interlop* [9, 10].



Obrázek č.3 Blazor a DOM komunikace – client side [9]

Postup sestavení a spuštění aplikace je následující:

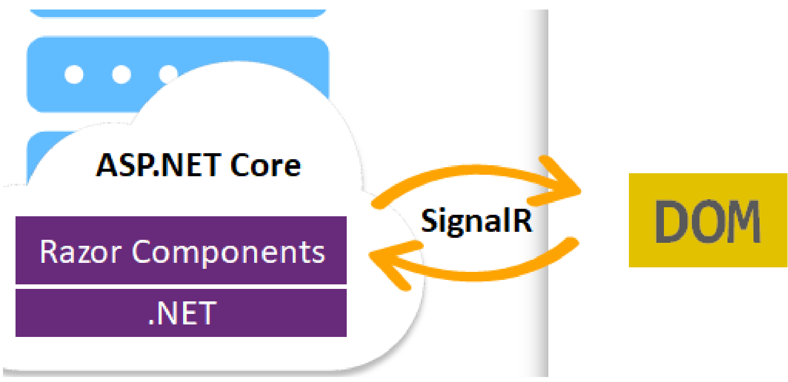
1. C# a Razor soubory jsou zkompilovány do .NET Assemblies

2. .NET Runtime a Assemblies jsou vloženy do prohlížeče (velikost výsledné aplikace je tedy velmi podstatná)
3. WebAssembly poté zavede běhové prostředí .NET a nakonfiguruje ho pro načtení a sestavení aplikace. Dále používá interlop pro DOM manipulaci a na volání API prohlížeče [9, 10].

Výhody tedy převážně spočívají v absenci serverové strany a JS (převážně). Nevýhodou je značná velikost aplikace, která se musí stáhnout do prohlížeče a nutnost WebAssemlby (který někde nemusí být podporován) [11].

1.2.3 Server side

Blazor umožňuje hostování Razor komponentů na straně serveru v ASP.NET Core aplikaci. Veškerá komunikace mezi klientem a serverem je poté zprostředkována *SignalR* připojením. To je také zodpovědné za komunikaci s Javascript interlop.



Obrázek č.4 Blazor a DOM komunikace – server side [9]

Běh aplikace je tedy převážně orientovaný na server, a kromě zpracování C# kódu jsou zde obslouženy UI události, aplikace jejich změn a zasílání zpět do prohlížeče [9, 10].

Výhodou je rovněž absence JS, malá velikost při načítání, přístup ke všem knihovnám .NET Core, debuggování a funkčnost na každém prohlížeči. Nevýhodou je poté nutnost serverové strany (nelze spustit off-line), vyšší latence a horší udržitelnost [11].

1.3 Bezpečnost v .NET

Klíčovými koncepty v případě zabezpečení jakékoliv aplikace je verifikace identity subjektu (autentizace) včetně procesu ověření, zda je subjekt oprávněn provést požadovanou akci (autorizace). Ověřit identitu subjektu můžeme v .NET pomocí rolí (role based), nebo pomocí nároků (claim based). Obě mají své výhody a nevýhody, které se dozvíme v kapitolách níže.

Uživatele, který má nějakou identitu a roli, nazýváme představitel (v .NET třída **PrincipalClaim**). Ten poté provádí akce, na které má práva, jménem uživatele. *Principal* třída tedy představuje bezpečnostní kontext daného uživatele a obsahuje jednu, nebo více jeho identit [12].

V .NET jsou tři druhy představitelů:

1. Generický, reprezentující uživatele a role nezávisle na Windows
2. Windows představitel, reprezentující Windows uživatele
3. Výchozí představitel, definovaný pro potřeby dané aplikace [12].

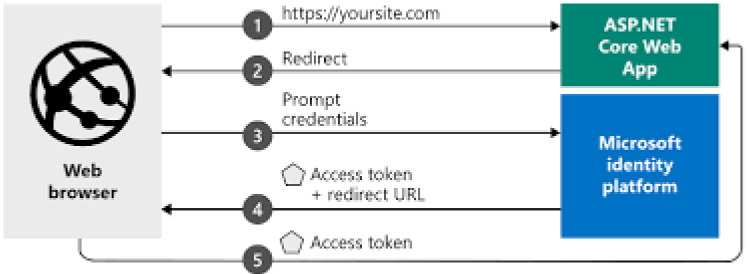
Autentizace v .NET je tedy ověření identity představitele (Principal) kontrolou přístupových údajů uživatele a validace vůči nějaké autoritě [12].

Autorizace poté nastává po autentizaci a využívá představitele pro získání informací o přístupu ke zdrojům pro daného uživatele (např. jaké koncové body může uživatel zavolat).

1.3.1 Autentizace

Jak už bylo několikrát zmíněno, jedná se o proces ověření identity uživatele. V .NET je autentizace řízená pomocí *IAuthenticationService.* Tato služba používá specifická autentizační schémata (cookies, tokeny apod.), které jsou na začátku nastaveny ve třídě *Startup* [13].

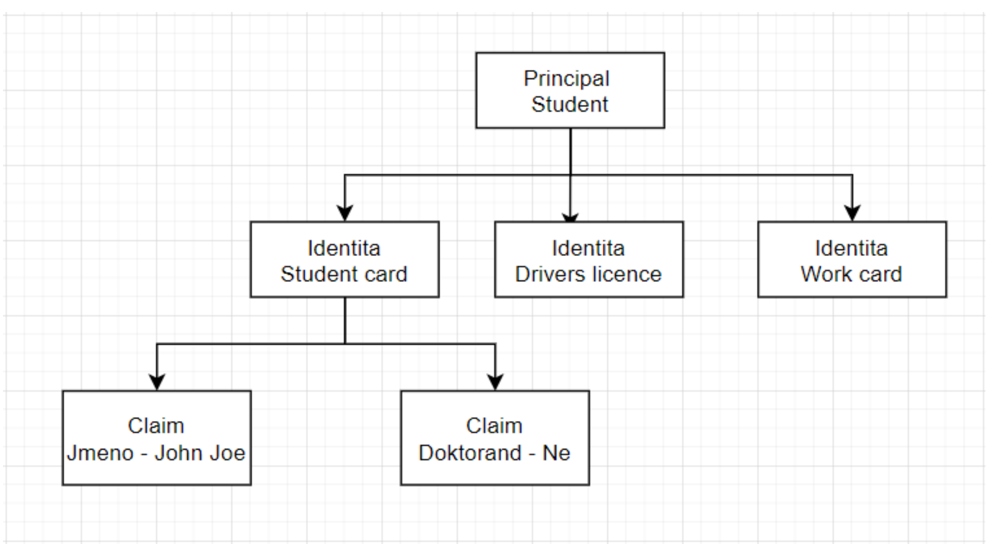
Proces autentizace můžeme v podstatě rozdělit na ověření identity proti dané autoritě, vygenerování bezpečnostního kontextu (v případě ověření identity) a na základě zvoleného autentizačního schématu je provedena autentizaci. Na obr. č.5 můžeme vidět celý proces.



Obr č.5 – Autentizace [14]

Autentizace je zodpovědná za poskytnutí třídy **ClaimsPrincipal** pro autorizaci [13]. *ClaimsPrincipal* třída se skládá z listu identit daného subjektu a každá identita se skládá z tzv. Claims. V realitě to může znamenat to, že např. student má více identit. Má kartičku studenta, řidičský průkaz, průkazku pro vstup do firmy. Na průkazkách má dále i klíč –

hodnota údaje, tedy Jméno – John Doe, Věk – 21 a další, to představuje Claims. Na jejich základě je poté provedena autorizace (například na studentské průkazce bude JeDoktorand – ne, přístup ke specifickému zdroji bude zamítnut). Na obrázku č.6 můžeme vidět ilustrační podobu ClaimPrincipal objekt.



Obr č.6 – ClaimsPrincipal hierarchie

1.3.2 Autorizace

Autorizace je ověření přístupových práv uživatele, které získáme z bezpečnostního kontextu po autentizaci. Autorizaci musíme volat až po autentizaci (v opačném případě by to nedávalo smysl). Nejčastěji je vynucena pomocí autorizačních atributů, které umisťujeme nad zabezpečené *Controllery*, popřípadě metody. Samozřejmostí je pak zavolání autorizační metody kdekoliv potřebujeme [15, 16].

Lze říct, že v .NET máme dva druhy autorizace. Přímočarou *role-based* a komplexnější *policy-based (v podstatě claims).* Když máme jasně definované role, které nevyžadují žádné komplexnější členění, je role-based ideální. V opačném případě je vhodnější policy-based. Zde můžeme velmi specificky nastavit, kdo má k danému zdroji přístup. Autorizace je tedy vyjádřena v požadavcích a ty jsou porovnávány oproti *user claims* (bezpečnostnímu kontextu) [15].

1.3.3 Identity API

.NET identity je API poskytující programátorovi správu uživatelů, hesel, profilových dat, rolí, nároků, tokenů, potvrzování mailů, dvoufázové ověřování a další (UserManager, SignInManager). Dále nám poskytuje UI pro přihlašování, odhlašování, registraci,

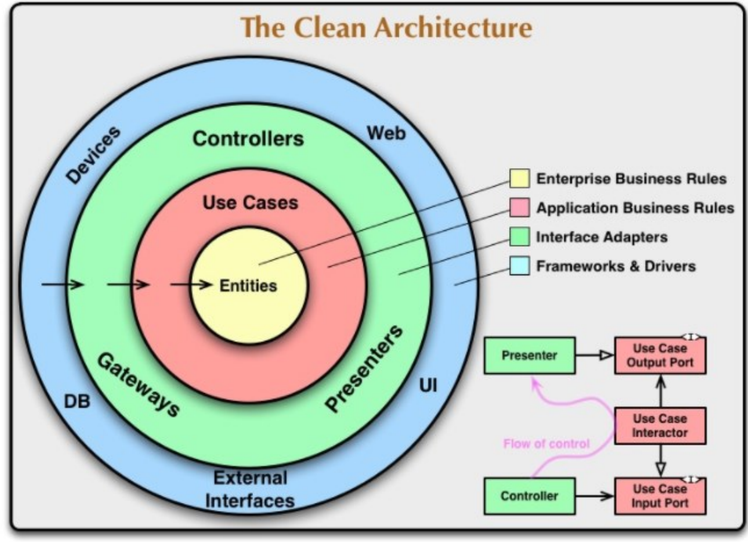
zapomenuté heslo, správu profilu a další. V podstatě nám zjednodušuje celý proces popsaný výše a autentizaci máme k dispozici hned po založení projektu (po vybrání příslušné nabídky), včetně nastavení ve SQL databázi. Informace o uživateli můžou být uloženy v databází pomocí Identity (např. s použitím *Entity framework*), popřípadě pomocí externího poskytovatele (Facebook, Google, Microsoft). Jediné, o co se programátor stará, je výměna generického UI (scafolding) a nastavení vlastních bezpečnostních pravidel [17].

2 ČISTÁ ARCHITEKTURA A KÓD

Strukturalizace kódu očekávaným způsobem velmi zvyšuje jeho čitelnost jak pro člověka, který kód psal (a vrací se k němu po několik měsících), tak i pro ostatní vývojáře, ať už členy teamu, nebo jiné. V této kapitole tedy bude popsán jeden ze způsobů, kterým můžeme projekt rozdělovat jak na BE části (*Onion* architektura), tak na té FE (atomický design). Kromě struktury je velmi podstatný samotný způsob psaní kódu. Jedna z metod, která nám pomůže psát čistší kód a zároveň eliminovat pravděpodobnost vzniku častých chyb, je využití funkcionálního paradigma, které C# přímo poskytuje přes některé novější jazykové konstrukty (C# je sám o sobě multiparadigmatický jazyk), a hlavně pomocí knihovny LINQ. V poslední části se zaměříme na jednotkové testy, jejichž důležitost zde ani nemusí být umocňována.

2.1 Čistá architektura

Čistá architektura, původně zmíněna v blogu Roberta C. Martina, je souhrnný název, pod kterým se nachází množství architektur se společným cílem. Tím je oddělení zodpovědnosti (seperation of concern). Toho je dosaženo rozdělením systému do několika vrstev (každá má svoji zodpovědnost). Výsledkem jsou systémy nezávislé na frameworku (např. není závislá na existenci nějaké knihovny), UI, databázi, a dalších. V podstatě naše business logika neví o okolním světě. Chceme například vyměnit UI za konzolovou aplikaci? Databázi ze SQL na MongoDB? Díky čisté architektuře to není problém, protože naše business logika není na pevno spojena s žádnou závislostí [18].



Obr č.7 – Čistá architektura [18]

Na obrázku č.7 můžeme vidět souhrnný obrázek jednotlivých vrstev. Velmi důležitým pravidlem je zde pravidlo závislostí – **The Dependenci Rule**

To říká, že směr závislostí by měl vždy mířit dovnitř. Tedy vnitřní kruh by neměl vidět nic z vnějšího kruhu. Dejme tomu, že ve vrstvě *Use Cases* vytvoříme veřejnou proměnnou. Ta za žádných okolností nesmí být použita ve vrstvě *Entities* [18].

Mezi specifické architektury, které dodržují principy výše patří – Haxagonal architektura, Onion architektura, Screaming architektura, DCI, BCE. Přesto, že se tyto architektury v detailech liší, jsou založeny na stejných myšlenkách, a dále je budeme vždy nazývat čistá architektura [18].

2.1.1 Popis jednotlivých vrstev

Entities

Reprezentují doménové objekty a zapouzdřují business logiku naší aplikace (popřípadě podobných aplikací v daném oboru, např. banka bude mít vždy celou sadu stejných entit). Obsahuje nejobecnější pravidla, které se prakticky nebudou měnit, a to i při změně externích závislostí [18, 19].

Use cases

Obsahuje business logiku specifickou pro danou aplikaci. Je to v podstatě to, co s aplikací budeme dělat a jsou zde implementovány všechny uživatelské požadavky. Od této vrstvy neočekáváme, že bude ovlivněna změnami externích závislostí, jako DB nebo UI [18, 19].

Adaptery rozhraní

Pomocí nich získáváme a ukládáme data z různých zdrojů (DB, sítě, souborový systém). Definujeme rozhraní a vrstva pro konkrétní zdroj dat jej implementuje (Use cases vrstva neví, z jakého místa data přicházejí, má pouze rozhraní) [18, 19].

Frameworky a ovladače

Vnější vrstva se skládá ze frameworků a knihoven, jako je například konkrétní typ databáze. Obecně zde není tolik kódu, primárně ten, který získá data z vnějšího zdroje a předá data vnitřní vrstvě [18, 19].

V praxi samozřejmě můžeme narazit na situaci, kdy nám pouze zmíněné vrstvy nestačí. Můžeme tedy přidat vrstvu podle potřeby, základním podmínkou je dodržení pravidla závislosti

2.1.2 Výhody a nevýhody

Každá architektura má pochopitelně své výhody a nevýhody. Co se hodí pro jeden typ aplikace se zákonitě nemusí platit pro jiný.

Mezi výhody patří:

Přehlednost

Případy použití jsou jasně viditelné v projektové struktuře. Člověk, který je s touto architekturou seznámený, bude rychleji produktivnější při přechodu na nový projekt. V projektu je zároveň přesně stanoven tok závislostí, takže program nebude skákat na neočekávaná místa [20].

Flexibilita

V případě potřeby změny libovolného frameworku nevzniká prakticky žádný problém, protože externí závislosti nejsou na pevno spojeny s business logikou. V podstatě stačí implementovat adaptér pro daný framework a přehodit jej pomocí DI [20].

Testovatelnost

Můžeme jednoduše plnit služby testovacími daty v unit testech (mocking).

Vhodná pro kombinaci s dalšími návrhovými vzory

Například SoC, IoC, MVP a další.

Mezi nevýhody patří:

Vyžaduje určitou disciplínu

Není například dobré vytvářet výjimky pro jednotlivé principy (například porušení závislostního pravidla) [20].

Učící křivka

Může být pro začátečníky velmi náročné na pochopení a zorientování.

Velké množství rozhraní

V případě větších projektů může počet rozhraní velmi vzrůst.

2.2 Atomický design

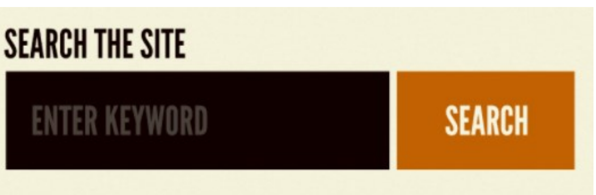
Jedná se o způsob, jakým můžeme přistupovat k designu webové stránky více metodickým způsobem. Základní myšlenka pochází z chemie, kdy všechna hmota je složena z atomů, ty tvoří molekuly, a ty zase organismy. Na konci poté máme veškerou hmotu ve vesmíru [21]. Analogicky v případě *Atomic designu* se naše stránka skládá z 5 různých vrstev:

Atomy

Mezi atomy můžeme zařadit komponenty, které už dále nejdou rozdělit do menších. Jedná se o HTML tagy, například tlačítka, vstupy, labely. Můžeme sem zařadit i více abstraktní elementy, jako barevné palety, fonty a jiné [21].

Molekuly

V případě že začneme kombinovat více atomů, vznikají molekuly (vytváříme separátní komponentu, která odkazuje na atomy). Zde můžeme například zkombinovat label, vstup a tlačítko pro vytvoření funkčního formuláře [21].



Obr č.8 – Molekula [21]

Základní myšlenkou je znovu používatelnost a kvalita při vytváření jednotlivých komponentů. To v budoucnu vede k lepší přehlednosti a ušetření času při vývoji.

Organismy

V případě organismů už začínáme být poměrně konkrétní a vytváříme jednotlivé části uživatelského rozhraní pomocí stávajících molekul. Můžeme si představit například hlavičku stránky obsahující logo, vyhledávač, tlačítka a sekci pro přihlášení. Atomy a molekuly jsou pro klienta poměrně nezajímavé, ale v případě organismů už to platit nemusí [21].

Šablony (templates)

Zde už se výtvor začíná velmi podobat výsledné verzi. Jedná se o skupinu organismů, která tvoří stránku. Pořád se ale jedná o šablony, takže tu nejsou reálná data [21].

Stránky

Je konkrétní instance šablony. Zástupný obsah je zde nahrazen reálným, který ve výsledku uvidí uživatel [21].

Atomicky design nám dává možnost postupovat při návrhu od velmi abstraktního po konkrétní. To přináší výhody jako znuvupoužitelnost, škálovatelnost a konzistenci [21].

V kapitole 1.2 bylo zmíněno, že základem Blazoru jsou komponenty, proto je velmi vhodný pro atomický design. Velmi dobře se v něm vytváří znovu používatelné komponenty, které jednoduše můžeme vnořovat do sebe.

2.3 Funkcionální programování v C#

C# je multiparadigmatický jazyk. Umožňuje se nám dívat na program pomocí různých pohledu či abstrakcí. To znamená, že můžeme řešit problémy jinými způsoby. V programování je vhodné použit správný nástroj na daný problém, což v konečném důsledků zajistí, že jej vyřešíme efektivněji, rychleji, popřípadě čitelněji. Z toho důvodu se tato kapitole věnuje funkcionální programování.

Funkcionální programování je deklarativní způsob programování založený na čistých matematických funkcích. K řešenému problému tedy nepřistupujeme jako k sérii kroků (imperativní přístup), ale spíše popisujeme to, co řešíme (tedy neříkáme jak, ale co). Místo příkazů používáme výrazy. FP je založeno na matematickém *Lambda calculus* frameworku a všechny funkcionální jazyky využívají myšlenky v něm obsažené. Vše, co lze spočítat v Lambda calculus, lze spočítat na *Turingově stroji* [22].

Dále budou rozebrány důležité koncepty funkcionálního programování, které se liší od ostatních paradigmat.

2.3.1 Základní principy FP

Asi nejdůležitějším konceptem jsou **čisté funkce**. To jsou funkce, které nemají žádné vedlejší účinky (side effects). Prostě na základě svých argumentů vrátí stejný výsledek vždy, nehledě na stavu systému. Tyto funkce nemění hodnotu svých argumentů a ani žádných globálních proměnných, neprovádí žádné vstupně výstupní operace, ani žádnou externí komunikaci (např. se sítí). Samozřejmě dříve, nebo později, je vždy nezbytné komunikovat s okolním světem, což se v FP řeší oddělením funkcí, které jsou čisté, a které provádí tuto komunikaci. Pokud naší aplikaci navrhneme dobře, její většina budou čisté funkce. Ty

většinou provádí nějakou transformaci s daty, nebo business logikou. Na hranicích systému poté voláme funkce, které např. zapíší do DB, nebo zavolají API [22, 23].

Dalším konceptem je **neměnnost (immutability)**. V čistě FP jazycích nemůžeme změnit hodnotu proměnné kromě prvotní inicializace. Můžeme vytvořit novou proměnnou a zde tuto hodnotu uložit, ale nikdy ne přepsat. To platí i pro objekty a další pokročilejší typy. Tento na první pohled omezující faktor umožňuje redukovat překvapivé vedlejší účinky, dále pracovat s více vlákny bez obvyklých problému (souběh a uváznutí) a zamezuje vzniku obtížně viditelných bugů (bugu, kde opravou je prohození pořadí proměnné) [22, 23].

Velmi důležité je rovněž použití **prvotřídních funkcí (first class funkce)**. V FP jazycích je funkce výraz, který můžeme uložit do proměnné, jako cokoliv jiného. Stejně tak funkce můžeme předávat jako parametr do jiných funkcí, které rovněž můžou funkci vracet jako návratovou hodnotu. Těmto funkcím se říká funkce vyššího řádu – *HOF* (Higher order function). Jedná se o funkce typu *Map, Filter, Reduce* a další, které se ukázaly jako velmi užitečné [22, 23].

Dále sem patří častější použití rekurze, rozpoznávání vzorů, rozšířený datový systém, koncepty jako *Functor, Applicative, Monad a Monoid*.

Vzhledem k prokazatelné účinnosti používání těchto principů začali moderní jazyky, jako C#, představovat jazykové konstrukty či knihovny, které podporují FP. V následujících kapitolách si řekneme o pár z nich.

2.3.2 Jazykové konstrukty podporující FP

C# podporuje poměrně velké množství FP konstruktů a knihoven, které s trochou disciplíny můžeme zavést do našeho projektu. Samozřejmě oproti čistokrevným FP jazykům, konstrukty jako čisté funkce, nebo neměnnost proměnných, nejsou vynuceny, ale např. pomocí *Resharper,* nebo jiných knihoven, můžou být detekovány při statické analýze kódu alespoň formou varování.

Jak tedy psát v C# čisté funkce?

Funkce by měla být krátká, deterministická, neovlivnitelná globální proměnnou (např. v kontextu třídy) či jinou externí logikou a nejlépe dělající pouze jednu věc. Výstupní hodnota by měla být spočítána **pouze** z funkčních argumentů, které nebudou modifikovány. [24].

Neměnnost proměnných v C# můžeme zajistit hned pomocí několika klíčových slov, jak můžeme vidět na obrázku níže.

```
1.  const int MAXITERATION = 10;
2.
3.  class Employee
4.  {
5.
6.      public readonly IEmployeeRepository _EmployeeRepository;
7.      public string Name { get; }
8.      public bool IsEmployeed { get; private set; }
9.  }
10.
```

Obr č.9 – Immutabilita v C# - 1

Na řádku 1 vidíme vytvoření konstanty, kterou už nikdy nebudeme moci změnit. O něco použitelnější jsou poté *readonly* proměnné, které jdou pouze inicializovat přímo, popřípadě z konstruktoru. Dále na řádku 7 se nachází proměnná pouze pro čtení (což je vynuceno smazáním *setteru* a ponecháním pouze *getteru*). Pod ní vidíme použití privátního setteru, což je od novějších verzí C# taky možnost. To už porušuje princip neměnnosti proměnných. Poslední novinkou jsou *init settery*, které umožnují inicializovat vlastnost objektu pomocí speciální syntaxe.

Co ale třídy? Můžeme nějakým lepším způsobem zajistit, aby vlastnosti objektu byly neměnné? V nových verzích jazyka C# můžeme použít klíčové slovo *Record* místo *Class* a speciální poziční syntaxi. V následující ukázce můžeme vidět použití [24].

```
1.  var empoyee = new Employee("John Doe", "doe@utb.cz");
2.  empoyee.name = "newName"; // ERROR
3.
4.  var student = new Student
5.  {
6.      Email = "",
7.      Name = "",
8.  };
9.  student.Email = "newMail"; // ERROR
10.
11. record Employee(string name, string email);
12.
13. record Student
14. {
15.     public string Name { get; init; }
16.     public string Email { get; init; }
17. }
18.
```

Obr č.10 – Immutabilita v C# - 2

Kód demonstruje použití *record* typu. Na řádku 11 vytváříme zaměstnance a pomocí poziční syntaxe mu udáváme jeho vlastnosti. Na řádku 1 poté vytváříme instanci tohoto objektu a na

řádku 2 se snažíme změnit vlastnost jméno, což samozřejmě nejde, protože proměnná je ve výchozím nastavení neměnná. Použití record typu má mnoho dalších výhod, mezi ně patří například chování při porovnávání. V případě třídy se porovnává reference, v případě record se porovnává hodnota. Rozdílů je více, ty už nejsou ale tolik relevantní pro FP. Na obr. č. 10 dále můžeme vidět alternativní zápis vlastností pouze pro čtení pomocí init syntaxe, která ním poskytne příjemnější inicializaci (řádek 4).

Dalším podstatným konstruktem je to, jak C# umožňuje pracovat s funkcemi jako hodnotami, tady ukládat je do proměnných, posílat jako parametry a vracet v návratovém typu. To je provedeno pomocí delegátů, typové bezpečných ukazatelů na funkce. Abychom vždy, když chceme definovat ukazatel na funkci nemuseli vytvářet delegát, byly do jazyka přidány dva delegáti – *Action* a *Func*, s prakticky libovolným množstvím argumentů [24]

```
1.  Func<int, int> multiply = a => a * a;
2.  Func<int, int, int> addTwoNumbers = (a, b) => a + b;
3.  Action<int> print = (a) => Console.WriteLine(a);
4.
5.  Func<int, int> TwoNumbersOperation(int a, Func<int, int, int> operation)
6.      => (b) => operation(a, b);
7.
8.  Console.WriteLine(TwoNumbersOperation(5, addTwoNumbers)(10)); // 15
9.
10. IEnumerable<T> Map<T>(List<T> data, Func<T, T> op)
11. {
12.     foreach (var item in data)
13.     {
14.         yield return op(item);
15.     }
16. }
17.
18. var lst = new List<int> {1, 2, 3, 4, 5 };
19. var newLst = Map(lst, (a) => a * 2); // 2 4 6 8 10
20.
```

Obr č.11 – HOF funkce

Na obr. č. 11 můžeme vidět, jak jednoduše lze v C# pracovat s funkcemi jako s hodnotami, předávat si je jako argument, provádět částečnou aplikaci a vytvořit si mapovací funkci (zde už se velmi blízko dostáváme implementaci LINQ knihovny).

Za zmínku také stojí rozpoznávání vzorů, *switch expression* a práce s *tuple* typem.

2.3.3 LINQ knihovna

Všechny koncepty výše byly použity pro návrh knihovny s názvem LINQ (Language-Integrated Query), FP knihovny pro práci s dotazy nad kolekcí dat (nad C# kolekcemi). V podstatě se deklarativním způsobem dotazujeme na data pomocí speciální syntaxe podobné SQL, popřípadě pomocí tečkové notace řetězíme rozšířené metody (extension methods). Motivací je rovněž jednotnost dotazování nad různými datovými zdroji. Nezáleží tedy, jestli dotaz provádíme nad polem, databází či XML souborem [25].

LINQ poskytuje velké množství *HOF* funkcí, které jsou známe z FP jazyků. Patří sem:

Select (Map) – procházení kolekce a mapování funkce na každý element

Where (Filter) – filtrování elementů z kolekce na základě podmínky

First, **Last**, **Any**, **All**, **Distinct**, a další.

Funkcí je opravdu velké množství, prakticky můžeme s daty provést libovolnou transformaci, poměrně v přehledné formě, bez většího časové postihu. Na následujícím obrázku můžeme vidět ukázku použití [25].

```
1.  IEnumerable<int> array = new List<int> {1, 2, 3, 4, 5};
2.
3.  var array1 = array.Select(a => a * 2); // 2 4 6 8 10
4.
5.  var array2 = array.Where(a => a % 2 == 0); // 2 4
6.
7.  var array3 = Enumerable.Range(1, 100)
8.      .Where(a => a % 2 == 0)
9.      .Select(a => a - 100)
10.     .OrderBy(a => -a)
11.     .First(); // 0
12.
```

Obr č.12 – LINQ kód

V první řádě na řádku 1 vytvoříme pole o 5 prvcích. Na řádku 3 poté voláme funkci *Select* nad tím polem a jako parametr předáme lambda funkci, kterou udáváme činnost. Z tohoto jednoduchého zápisu můžeme odvodit tři věci charakteristické pro FP. První je, že neudáváme, jak se má co stát, krok po kroku (neříkáme, že musíme iterovat přes pole a aplikovat na každý element funkci), ale deklarujeme, co se má stát (aplikuj tuto funkci na každý element). Dále vidíme, že předáváme lambda funkci jako parametr. Tato funkce nemá žádný vedlejší účinek, neovlivňuje jí žádný globální stav, je tedy čistá. Tyto jednorázové funkce jsou pro FP velmi charakteristické. V poslední řadě zachováváme princip neměnnosti, protože vždy když se zavolá téměř jakýkoliv LINQ dotaz, vytvoří se nová

kolekce, do které se hodnoty uloží. Pole s názvem *array* na řádku 1 zůstane tedy i na konci programu nezměněno. Na řádku 7 dále můžeme vidět, jak snadné je řetězit jednotlivé funkce a transformovat data.

2.4 Unit testy

Druh testování SW, kdy cílem je ověření, že každá jednotka kódu funguje podle očekávání. Provádí ho programátoři během vývoje aplikace. Unit testy izolují určitou část kódu a snaží se ověřit, že funguje tak jak má. Tato část může být funkce, třída, modul, objekt, aj. Jedná se o techniku bíle skříňky, znalost kódu je tedy nezbytná, proto je důležité, aby byla prováděna vývojářem (často ale v praxi QA dělá také unit testy). Unit testy by vždy měly být dodány s přírůstkem kódu (mělo by to být pravidlo). Vývojáři se často snaží ušetřit čas neděláním unit testů, ale to v konečném důsledku vede k vyšším nákladům při opravě bugů, které by mohl unit test snadno najít. Pokud jsou tedy unit testy dělány pravidelně, na začátku vývoje, pro každý přírůstek kódu, šetříme pak čas i peníze [26].

Jaké jsou tedy další klíčové důvody pro psaní jednotkových testů?

1. Pomáhají opravovat chyby v brzkém stádiu vývoje, tím šetří čas i peníze
2. Vývojáři jsou nuceni testovat svůj kód, často dojde k nalezení jiné chyby či situace, kdyby potencionální chyba mohla nastat (lepší pochopení)
3. Můžou sloužit jako projektová dokumentace
4. Pomáhají při znovuožívání kódu [26]

Nejčastěji se unit testy provádějí automatizovaně (jdou ale provést i manuálně). Často se využívá nějaký framework (např. UnitTest, Nunit, Xunit) a knihovny, které nám pomáhají s ověřováním (*Assert*) a s *mockováním* služeb [26].

V prvním kroku vytvoříme třídu (testovací sadu), do které poté umisťujeme jednotlivé testy (metody). Uvnitř testů poté používáme například AAA vzor (Arrange, Act, Assert).

Arrange – Příprava testovacích dat, mockování služeb

Act – Provedení testovacích funkcí

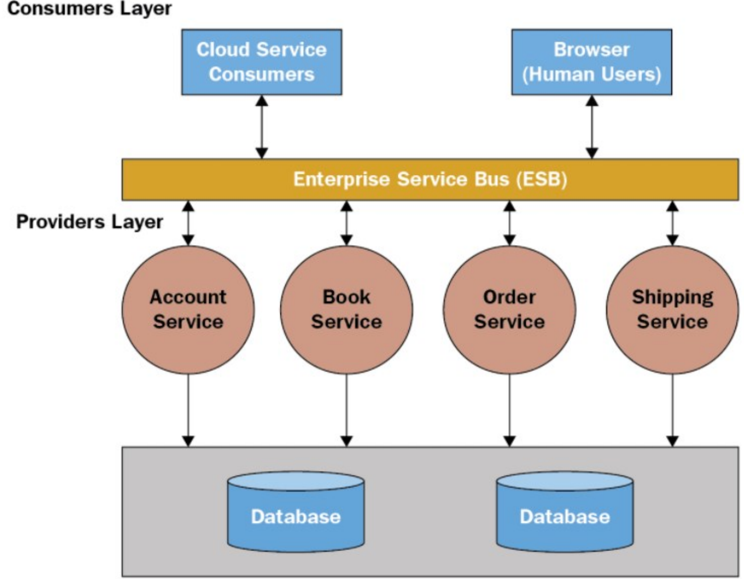
Assert – Ověření, jestli výstupní data odpovídají očekávaným.

3 SOA ARCHITEKTURA

Architektura orientovaná na služby (Service oriented architecture) je způsob, jak udělat SW komponenty více znovupoužitelné. Rovněž udává, jak zajistit jejich komunikaci pomocí servisních rozhraní. V praxi se setkáme se systémy, které můžou být velmi složité. Myšlenka tedy je, proč tyto druhy aplikací neposkládat ze skupiny na sobě nezávislých komponent poskytujících služby. Proč bychom například problémy jako autentizaci, navigaci a podobné museli řešit sami, když existují externí služby zpřístupněné pomocí standardního rozhraní (a komunikují přes síť). Tyto služby mohou být vytvořeny i jednou organizací v rámci interního systému [27].

Rozhraní pro služby poskytuje volné propojení (loose coupling), což znamená, že službu můžeme volat s prakticky žádnou znalostí o tom, jak je implementována. Jaké další věci jsou charakteristické pro SOA?

- SOA umožňuje uživateli kombinaci velkého množství funkcionalit ze stávajících služeb k vytvoření aplikací
- SOA poskytuje sadu návrhových vzorů a postupů pro strukturovaný vývoj systému a poskytuje prostředky k integraci většího množství služeb do koherentního decentralizovaného systému
- SOA balíčky spojují funkce do sady služeb, které můžeme integrovat do více SW systému [27, 28]



Obr č.13 – SOA architektura [29]

Na obr. č.13 můžeme vidět rozdělení architektury na dvě hlavní vrstvy, vrstva pro spotřebitele a vrstva pro poskytovatele.

Poskytovatelé služeb vytvářejí, udržují a poskytují jednu či více služeb k použití. Aby se o službě vědělo, zapíše ji do společného registru a zároveň poskytne informace o službě, jak s ní komunikovat (služební kontrakt), jak se používá a její případné zpoplatnění. Příkladem může být například Microsoft, který poskytuje služby jako *Azure cloud services* [28].

Důležitou součástí, která umožní větší znovu používatelnost komponent, je ESB (Enterprise service bus – servisní sběrnice). Jedná se o architektonický vzor, který slouží pro konektivitu – zasílání a odesílání zpráv, transformace datových modelů, směrování a jiné. ESB zpřístupňuje tyto operace jako servisní rozhraní pro opětovné použití novými aplikacemi. Samozřejmě by bylo možné nepoužít ESB, v tom případě by se aplikace musela připojit ke službě přímo, což ale vyžaduje více práce a v budoucnu to představuje problémy s údržbou. Proto v praxi SOA architekturu bez ESB téměř nikdy neuvidíme (tyto pojmy se proto mylně zaměňují) [27].

Spotřebitelé v registru najdou metadata o poskytované službě a vytvoří požadované klientské komponenty pro komunikaci se službou [28].

3.1 SOA principy

SOA je založena na několika základních myšlenkách. Patří sem:

Standardizovaná servisní smlouva

Co služba poskytuje? Jak s ní komunikovat? Všechny tyto informace najdeme v popisu těchto dokumentů, které každá společnost, nabízející nějakou službu, musí poskytnout [28].

Volné spojení (Loose coupling)

Služby jsou vytvořeny jako znovupoužitelné komponenty. Komponenty jsou na sobě závislé v co nejmenší možné míře [28].

Abstrakce

Veškerá logika a komplexita dané služby je skrytá za jasně definovaným komunikačním rozhraním – servisní smlouva, dokumentace. To, jak služba vnitřně funguje by pro spotřebitele mělo být irelevantní [28].

Znuvupoužitelnost

Designovat systém jako znovupoužitelné komponenty vede ke snížení vývojového času [28].

Autonomie

Služby mají kontrolu pouze nad logikou, kterou zapouzdřují. Z pohledu spotřebitele není potřeba vědět o implementaci [28].

Objevitelnost

Spotřebitelé, kteří hledají konkrétní službu, budu schopni pomocí jasně definovaných metadat ji nalézt. Tento bod by měl zaručit to, že dojde k efektivnímu využití služby jako zdroje třetích stran [27, 28].

Složitelnost

S použitím služeb (jako stavební kameny) můžeme vytvořit celé aplikace, které nám pomůžou dosáhnout našich obchodních cílů [28].

3.2 Monolitická architektura

Abychom dostatečně pochopili výhody použití SOA architektury, je nutné si krátce popsat, jak byla architektura převážně dělána doposud.

Monolitická architektura je považována za konvenční metodu vývoje SW. Výsledná aplikace je vyvinuta jako jeden balíček. Samotná aplikace je samozřejmě rozdělena na vrstvy v rámci daného řešení (např. hexadecimální architektura a jiné v rámci čisté architektury – viz. kapitola č.2). I když je samotná aplikace vytvořena v rámci jednotlivých vrstev, v konečném důsledku je zabalena do jednoho monolitu a nasazena. Z tohoto přístupu poté vyplývá několik vlastností charakteristických pro monolitickou architekturu, které u některých projektů můžeme označit za nevýhodné. Samozřejmostí zůstává ale fakt, že musíme vždy zvolit nejlepší architekturu pro konkrétní řešení a jsou případy, kdy je monolitická architektura více než vhodná [30].

Jaké jsou tedy základní charakteristické rysy pro monolitickou architekturu?

Nutnost použití jedné technologie

Obnáší poměrně velké množství omezení. Příkladem je migrace zastaralých částí aplikace na novější technologie či jazyky, které jsou například bezpečnější. To je v některých

případech téměř nemožné a často je lepší přepsat celou aplikaci. V případě SOA architektury může každá služba používat svoji vlastní technologii či jazyk [30].

Izolace poruch

V případě že selže jeden modul v monolitické architektuře, často to může způsobit řetězovou reakci ovlivňující fungování aplikace jako celku. V případě SOA či Microservice architektury (kapitola 3.3) přestane fungovat jen konkrétní služba, pokud není kritická pro chod aplikace, není ovlivněn uživatelský zážitek [30].

Velikost kódu

Asi není potřeba zmiňovat, že v případě monolitu je množství kódu enormní. Pro začátečníky může být čím dál těžší se v kódu zorientovat (stejně jako pro programátory, kteří program navrhli). V případě dodržení zásad SOA je množství kódu rovnoměrně rozděleno mezi služby [30].

Tento list by dále šel ještě rozšířit. V podstatě můžeme konstatovat, že monolitická architektura je vhodná pro malé a jednoduché aplikace.

3.3 Microservice architektura

SOA a Microservice (mikro služba) architektura jsou často zaměňovány, ale je mezi nimi pár zásadních rozdílů. V první řadě si zodpovězme na otázku, co to je?

Jedná se o modulární přístup k vývoji aplikací tak, že je vytvořena sada malých, nezávislých a autonomních modulů, které poskytují různé služby. Každá služba je samostatná a měla by implementovat jedno obchodní pravidlo (business rule). Jednotlivé funkce aplikace jsou tedy rozděleny do modulů, které komunikují pomocí API (API gateway). Mikro služby jsou nezávislé na technologii, jazyku, typů uložišť. Rovněž řeší velké množství nevýhod u větších projektů zmíněných například v kapitole 3.2. Velká aplikace, rozdělená na více malých samostatných jednotek, se mnohem lépe spravuje, než velký monolit [27, 30].

Microservice architektura se stala populární hlavně v poslední době, s příchodem virtualizace, cloud computingu, agilních praktik a Devops [27].

3.3.1 SOA vs Microservice architektura

Existuje poměrně velké množství článků, které se snaží srovnat tyto dvě architektury. Jeden z názorů uvádí, že Microservice architektura je podkategorie SOA, jiný že se jedná o další krok ve vývoji. V každém případě zde rozdíly jsou [27].

Základním je způsob, jakým se spojují komponenty a dále například rozsah použití. V případě SOA můžeme říct, že je to integrační architektonický vzor, který je více obecný pro velké množství aplikací (enterprise-scope). Umožňuje fungujícím aplikacím, aby byly zpřístupněny pomocí veřejného rozhraní, a jiné podniky tyto služby můžou využívat, a to i k vytváření nových aplikací [27].

Microservice architektura se zaměřuje spíše na to, jakým způsobem strukturovat samotnou aplikaci (application-scope) rozdělením na malé, nezávislé díly. Nedefinuje na rozdíl od SOA to, jak spolu jednotlivé aplikace mají komunikovat.

3.3.2 Porovnání jednotlivých architektur

Nevýhody monolitické architektury byly převážně zmíněny v kapitole 3.2. Tato kapitole se věnuje hlubšímu porovnání SOA a Microservice architektury z pohledu implementace.

Mezi hlavní rozdíly patří to, že SOA je ve své podstatě monolitická (jednotlivé služby mohou dosahovat obrovských rozměrů) a zaměřuje se na znuvupoužitelnost jednotlivých služeb. Ke komunikaci používá převážně ESB. Tyto služby jsou často dobře dokumentované (aby je třetí strany mohli využívat). Rovněž je designovaná pro sdílení zdrojů v rámci služby (často má pouze jednu vrstvu pro ukládání dat, kterou využívají všechny ostatní služby). SOA je výhodná pro rozsáhlé integrace (enterprise-scope) [31].

U mikro služeb má každá služba svoji vlastní dedikovanou databází, ve které má uložené pouze věci, které potřebuje ke svému fungování. Používají často komplexní API a jsou více zaměřeny na oddělení (autonomii). Na důležitosti zde nabývá používání Devops (CD, CI). Je vysoce škálovatelná, není limitována používáním nějaké konkrétní technologie, každá mikro služba může být napsána v jiném jazyku a zapouzdřuje jedno obchodní pravidlo [31].

V závěru je podstatné zmínit, že všechny typy architektur se v praxi často kombinují, protože ne vždy máme možnost začít psát celý projekt od začátku a navrhnout vše podle našich představ.

4 ZPROSTŘEDKOVATEL ZPRÁV (MB)

Jedná se o SW umožňující aplikacím, systémům a službám spolu komunikovat a vyměňovat si informace. Provádí to překladem mezi formálními protokoly zasílání zpráv. Umožňuje tedy nezávislým službám spolu komunikovat, i když byly vytvořeny pomocí jiné technologie na jiné platformě. Postup je poměrně jednoduchý, zdrojová aplikace (producer) zašle data do zprostředkovatele zpráv (message broker – MB). Zde se provádí seřazování, směrování, překlad, ukládání a doručení všem spotřebitelům, kteří o zprávu mají zájem. Zdrojová aplikace tím pádem není zodpovědná za doručení zprávy (což může být poměrně komplexní činnost), pouze za odeslání do zprostředkovatele zpráv. Tento přístup sebou nese množství výhod, například v případě, že aplikace přijímající zprávu je nefunkční, data jsou přesto zaslány do MB (který se postará o doručení při opětovné dostupnosti) a nedojde zde k selhání. Problém může být rovněž rychlost sítě, zabezpečení a fakt, že aplikace nemusí mluvit stejným jazykem [32, 33].

Zprostředkovatel zpráv je v podstatě *middleware*, umožňující standardizovaný způsob toku dat mezi aplikacemi, aby se mohly soustředit na svoji hlavní logiku. Může sloužit jako vrstva distribuované komunikace. Odesilatel zprávy neví, kde se nachází příjemce. Ví pouze kam má data poslat a služby, které mají o data zájem (a mají na ně právo), je odebírají (to posléze usnadňuje oddělení procesů v rámci systému) [32].

Jak už bylo nastíněno výše, jedna ze základních charakteristik MB musí být spolehlivé ukládání zpráv a garantované doručení. Toho je docíleno pomocí fronty zpráv (FIFO). Ta ukládá přijaté zprávy, dokud je cílová aplikace nezpracuje (v pořadí, v jakém přišli) a poté čeká na potvrzení. Tento proces můžeme označit jako asynchronní zasílaní zpráv a dovoluje systémům fungovat i v případě přerušovaného spojení a problému s internetem [32, 33].

Použití MB je rozsáhlé, od různých finančních systému (které musí zaručit zaslání platby pouze jednou), e-shopů, aplikace s vysoce citlivými daty (pomalejší, ale bezpečný přenos) a mnoho dalších.

4.1 Modely MB

Poskytuje dva způsoby distribuce zpráv.

Zasílání zpráv z bodu do bodu (Point-to-point)

Distribuční vzor používaný ve frontách zpráv se vztahem jedna ku jedné mezi odesilatelem a příjemcem. Každá zpráva zaslaná do fronty, je poslána pouze jednomu příjemci (je

konzumována pouze jednou). Point-to-point používáme tehdy, pokud chceme, aby daná zpráva byla zkonzumována jedním příjemcem. Ideálním příkladem jsou finanční transakce. Každá musí být zpracována pouze jednou [32].

Zasílaní zpráv Publikovat / Odebírat (Publish / Subscribe)

V tomto distribučním vzoru (často označovaném jako pub/sub), producent zašle zprávu do tzv. *topic*. Jednotlivý spotřebitelé poté začnou odebírat zprávy (subscribe) z topicu. Všechny zprávy do něj vložené budou distribuovány do všech aplikací, kteří je chtějí odebírat (broadcast-styl). Příkladem může být letiště, odesílající informace o aktuální statusu letů. Více stran, které tyto informace potřebují, je můžou odebírat a používat [32].

4.2 MB vs API

REST API (standardně používané např. pro komunikaci mezi službami), definuje sadu principů a omezení určené pro programátory při vytváření webové služby. Služby, které je dodržují, budou schopny komunikovat prostřednictvím operátorů a požadavků. API označuje kód, který pokud vyhovuje REST pravidlům, umožňuje službám spolu komunikovat. Používá http, protože je to standardní komunikační protokol na internetu. Ten je ale nejvhodnější na synchronní požadavky typu žádost – odezva. To znamená, že služba, která používá REST API musí být vytvořena tak, aby očekávala okamžitou odpověď. Pokud tedy klient, na kterého je poslán požadavek, bude neaktivní, odesílající klient bude blokován do momentu příchodu odpovědi [32].

MB naopak umožňuje asynchronní komunikaci mezi službami. Odesílající služba nemusí čekat na odpověď. To zlepšuje celkovou odolnost systému, nejen proti chybám. Dále usnadňuje škálování systému, protože MB typu pub/sub může snadno podporovat neomezený počet služeb [32].

4.3 ESB vs MB

V kapitole 3 bylo vysvětleno, co to je ESB. Jedná se o architektonický vzor používaný v SOA architektuře napříč organizacemi. V podstatě kombinuje komunikační a datové formáty do společného jazyka, který mohou sdílet všechny služby a aplikace (v dané architektuře). Může například překládat požadavky z XML do JSON. Dále poskytuje konektivitu, směrování a zpracování požadavků a mnoho dalších [32].

V realitě se tyto dva pojmy často zaměňují, ale existuje mezi nimi pár rozdílů. *ESB* infastruktura je velmi komplexní a tím pádem náročná na údržbu. V produkčním prostředí je často obtížné najít zdroj problému. Ten také nastává při škálování systému. *MB* jsou více odlehčené oproti ESB. Poskytují podobnou funkcionalitu (komunikaci mezi službami), ale jednodušeji a za menší cenu [32].

4.4 RabbitMQ

RabbitMQ je open-source MB poskytující veškerou sadu nástrojů a funkcionalit zmíněných výše. Jedná se o komplexní (ale jednoduchý) nástroj, který může být nasazen jak na cloudu, tak v rámci organizace (on premise). Podporuje velké množství protokolů zasílání zpráv (AMQP, MQTT, a další.). Je vysoce škálovatelný a dostupný (poskytuje integraci pro více jazyků). Je implementován pomocí *Erlangu OPT* (který je potřeba rovněž nainstalovat), technologii vyvinuté pro budování vysoce bezpečných a vysoce škálovatelných systémů [34, 35]

Co tedy dále nabízí RabbitMQ v porovnání s ostatními MB?

Vysoká spolehlivost

Zálohování dat, potvrzování doručení zpráv [36]

Flexibilní směrování

Zprávy jsou přenášeny přes tzv. *Exchanges* (jeden z druhů je právě již zmíněný topic) před tím, než se dostanou do front [36].

Shlukování (Clustering)

Více RabbitMQ serverů v lokální síti může být sloučeno do jednoho clusteru [36].

Vysoce dostupné fronty

Ty můžou být zrcadleny skrz více strojů v clusteru (v případě HW nehody) [36].

Podpora více protokolů a klientů

Podporuje téměř libovolný jazyk a protokol pro zasílání zpráv (dokumentace je rovněž velmi rozsáhlá i pro ostatní jazyky) [36].

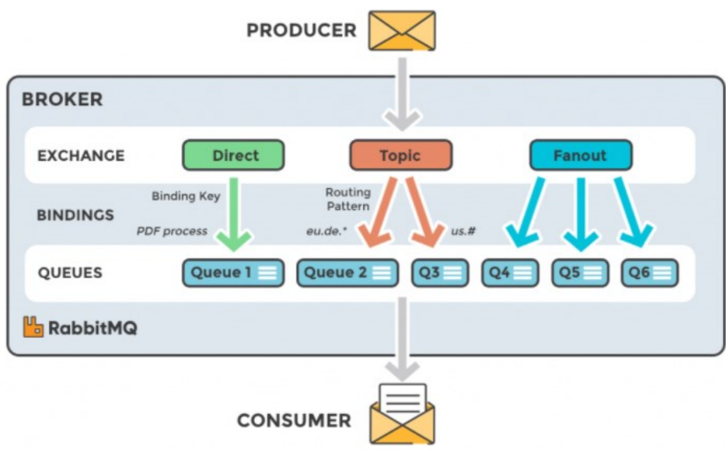
Management UI

Poskytuje velmi jednoduché UI pro správu RabbitMQ. Umožňuje nastavit (popřípadě monitorovat) prakticky každý aspekt MB (velmi vhodné i pro administrátory) [36].

Systémy pluginů, velká komunita, federační model a mnoho dalších.

4.4.1 Základní pojmy a principy

Následující obrázek shrnuje základní principy RabbitMQ.



Obr č.14 – Princip RabbitMQ [37]

První, čeho si na obrázku můžeme všimnout je producent zprávy, který ji zasílá do RabbitMQ. U něho je potřeba se chvíli zastavit. Aby producent úspěšně poslal zprávu, musí navázat spojení s běžící RabbitMQ službou (kde poskytne přihlašovací údaje, případně virtuální adresu kvůli autentizaci). Posléze vytvoří kanál, v rámci něhož se bude komunikovat. Producent posílá zprávu do některého druhu Exchange. Hlavní myšlenka RabbitMQ je ta, že producent nikdy nepošle data přímo do fronty (jak to může působit u **Direct exchange**). Dokonce ani neví, jestli data do nějaké fronty budou zaslána. Místo toho je posílá právě do Exchange. Jaké máme druhy Exchange [35, 38]?

Direct

Jedná se o druh point-to-point, který jsme popsali výše. Zpráva je doručena do fronty (často nemusíme udávat Exchange, v pozadí je stejně ale použit), kde vazební klíč je roven směrovacímu klíči (o tich bude zmínka níže) [35, 38].

Fanout

Některé zprávy potřebují být rozeslány do všech front a přesně k tomu můžeme použít tento typ Exchange. Každá služba v tomto případě připojí frontu do dané Exchange bez nutnosti specifikovat vazební klíč. Poté při příchodu zprávy tato Exchange automaticky rozešle zprávu všem frontám, které jsou na ni připojené. Fronty jsou často generovány automaticky odebírajícími klienty a mají náhodný název (*guid*) [35, 38].

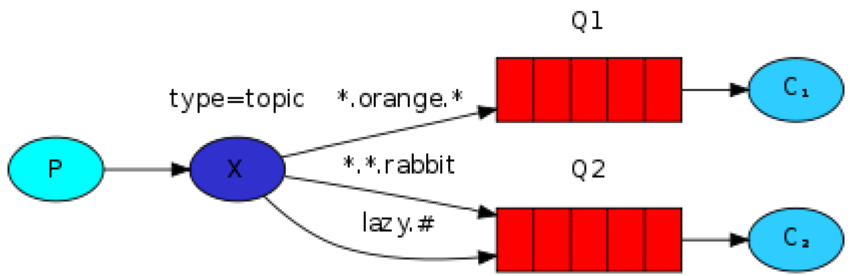
Topic

Asi nejuniverzálnějších ze všech zmíněných (které by bylo možno nahradit pomocí Topic). Poskytuje komplexnější filtrování zpráv z Exchange do jednotlivých front pomocí *wild-card* (opět porovnává dva druhy klíčů popsané v kapitole 4.4.2). To nám umožňuje psát komplexní logiku pro odesílání zpráv v našich službách [38].

4.4.2 Směrování a klíče

Směrování zpráv je velmi důležitá součást RabbitMQ. Aby systém fungoval správně a poskytl dostatečnou komplexnost v případě rozsáhlých systému, používají se pro směrování dva klíče. Směrovací klíč (*rounting key*) a vazební klíče (*binding key*). Směrovací klíč vkládá odesilatel do zprávy. Aby poté Exchange věděla, kam může a nemůže zprávu poslat, potřebuje vazební klíč. Ten obsahuje daná fronta připojená do Exchange. V případě že porovnání klíčů je úspěšné, zpráva může být poslána [38].

Každý druh Exchange poté přistupuje k porovnávaní klíčů jinak. V případě Direct Exchange je klíč jednoslovný. Například můžeme mít tři klíče (v případě zasílaní logů) – *Message*, *Warning*, *Error*. Zpráva s tímto klíčem bude zaslána do fronty se stejným klíčem. Fanout Exchange poté klíč ignoruje a zasílá zprávu do všech připojených front. Nejzajímavější je Topic, který umožňuje filtrování pomocí wild-card, jak vidíme na obrázku č. 15 [38].



Obr č.15 – Princip Topic exchange [38]

Jednotlivé části klíče oddělujeme pomocí tečky. V klíči dále můžeme použít hvězdičkovou nebo hashtag notaci.

‘*’ – nahrazuje přesně jedno slovo

‘#’ – nahrazuje 0 nebo více slov

Vždy si musíme tedy stanovit nějakou konvenci, na obrázku výše to je *rychlost.barva.druh*. Poslední fronta bude například brát všechny líná zvířata, první jen oranžová [38].

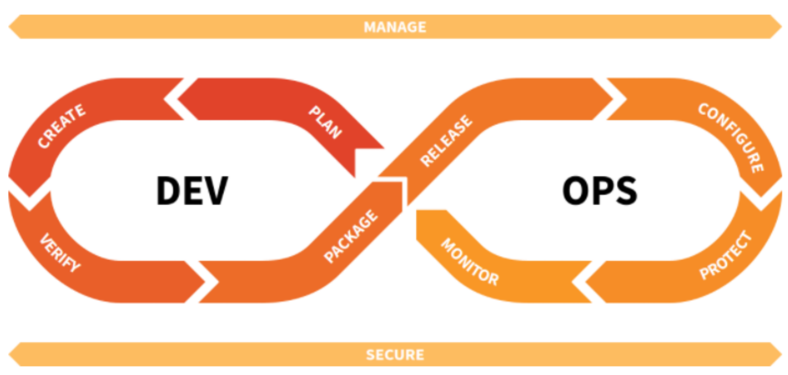
5 DEVOPS

Je to kombinace lidí, procesů, technologií a nástrojů pro zajištění průběžného doručování produktů s vysokou kvalitou pro zákazníka. Devops umožňuje organizacím lépe sloužit svým zákazníkům a v důsledku toho být na trhu úspěšnější. Tyto společnosti dokáží vyvíjet a vylepšovat produkt v mnohem rychlejším tempu než pomocí klasických konvenčních metod. Devops poskytuje izolovaným rolím (vývoj, IT, QA, a další.) techniky pro vzájemnou spolupráci a koordinaci. Vychází z agilního přístupu k vývoji a vyžaduje určitou změnu paradigmatu při vyvíjení SW [39].

Osvojení si Devops poskytuje velké množství výhod, patří sem primárně:

1. Zkrácení uvedení doby na trh
2. Přizpůsobení trhu a konkurenci
3. Zajištění spolehlivosti a stability systému
4. Zkrácení průběžné doby vývoje [39].

Devops můžeme rozdělit do několika fází, které zobrazujeme jako znak nekonečna (což vyjadřuje kontinuální, nepřetržitý cyklus).



Obr č.16 – Devops fáze diagram [40]

Plan – co je potřeba udělat, prioritizace a sledování jednotlivých částí

Create – psaní kódu, návrh změny a diskuze se spolupracovníky

Verify – automatické testování pro ověření korektnosti programu

Package – uložení aktuálního stavu SW do znovupoužitelné podoby

Secure – statická a dynamická analýza kódu a další druhy testů pro nalezení zranitelností

Release – nasazení SW pro koncové uživatele

Configure – správa infrastruktury a SW platformy

Monitor – logování, poskytování dat pro efektivní reagování na incidenty

Protect – zabezpečení infrastruktury, na které SW běží (aktualizace kontejnerů) [40].

Organizace, které nabízí Devops SW, často poskytují nějakou skupinu služeb v rámci jednoho balíčku. Ten obsahuje všechny nástroje pro jednotlivé fáze (například Azure Devops – kapitola 5.4) [40].

5.1 Základní principy a nástroje

Pro každou fázi a životní cyklus aplikace má Devops připravenou širokou škálu praktik a nástrojů.

Správa zdrojového kódu

Vzhledem k množství kódu a počtu lidí, kteří na něm pracují, je verzovací systém (podobný Gitu) velmi důležitou součástí. Cílem verzovacího systému je co nejlepším způsobem udržovat informace o změnách a dokumentovat postup při vývoji. Zároveň je to způsob, jak předejít řešení možných konfliktů při slučování kódu více lidí (*merge conflict*) [40].

Agilní správa projektů

Plánování, kontrolování, odhady, dokumentace práce a další činnosti v rámci agilního vývoje jsou opět velmi důležitou součástí u velkých projektů. Devops systém by měl poskytnout nástroj pro agilní způsob vývoje, jako vizualizace úkolů (task) a uživatelských požadavků (user stories), role, ceremonie a další [40].

Průběžná integrace (CI)

Ve zkratce se jedná o způsob, kterým můžeme automatizovat sestavování programu a testování okamžitě, po vložení nového přírůstku do našeho verzovacího systému (více o CI v kapitole 5.2) [40].

Průběžné doručování (CD)

Po provedení CI nyní můžeme vzít artefakt (výstupní soubor po sestavení, který můžeme nasadit na serveru), otestovat ho, zabalit, nakonfigurovat a nasadit do cílového prostředí. To vše rovněž automatizovaně (více kapitola 5.3) [40].

Zabezpečení Shift-Left

Způsob, jak identifikovat zranitelná místa během vývoje (ne až na konci, jak to v případě bezpečnosti často bývá) pomocí užitečných informací [40].

Monitorování, logování a okamžitá zpětná vazba

Devops často poskytují velmi komplexní systém logování (application insights), ze kterého můžeme vyčíst aktuální selhání, výkon aplikace v různých stádiích či v provozu, a další. V případě komplexního systému, ve kterém nastane nějaká neočekávána situace, můžeme pomocí logů často velmi rychle zjistit odpověď (místo snahy ji nasimulovat) [40].

Rychlá inovace a reakce na změnu

Rychlá zpětná vazba přes všechny teamy a také od koncových zákazníků či z produkčního prostředí [40].

5.2 Průběžná integrace (CI)

Jedná se o vývojářskou praktiku, při které dochází k integraci kódu do společného repositáře opakovaně, například i několikrát denně. Po každé integraci se v daném prostředí sestaví naše aplikace a spustí se testy. Výhodou tohoto přístupu je, že se okamžitě najdou chyby (typu na mém stroji to funguje), které vývojář může hned opravit. Jaké jsou tedy klíčové praktiky při nastavení průběžné integrace v projektu?

Důležité je používat jeden verzovací systém. Dále musíme automatizovat sestavení, k tomu využijeme nástroje v rámci našeho Devops frameworku. Toto sestavení by se mělo provést na nezávislém stroji a po něm můžeme spustit jednotkové testy. Kroky, které vedou k provedení celého procesu, nastavujeme v tzv. '*build pipeline*'. Na obrázku níže vidíme, jak může vypadat [41].

```
10  steps:
11
12    - task: DotNetCoreCLI@2
13      displayName: Run unit tests
14      inputs:
15        command: test
16        projects: "**/*Tests/*.csproj"
17        arguments: "--configuration $(buildConfiguration) --logger trx"
18
19    - task: PublishTestResults@2
20      displayName: Publish test results
21      condition: succeededOrFailed()
22      inputs:
23        testRunner: VSTest
24        testResultsFiles: "**/*.trx"
25
26    - script: dotnet publish ReviewMe.API/ReviewMe.API.csproj -o $(Build.ArtifactStagingDirectory)
27      displayName: "dotnet publish ReviewMe.API"
28
29    - task: PublishBuildArtifacts@1
30      displayName: "Publish Artifact: drop"
31      inputs:
32        PathtoPublish: "$(build.artifactstagingdirectory)"
```

Obr č.17 – CI – build pipeline

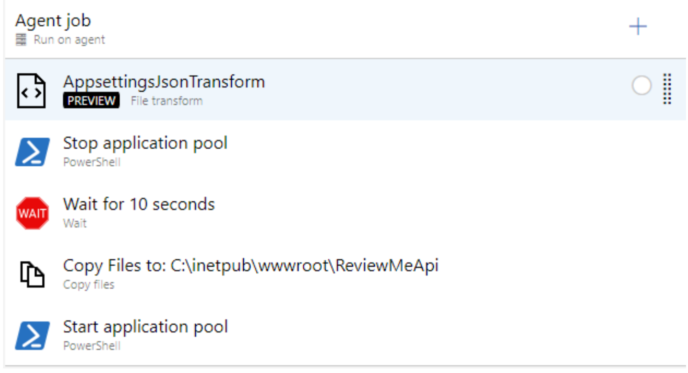
Na obrázku č.17 vidíme *build pipeline* v Azure Devops, která v jiných službách vypadá velmi podobně. Prakticky se jedná o sérii kroků, ve kterých říkáme, co se má stát (na začátku probíhá nastavení – cesta k projektu v repositáři). Na řádku 12 se v první řadě pouští jednotkové testy, které jsou v dalším kroku zveřejněny. Poté voláme příkaz ‘*dotnet publish*’, který sestaví aplikaci a připraví ji jako balíček pro hostovací systém. V posledním kroku vygenerujeme tak zvaný **artefakt** – což je výstupní soubor obsahující sestavený program včetně všech externích závislostí a nástrojů [41].

Jednotlivé kroky pak můžeme rozšířit o další operace, které nám můžou pomoc při vývoji, testování a detekování nedokonalostí. Příkladem může být statická kontrola kódu (například pomocí *Resharper* rozšíření) a podobné.

5.3 Průběžné doručování (CD)

Jedná se o způsob, jak nové přírůstky (opravy bugů, nové funkce, aj.) dostat do produkce prakticky okamžitě, po vložení do verzovacího systému, a to bezpečně, rychle a udržitelně. Po tom, co se provede CI a je vygenerován artefakt, ho můžeme použít k automatickému nasazení na server. Nasazení se rovněž provádí v několika krocích, které popisujeme v ‘*release pipeline*‘. Cílem je udělat z nasazování předvídatelnou, rutinní záležitost, která se provádí po jakékoliv změně (ať už velké, nebo menší). Toho docílíme právě tím, že kód bude vždy v nasazeném stavu, bez ohledu na počet vývojářů provádějících změnu [42].

To vše samozřejmě přináší značné množství výhod, mezi základní patří přírůstky s nízkým rizikem, rychlejší uvedení na trh, vyšší kvalita, nižší ceny a spokojenější team.

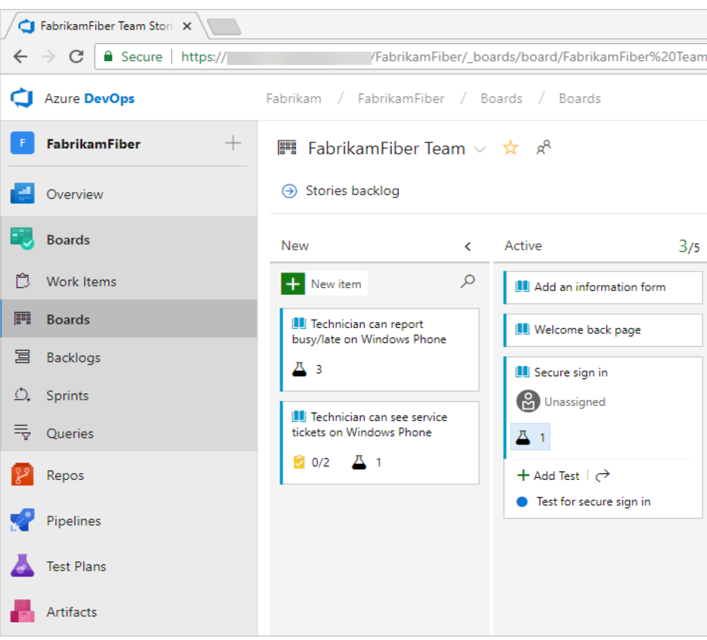


Obr č.18 – CD release pipeline

Na obr. č.18 můžeme vidět jednotlivé kroky v pipeline. Jedná se v podstatě o stažení artefaktu, zastavení aplikačního pool (v případě že by aplikace už běžela), a zkopírování aktuální verze aplikace na server. Poté už stačí jen aplikaci nastartovat.

5.4 Azure Devops

Azure Devops poskytuje vývojářům všechny již zmíněné služby a nástroje pro práci v Devops teamu. V podstatě se jedná o sadu komplexních nástrojů, které umožní společnosti profesionálně vytvářet SW. Samozřejmě se jedná pouze o alternativu, spousty dalších společností (Github, Gitlab, Jenkins, a další.) poskytují podobnou sadu nástrojů. Azure je však zralá, rychle se rozvíjející a vyladěná technologie, která se velmi dobře používá díky rozsáhlému webovému uživatelskému rozhraní. Mezi další výhody patří automatické aktualizace, dobrá cena, certifikace (např. z hlediska bezpečnosti), přístupnost, vysoká doba provozuschopnosti (garance až 99.9 % času), přístup k velkému množství služeb na cloudu (application insights, office) a mnoho dalších [43].



Obr č.19 – Azure Devops [44]

Na obr. číslo 19 můžeme vidět webovou stránku Azure Devops. V levé části obrázku se nachází vertikální menu. V něm máme přístup k veškerým nástrojům. První položka **Overview** obsahuje obecné informace o projektu, dokumentaci a podobné. Následuje **Boards**, obsahující sadu agilních nástrojů pro podporu plánování, vizualizace (kanban tabule) a sledování postupu. Další v řadě je **Repos** poskytující Git repositář (verzovací systém) pro kód teamu. Poté následuje **Pipelines**, poskytující CI a CD, tedy automatické sestavení a nasazení projektu. Předposlední je položka **Test Plans**, s nástroji pro testování naší aplikace a poslední **Artifacts**, pro sdílení balíčků (např. *NuGet*) a jejich integrace do naších pipeline. Nachází se zde samozřejmě mnohem více nastavení, přístupných například z *Azure Portal*.

II. PRAKTICKÁ ČÁST

6 POŽADAVKY NA APLIKACI

Požadavků na ReviewMe bylo poměrně velké množství. Netýkaly se ale jen samotné aplikace, ale i jejího začlenění do firemního systému, včetně aktualizace a zasílaní dat. Tyto požadavky byly nejčastěji konzultovány s lidmi z praxe, kteří by s podobnou aplikací pracovali. Byla zde rovněž snaha najít aplikace podobného rázu, ale neúspěšně (kapitola 6.4). Subjektivně jsem při návrhu zamýšlel, ať lidé nepřistupují k ohodnocení negativně, popřípadě se mu vyhýbají. V případě PR můžou kolegům, kteří jsou opravdu dobří pomoci (což může například vést i k lepšímu platu). Naopak u kolegů, se kterými se nedá spolupracovat a ani domluvit, může být ohodnocení horší.

6.1 Funkcionální a nefunkcionální požadavky

Uživatelské požadavky (User stories) si tedy v této kapitole rozdělíme do třech skupin podle rolí. Požadavky z pohledu **zaměstnance**, požadavky z pohledu **administrátora** a celkové požadavky **aplikace**. Patří sem jak funkcionální, tak nefunkcionální požadavky.

6.1.1 Zaměstnanec

Jak tedy bude standardní zaměstnanec pracovat s ReviewMe? Pro zapsání požadavků budeme používat schéma – **Jako uživatel chci**

Notifikace

Jako uživatel chci být informován (přes email) o tom, koho mám hodnotit, do kdy ho mám hodnotit a kde ho mám hodnotit. Aby uživatelé nezapomínali posílat ohodnocení, je dobré jim to nějakým způsobem připomenout. Email bude tedy vstupní bod, ze kterého často uživatelé vstupují do aplikace.

Proces hodnocení člověka

Jako uživatel chci mít přehledný, jednoduchý a rychlý způsob, kterým ohodnotím daného zaměstnance. Vhodná bude textová oblast s předpřipravenými sekcemi pro jednotlivé oblasti (technické či komunikační schopnost, a další).

Odložení rozepsaného ohodnocení

Jako uživatel chci mít možnost odložit rozepsané ohodnocení (uložit návrh) a vrátit se k němu později.

Přehledná forma v tabulce pro jednotlivé ohodnocení

Jako uživatel chci mít možnost zobrazit si přehledně všechny úkoly pro ohodnocení. Tato tabulka bude obsahovat fotku, jméno, pozici, datum (do kdy) a jednotlivé operace (provedení ohodnocení, odmítnutí ohodnocení).

Odmítnutí ohodnocení

Jako uživatel chci mít možnost odmítnou ohodnocení v případě, že s kolegou přímo nepracuji, neznám ho, nebo ho nechci hodnotit z jiného důvodu. Podmínkou je konkrétní uvedení důvodu, abychom předešli odmítání ohodnocení z lenosti. Tato možnost je poté nabídnuta i v notifikaci.

Historie všech ohodnocení

Jako uživatel chci vidět tabulku všech ohodnocení s informacemi, jestli jsem úspěšně ohodnotil, odmítl, popřípadě vypršel čas do kdy jsem ohodnotit mohl.

6.1.2 Administrátor

V případě ReviewMe byla velmi kritická práce administrátora. Ten v ní i pravděpodobně bude trávit více času. Administrátorem je v případě aplikace často myšlena pozice HR (popřípadě jiné, které se zabývají lidmi ve firmě). Administrátor, kromě toho, že je rovněž zaměstnanec (a bude také ohodnocen), má přístupnou kompletně novou sekci pro správu ohodnocení. Jaké jsou tedy požadavky?

Admin sekce se zaměstnanci

Jako administrátor chci mít sekci pro zobrazení všech zaměstnanců, jejich jména, pozice, lokace, datum ohodnocení, datum PR a operační tlačítko (pro vytvoření ohodnocení, popřípadě editace).

Filtrování zaměstnanců

Jako administrátor chci v sekci pro zaměstnance být schopen vyfiltrovat specifického zaměstnance podle různých vlastností (jméno, pozice, lokace, datum PR, stavu ohodnocení).

Stránka pro vytváření a modifikaci ohodnocení (assessment)

Jako administrátor chci sekci pro vytváření ohodnocení pro daného zaměstnance (přístupnou z tabulky zaměstnanců). Tato sekce bude obsahovat další informace o zaměstnanci, formulář pro dva datumy – PR datum, a datum do kdy se mají sesbírat ohodnocení. Dále zde bude sekce se všemi kolegy daného zaměstnance, které můžu vybrat pro ohodnocení (dojde jim

notifikace). V poslední řadě potřebuji ovládací panel pro vytvoření / zrušení / smazání ohodnocení.

6.1.3 Aplikace v systému

Mnoho důležitých rozhodnutí bylo potřeba učinit i v případě zasazení samotné aplikace do celkového systému (či architektury). Tato část se stala poměrně náročná, protože cílem práce bylo vytvořit precedent pro případné další aplikace (na základě aktuálních technologií a know-how), které se poté snadno do systému napojí (ať už nové, nebo již existující). Zároveň bylo potřeba neovlivnit stávající uživatelský zážitek již existujících aplikací, které se do systému časem vloží. Jaké jsou tedy požadavky pro aplikaci?

Použít FE aplikaci jako modul

Jedním z požadavků pro aplikaci bylo, aby šla vložit do jiné aplikace jako modul. Například firma může mít centrální aplikaci sdružující sadu dalších interních aplikací. Ty poté budou uvnitř vloženy a vykresleny jako modul (nedojde k přesměrování).

Jednoduchý FE

Dalším požadavkem bylo, aby FE část aplikace byla co nejjednodušší a prakticky sloužila pouze pro zobrazení výsledku z BE Api. BE a FE aplikace jsou tedy nasazeny zvlášť a FE část by mohla být lehce nahrazena.

Napojení do SOA architektury

Důležitým bodem bylo vložení ReviewMe do SOA architektury. Ta už může v podniku existovat, ale počítá se i s možností, že ReviewMe bude první aplikace v systému, a proto stanoví určitá pravidla a konvence. Tento krok by měl zajistit jednotný a efektivní způsob pro zpracování a tok dat mezi jednotlivými službami.

Využívat mikro služby kde je to možné (implementace EmployeeMicroservice)

Podnikové aplikace jsou často ideálním kandidátem pro vytváření mikro služeb, zapouzdřující funkcionalitu, která by se často opakovala v rámci systému. V případě této architektury je vhodným kandidátem mikro služba pro rozesílání dat z externího zdroje. Zároveň byla snaha vzít to nejlepší z obou architektur (mikro služby a SOA) při vytváření všech komponent. Například ReviewMe či mikro služba si drží pouze data, které nezbytně potřebují.

Centrální zprostředkovatel zpráv

Mnoho firem řeší situaci, kdy používá portál nějaké externí firmy pro správu zaměstnanců. Tyto data poté potřebují služby v rámci firemního systému. Aby nedocházelo k duplicitnímu ukládání dat, neoprávněnému přístupu či neefektivnímu toku dat, je vhodné v rámci architektury použít systém pro bezpečné rozesílání dat na správné místo.

Bezpečnost dat

Vzhledem k tomu, že práce s daty je v rámci systému a aplikace velmi důležitá, je potřeba zajistit jejich ochranu. Tato ochrana musí dodržovat pravidla GDPR.

6.2 Rešerše podobných aplikací

Před zahájením práce byl proveden průzkum ohledně podobných aplikací, jak na vnitrostátním, tak zahraničním trhu. V prvním případě bylo nalezeno několik firem, nabízejících elektronické hodnocení zaměstnanců. Tyto služby ale neposkytují ucelený systém zprávy ohodnocení ze strany administrátora (PR), notifikace na míru a jednotnou distribuci dat. Jedná se spíše o zhotovení formulářů pro ohodnocení na míru, úsporu administrativní práce s tím spojenou a následně vyhodnocení, včetně rad. V ReviewMe je hlavním cílem při vyplňování jednoduchost (není potřeba rozsáhlé formuláře) a dále automatizace celého systému díky znalosti dat interní sítě (kdo je kolega s kým). Tím je na mnoha místech eliminován lidský faktor. Jedná se tedy o služby, které by mohly být užitečné, ale ne jako náhrada za ReviewMe. To v poslední řadě přináší jednotný a bezpečný způsob asynchronní distribuce dat v rámci systému.

Zahraniční firmy nabízí podobné aplikace zmíněné výše a často se zde používají psané formuláře. Novým typem jsou webové služby, které jsou přímo zaměřeny na PR. Poskytují komplexní nástroje pro hodnocení a výpočet statistik. Kromě toho nabízí stovky dalších nástrojů pro zpětnou vazbu, poznámky, cíle, slabiny a silné stránky. Jedná se o externí nástroje, které by rovněž mohly koexistovat s ReviewMe. Opět to ale není náhrada za službu, kterou je možné umístit do firemní sítě s využitím interních dat.

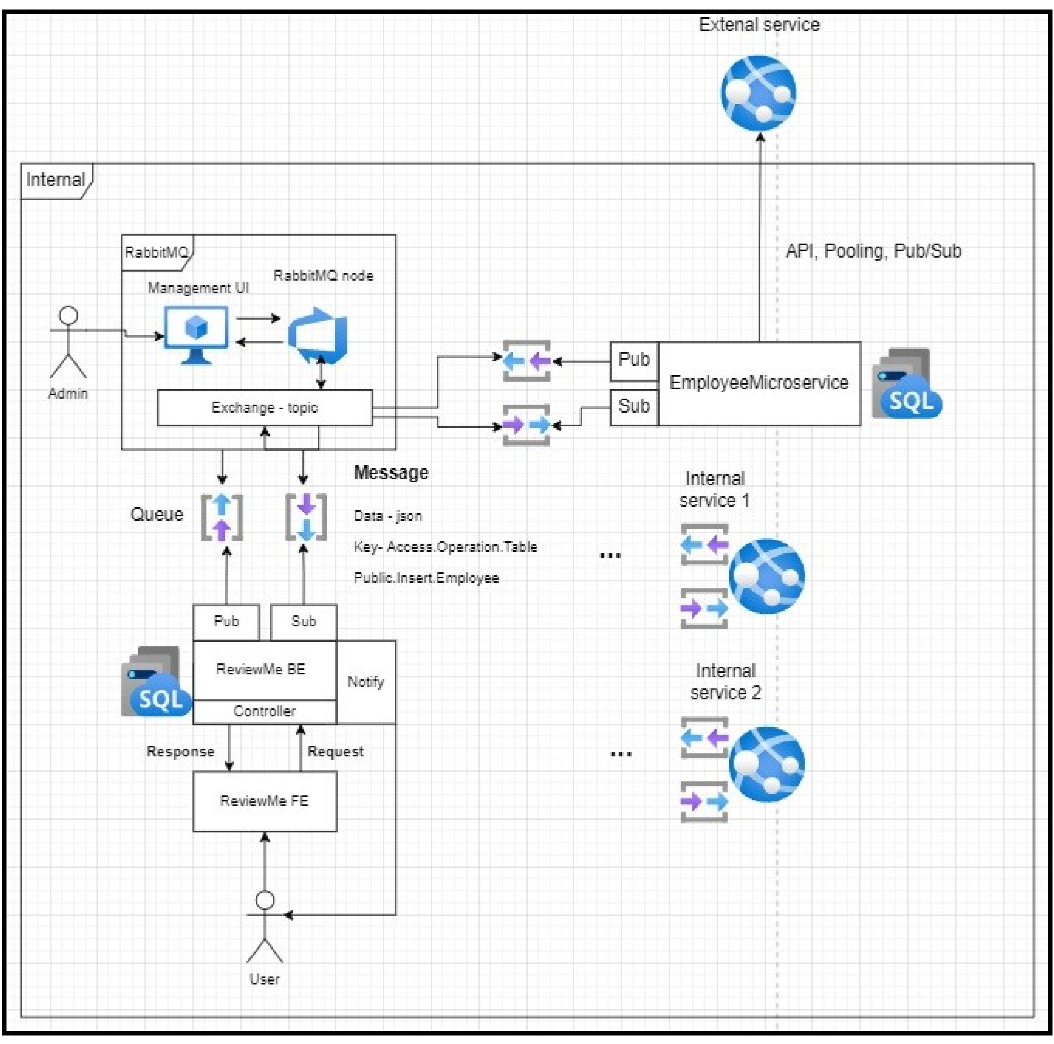
ReviewMe tedy kromě zaslání zpětné vazby (což bylo předmětem nalezených služeb) poskytuje mnoho dalších funkcí, zabalených v rámci jednoho balíčku, s jednoduchou integrací do stávajícího systému.

7 NÁVRH ARCHITEKTURY

Tato kapitola se věnuje nejen návrhu architektury pro samotné ReviewMe, ale pro celkový systém, ve kterém se bude nacházet. Patří sem tedy nejen mikro služba, která bude data získávat a posílat, ale i zprostředkovatel zpráv, rozesílající data dále do systému. Jednotlivé rozhodnutí byla snaha zdůvodnit a vybrané technologie jsou poté popsány v teoretické části této práce. Jak už bylo zmíněno v kapitole 6, mnoho rozhodnutí bylo ovlivněno požadavky získaných od lidí z praxe a mnoho jich vycházelo z limitací z již existujícího systému. Mnoho systémů v praxi (zvláště u menších firem) není navrženo ideálním způsobem (zejména v rámci komunikace a toku dat u více služeb), proto podobné limitující faktory byly spíše přínosem.

7.1 Schéma architektury

Na obrázku číslo 20 vidíme navržené schéma.



Obr č.20 – Schéma celkové architektury

Ze schématu je patrné, že komunikace primárně probíhá pomocí pub/sub (viz. kapitola č.4) zasílání zpráv. V rámci systému se tedy jedná o asynchronní komunikaci. Ta je přesto velmi rychlá (v řádech milisekund). RabbitMQ umožňuje pomocí RPC i synchronní komunikaci, která je využita zejména pro synchronizaci dat mezi službami v případě výpadku. Synchronní komunikace pomocí REST API je použita jen tam, kde je potřeba okamžité odezvi mezi uživatelem a serverem (např. mezi ReviewMe FE a BE) [45].

ReviewMe část

Jako první se zaměříme na ReviewMe část. Dle požadavků bude BE a FE část aplikace nasazena zvlášť. FE bude jednoduchá a nezávislá na BE (a také velmi snadno nahraditelná). Hlavní středobodem tedy bude BE část (na obr. č.20 vlevo dole). Jedná se o znovupoužitelnou a autonomní službu zveřejňující jednoduché REST API. Cílem je, aby tato služba udržovala data, která nezbytně potřebuje k chodu. V případě výpadku externího zdroje dat, na nějakou dobu, nebude ovlivněna a může fungovat nadále. Služba je rovněž nezávislá na poskytovali databáze, notifikací a zprostředkovali zpráv (takže je možné je velmi snadno nahradit, viz. kapitola č.2). Jádro aplikace tedy bude použitelné i v případě značných změn. Jak už bylo zmíněno, součástí je i systém notifikací, informující uživatele pomocí emailu a dále vrstva pro zpracování a zasílání zpráv do systému s použitím RabbitMQ. Více o podobě zpráv se dozvíme v kapitole č. 7.3. V příloze (P 2) je umístěn ER diagram pro ReviewMe se všemi entitami a vztahy.

EmployeeMicroserice část

Abychom využili předností SOA a Microservice architektury (a dále nezvětšovali ReviewMe), další součástí bude nezávislá mikro služba (vpravo nahoře), s jasně daným cílem. Získávat data z externího zdroje a uložit je k sobě do databáze. Dále veškeré aktualizace vzít, označit (bezpečnostní kontext) a publikovat je do zprostředkovatele zpráv. Vidíme, že se bude jednat o opravdu jednoduchou mikro službu, která má podle definice pouze jeden reálný účel a to distribuci dat. Externí data budou získávány buď přes API, *polling* z databáze, popřípadě další pub/sub (to už je ovlivněno poskytovatelem). Mikro služba má rovněž svoji vlastní databázi a v podstatě je nezávislá. Nyní už můžeme vidět jednu z výhod tohoto rozhodnutí. V případě, že externí služba bude nedostupná, mikro služba může nadále vracet data, které budou nějakou chvíli aktuální.

RabbitMQ část

Velmi důležitou součástí zajištující správnou distribuci dat je právě RabbitMQ část (vlevo nahoře). Jedná se o službu (RabbitMQ Node) nasazenou v interní síti. Tu můžeme ovládat pomocí příkazové řádky, popřípadě pomocí UI pluginu. Zde také vytváříme veškeré Exchange, fronty, virtuální hosty, opatření, uživatele a další. Dalším nezbytným nastavením jsou přístupy. Pro každého uživatele (aplikaci zveřejňující data do RabbitMQ) nastavujeme virtuálního hosta a práva na změnu, zápis a čtení (zapisujeme v podobě regex). Další úroveň ochrany pak poskytují tak zvané *topic oprávnění*. Ty umožňují provádět autorizaci pomocí Topic Exchange na základě obou klíčů. Dále vidíme, že vždy při komunikaci se službou máme dvě fronty, jednu pro publikování dat ze služby do systému a druhou pro odebírání dat ze systému do služby [46].

Další služby

Systém je navržen tak, aby byl vysoce škálovatelný. Není tedy problém připojit libovolné množství dalších služeb obdobným způsobem. V praxi dokonce můžeme zkopírovat část pro Pub/Sub u každé mikro služby, a pouze upravit typ dat, které se budou posílat či ukládat. Topic Exchange je navržen, aby zvládal obrovský provoz s velkým množstvím služeb. Vždy však záleží na konkrétní situaci a do budoucna není problém přidat další Exchange.

7.2 Výběr technologií

Jeden ze základních cílů architektury výše je nezávislost služeb v systému na technologii. Je v podstatě jedno, jestli je naše služba vytvořena v C#, C++ či Javascriptu. Jedinou podmínkou je, že RabbitMQ podporuje integraci pro daný jazyk (ten podporuje integraci pro obrovské množství jazyků) a tím umožní napojení na systém.

Přesto může vzniknout otázka, proč byl vybrán zrovna RabbitMQ místo jiného zprostředkovatele zpráv, popřípadě místo ESB (viz. kapitola č. 4). Proč byl pro FE zvolen Webassembly Blazor, nebo z jakého důvodu pro DevOps byl zvolen Azure Devops (a proč nebyl využit jeho zprostředkovatel zpráv).

Proč RabbitMQ?

Jak bylo zmíněno v kapitole 4.3, ESB je poměrně komplexní a těžkopádná varianta poskytující mnoho funkcionality navíc. Je tedy poměrně náročná na dlouhodobou údržbu a hůře škálovatelná. Cílem bylo najít něco odlehčeného, co bude poskytovat dostatek funkcionalit a nebude složité se zorientovat. Oproti Azure Devops umožňuje RabbitMQ

jednoduché nasazení na interním serveru (*Azure Devops bus* je spíše na Cloud). A co další podobné služby jako *Kafka* či *Redis*? Zde vzniká otázka, co v daném systému potřebujeme více. Jedná se o rychlost či zachování dat. V případě rychlosti (počet zpráv za sekundu) je Redis vhodný kandidát (na úkor zachování dat). Rovněž je principiálně *Push based,* zpráva je poslána spotřebiteli, jakmile je ve frontě (což je i RabbitMQ). Kafka je naopak *Pull based,* spotřebitel se na zprávu musí ptát, což není moc vhodné pro ReviewMe či jiné služby. RabbitMQ poskytl vhodný poměr mezi rychlostí a zachováním dat, zároveň je dobře vertikálně škálovatelný (na počet služeb) a poskytuje plno dalších funkcí [47].

Proč Webassemlby Blazor?

Vzhledem k tomu, že BE aplikace byl vytvořen v .NET (C#), byla poměrně přirozená volba dělat i FE část v .NET ekosystému (a nepoužívat JS a různé frameworky). To je nyní možné pomocí Webassemlby (která se nachází ve většině prohlížečů) a volbou byl tedy Blazor. Nebyla zde tedy při přechodu z BE na FE potřeba měnit jazyk a ekosystém. Client side byl zvolen kvůli nezávislosti FE na BE.

Proč Azure Devops?

Zde bylo rozhodnutí spíše subjektivní, ale vzhledem k tomu, že byl použit .NET ekosystém, byla volba Azure Devops (namísto Jenkins, nebo Github) poměrně přirozená.

7.3 Konvence

V rámci systému musely být nastoleny různé konvence hlavně pro formát přenosu dat a směrovací klíče.

Konvence dat

Data mezi službami jsou přenášeny ve formátu Json. Tabulky a názvy vlastností začínají velkými písmeny.

Konvence klíčů

V případě Topic Exchange je potřeba definovat podobu klíče. Jak klíče v RabbitMQ fungují bylo popsáno v kapitole 4.4.2. V případě tohoto systému bylo navrženo, že klíč bude mít tři části začínající velkými písmeny: *Access.Operation.Table (Public.Insert.Employee)*. Tento klíč zodpoví na tři otázky. Mám právo zpracovat tyto data? (Topic permissions). Jakou operaci mám s daty provést? Nad jakou tabulkou mám provést tuto operaci?

8 DEVOPS A RABBITMQ NASTAVENÍ

Před zahájením samotného vývoje jednotlivých služeb bylo potřeba nastavit Azure Devops z důvodu vizualizace postupu, plánování, verzování a dalším pomocných nástrojů v rámci Devops (CI, CD). Následně bylo potřeba promyslet a nastavit RabbitMQ tak, aby při vývoji už nenastaly další komplikace, popřípadě změny.

8.1 DevOps

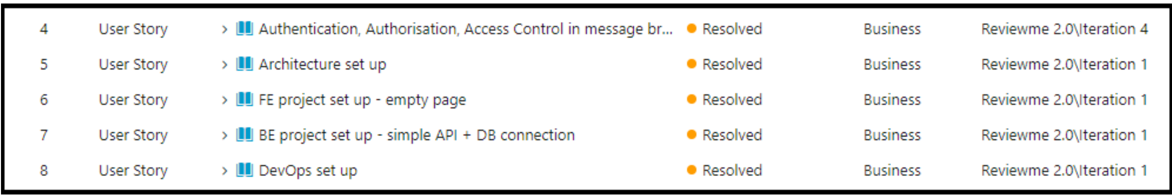
Teoretický úvod do Devops byl popsán v kapitole číslo 5. Následující sekce jsou tedy věnovány jeho nastavení pro konkrétní projekt. V první řádě si musíme založit Microsoft účet a následně vytvořit projekt se jménem naší aplikace, popřípadě systému. Následně z nabídky vybereme náš nový projekt. Poté bychom v pravé části obrazovky měli vidět vertikální menu pro jednotlivé kategorie činností. Začneme plánováním – *Boards*.

8.1.1 Backlog a plánování

V nabídce Boards nás budou převážně zajímat sekce – Backlog a Sprints. Boards vychází z agilního přístupu k vývoji SW, takže plánování, přírůstky, testování, demo a retrospektivu provádíme v rámci jednotlivých časových období (sprintů). Všechny plánovací nástroje jsou tomu tedy přizpůsobeny.

Backlog

V backlogu se nachází list rozdělaných vlastností, na kterých team pracuje (*user stories / work items*). Tyto vlastnosti jsou buď sepsány projektovým manažerem (z pohledu uživatele), popřípadě programátorem. V tomto listu jsou poté řazeny podle priority a vkládány do aktuální sprintu během plánování. Součástí každé položky v listu jsou poté jednotlivé úkoly (*tasks*), které je potřeba splnit, abychom danou vlastnost mohli považovat za dokončenou. Na obrázku můžeme vidět zkrácený list pro ReviewMe, během začátku vývoje. Ten byl poté aktualizován pro každou novou funkcionalitu.



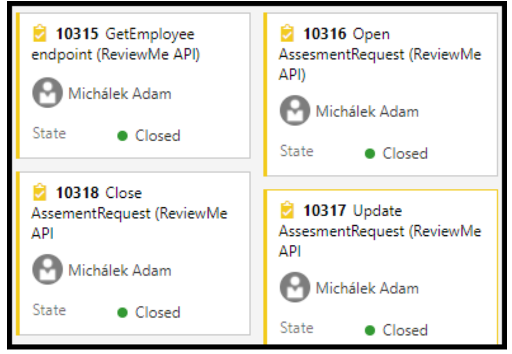
4	User Story	Authentication, Authorisation, Access Control in message br...	Resolved	Business	Reviewme 2.0\Iteration 4
5	User Story	Architecture set up	Resolved	Business	Reviewme 2.0\Iteration 1
6	User Story	FE project set up - empty page	Resolved	Business	Reviewme 2.0\Iteration 1
7	User Story	BE project set up - simple API + DB connection	Resolved	Business	Reviewme 2.0\Iteration 1
8	User Story	DevOps set up	Resolved	Business	Reviewme 2.0\Iteration 1

Obr č.21 – Backlog pro ReviewMe

Každá vlastnost se může nacházet ve stavu *New*, *Active*, *Resolved*.

Sprints

Zde se nachází soupis všech vlastností a jejich úkolů pro danou iteraci, opět vizuálně znázorněno pomocí *kanban* tabule (New, Active, Closed). Jednotlivé úkoly pro dokončení funkcionality jsou sepisovány samotným programátorem. Na obrázku můžeme vidět několik dokončených úkolů pro ReviewMe.



Obr č.22 – Úkoly pro ReviewMe

V Sprints sekci bylo tráveno poměrně velké množství času, protože poskytuje několik výhod, například vytvořit *branch* pro daný úkol nad daným repositářem.

8.1.2 CI

Před nastavením CI a CD je potřeba mít v repositáři uložen daný projekt (stejný proces jako v Github). Následně můžeme přejít do záložky *Pipelines* a *builds*. Zde si poté pomocí tlačítka *New* vytvoříme tři *Build pipeline* (ReviewMe FE, ReviewMe BE a EmployeeMicroservice). Následně je potřeba nastavit jednotlivé kroky. V první řádě nastavíme, aby se proces spustil při operaci nad *Master branch*. Poté nastavíme Pool, verzi .NET a instalaci NuGet balíčků.

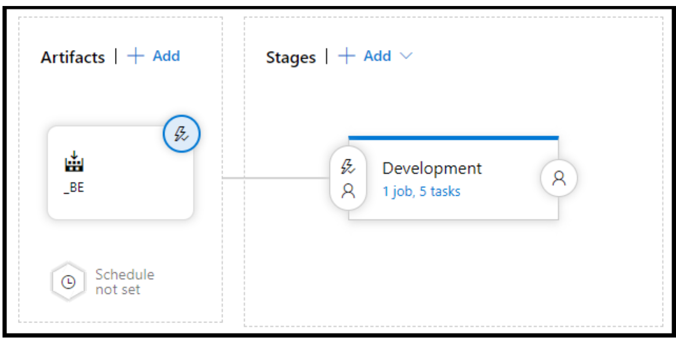
```
steps:
  - task: UseDotNet@2
    displayName: Use Dotnet 6
    inputs:
      version: '6.0.x'
      includePreviewVersions: true # Required for preview versions

  - task: NuGetToolInstaller@0
    displayName: "Install NuGet"
    inputs:
      versionSpec: 6.0.x
      checkLatest: true

  - task: DotNetCoreCLI@2
    displayName: Run unit tests
    inputs:
      command: test
      projects: "**/*Tests/*.csproj"
      arguments: "--configuration $(buildConfiguration) --logger trx"

  - script: dotnet publish ReviewMe.API/ReviewMe.API.csproj -o $(Build.ArtifactStagingDirectory)
    displayName: "dotnet publish ReviewMe.API"

  - task: PublishBuildArtifacts@1
    displayName: "Publish Artifact: drop"
    inputs:
      PathtoPublish: "$(build.artifactstagingdirectory)"
```

Obr č.23 – CI nastavení

Následně se snažíme sestavit samotnou aplikaci, spouští se dodatečné kontroly a generuje se artefakt. Zkrácenou verzi vidíme na obr. č.23.

8.1.3 CD

Po úspěšném provedení CI je vygenerován artefakt. Ten poté využijeme při nastavení *Release pipeline*. Přejdeme tedy do záložky Releases a vytvoříme nový release pro všechny naše služby.



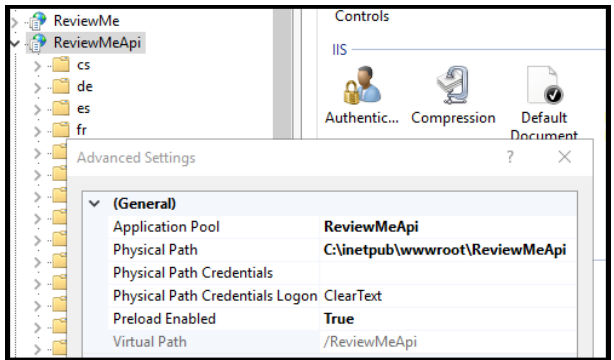
Obr č.24 – CD nastavení stage

Na obr. č. 24 můžeme vidět nastavení pro ReviewMe BE. V první řádě přidáme artefakt a nastavíme spouštěč po CI nad master branch. Následně přidáme stage pro vývoj, skládající se z jedné práce a 5 kroků. To si můžeme představit jako prostředí, kde se nasadí naše aplikace prakticky pár minut po aktualizaci repositáře. Jednotlivé kroky jsou poté velmi jednoduché, a byly ukázány a popsány v kapitole 5.3.

Jak bylo zmíněno, postup byl analogicky proveden pro každou službu či projekt, se kterým se v rámci systému pracovalo.

8.1.4 Nastavení serveru

Poslední částí je nastavení serveru, na kterém aplikace bude běžet. V praxi by se jednalo o interní server v rámci firmy. To bude v rámci práce simulováno jako lokální server. Na jeho nastavení použijeme IIS (Internet information service), kde můžeme registrovat naši službu na konkrétním portu a vyčlenit virtuální adresář pro naši aplikaci na disku. Program rovněž umožnuje nastavit mnoho dalších věcí, jako zabezpečení, cesty, certifikáty apod.



Obr č.25 – Nastavení serveru

V první řádě vytvoříme v *Sites* a *Default web sites* adresář pro naši aplikaci, jak vidíme na obr. č. 25. Poté u každé aplikace nastavíme aplikační pool a fyzickou a virtuální cestu. Po provedení dalších menších nastavení bychom měli mít připravený server.

Posledním krokem je nastavení aplikačního pool v Azure Devops pro náš server a přirazení serveru k Release pipeline, aby bylo zřejmé, na jaký server má být zveřejněn nový obsah. Celý proces je nyní automatizován. Kód, který vývojář přidá do repositáře, se nyní automaticky sestaví, otestuje a nasadí na serveru.

8.2 RabbitMQ

RabbitMQ služba by rovněž běžela na serveru firmy. Stejně jako v případě serveru, pro testování a simulaci bude spuštěn lokální server. V první řádě nainstalujeme nejnovější verzi *Erlangu* jazyku, ve kterém je RabbitMQ implementován. Poté samotnou aplikaci RabbitMQ. Její součástí jsou všechny potřebné komponenty a pluginy. Při prvním spuštění musíme službu ovládat pomocí příkazové řádky. Spustíme tedy *RabbitMQ Command Prompt* a vložíme následující příkazy:

rabbitmq-service start – nastartování služby

rabbitmqctl start_app – nastartování node na portu 5672 (v něm bude probíhat přenos zpráv)

rabbitmq-server -detached – odpojíme server od konzole (takže bude běžet pořád v pozadí)

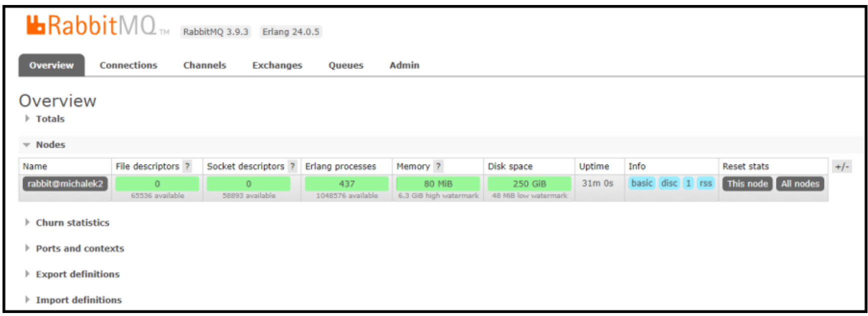
Abychom dále nemuseli další nastavení provádět v konzoli (často je to žádoucí, ale pro prvotní konfiguraci poněkud zdlouhavé), spustíme první plugin.

rabbitmq-plugins enable rabbitmq_management – webový UI plugin pro jednoduché ovládání RabbitMQ node

Nyní můžeme na localhostu (port 15672) provést další nastavení.

8.2.1 RabbitMQ UI management

Po prvotním přihlášení (login – guest a password – guest) jsme přesměrování na stránku a vidíme poměrně přehledné webové rozhraní skládající se z horizontálního menu v horní části obrazovky, a poté samotné tělo. Vpravo v hlavičce jsou další informace o konkrétním účtu. Nás budou zajímat primárně záložky *Overview*, *Exchanges*, *Queues* a *Admin* (*Connections* a *Channels* jsou také užitečné, ale spíše poskytují pouze informace o aktuálním připojení a vytvořených kanálech). Na obr. č.26 můžeme vidět rozhraní.



Obr č.26 – RabbitMQ management UI

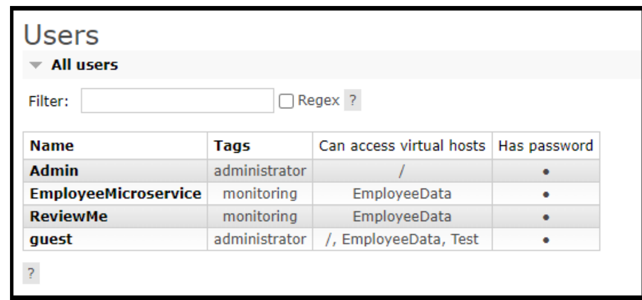
Overview

Zde se nachází obecné informace o aktivitě ve sběrnici, počtu zpráv za minutu, počtu spojení a kanálu, front, konzumentů a dalších statistik. Tyto informace se hodí pro debugging a pro kontrolu plynulosti toku dat. Jako admin můžeme rovněž exportovat, nebo importovat veškeré nastavení RabbitMQ, což je velmi užitečné v případě práce na více zařízeních.

Admin

Velmi důležitá sekce zahrnují vertikální menu pro nastavení uživatelů (aplikací), zabezpečení, virtuálního hosta, flagů, politik, limitů a clusterů. V systému budeme mít tři druhy zabezpečení. Aplikace, která chce komunikovat s RabbitMQ node, se musí autentizovat pomocí přihlašovacích údajů. Následně pro autorizaci musí být administrátorem přiřazena do určitého virtuálním hostu (včetně read-write-configure nastavení). Posledním druhem zabezpečení je tzv. *Topic permissions*. V tomto případě je směrovací klíč publikované zprávy brán v potaz při vynucení oprávnění. Tento druh zabezpečení zajistí, že aplikace, které nemá právo číst konkrétní typ zpráv (např. při určité úrovni zabezpečení), nebude schopna zprávu přečíst. To vše je nastaveno v této sekci.

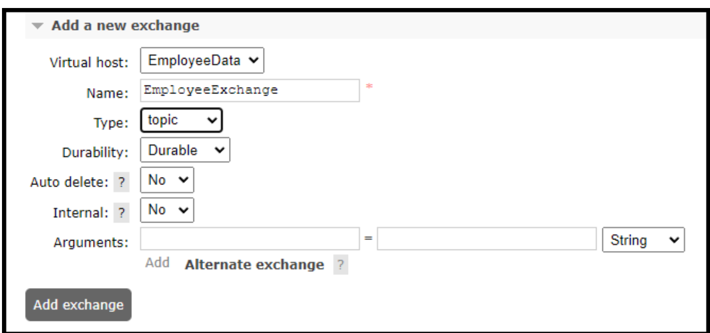
Jako první tedy přejdeme do záložky *Virtual Hosts,* a vytvoříme hosta s názvem EmployeeData. Poté v záložce *Users* vytvoříme účty pro naše aplikace (včetně administrátora) a přiřadíme je k virtuálnímu hostu, jak můžeme vidět na obrázku č.27 níže.



Obr č.27 – Users záložka

Exchanges

Po provedení dalších menších nastavení nyní můžeme přejít do sekce *Exchanges*. Zde vytvoříme EmployeeExchange nad konkrétním virtuálním hostem s následujícím nastavením:



Obr č.28 – Exchange setup

Při zvolení EmployeeExchange ze seznamu je zobrazeno spoustu užitečných statistik, a hlavně jaké služby jsou zrovna k Exchange připojené pomocí fronty. Rovněž můžeme nějakou frontu připojit manuálně, včetně nastavení směrovacího klíče.

Queues

Zde můžeme vidět všechny vytvořené fronty, napojené na konkrétní Exchange. Fronty se většinou vytvářejí, napojují a mažou automaticky (přímo z kódu). Pokud bychom však chtěli, můžeme zde vytvořit trvanlivou frontu a nastavit ji přesně podle našich požadavků. Rovněž můžeme zprávy manuálně číst, posílat, mazat, posouvat a sledovat užitečné metriky.

8.2.2 Topic persmissions

V rámci práce jsem navrhl testovací úrovně zabezpečení pro odesílané zprávy. Tyto úrovně by se v praxi musely důkladně prodiskutovat a namodelovat podle druhu dat a GDPR. Dejme tomu, že budeme mít tři skupiny – **public**, **private**, **protected**.

Public – veřejně dostupná data jako jméno, příjmení, pozice, věk

Protected – data s nějakou citlivější složkou (mobilní číslo, adresa, ssn)

Private – soukromá a chráněná data (plat)

ReviewMe bude mít přístup pouze k public a protected skupině a zapisovat bude moci pouze public data. Mikro služba bude moci zapisovat všechny druhy dat (protože bude mít přístup ke zdroji). Na obrázku níže můžeme vidět nastavení.

Topic permissions

Current topic permissions

User	Exchange	Write regexp	Read regexp	
EmployeeMicroservice	EmployeeDataExchange	^(public|private|protected).*	^public.*	Clear
ReviewMe	EmployeeDataExchange	^public.*	^(public|protected).*	Clear

Obr č.29 – Topic permissions

Jak bylo vysvětleno, Topic permissions funguje na základě porovnání klíčů. Proto je potřeba pomocí Regexu specifikovat, jak klíče porovnávat. V kapitole 7.3 byla vysvětlena konvence klíčů. Zajímá nás tedy pouze první část klíče udávající úroveň zabezpečení. Na obr. č. 29, například pro mikro službu, nastavujeme, že zapisovat může v případě existence jedné z úrovní v klíči. Zbytek klíče už je poté irelevantní (z hlediska zabezpečení).

Dejme tomu, že se ReviewMe snaží číst (nebo zapisovat) private data. V tomto momentě vyskočí výjimka, protože na to nemá právo.

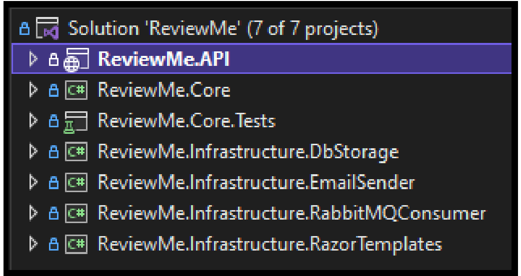
9 REVIEWME BE ČÁST

Asi nejdůležitější a nejrozsáhlejší částí práce je BE ReviewME. Rozsahově už se jedná o poměrně velkou aplikaci, proto do kapitol níže pochopitelně nešlo vepsat vše. Byla tedy snaha zachytit části, které jsou něčím zajímavé, popřípadě důležité z technologického, či business hlediska. Například nemá smysl popisovat každý koncový bod API, nebo metody v Core, popřípadě DB repositáři, protože logika často bude velmi podobná, nebo nezajímavá. Navíc v rámci práce byla snaha jednoty ve stylu programování, proto je jednodušší po pochopení jedné metody, porozumět dalším.

V první kapitole se podíváme na rozdělení projektu do vrstev podle čisté architektury (viz. kapitola č.2). Následně je provedena implementace jednotlivých uživatelských scénářů (business logiky). Poté se podíváme na systém notifikací a zasílání emailů. Poslední a velmi důležitá je kapitola o vytvoření RabbitMQ subscribera, který získává data pro naši aplikaci, popřípadě odesílá data do systému.

9.1 Vrstvy architektury a jejich popis

Samotná aplikace je vytvořená jako ASP.NET Core Web API pro .NET Core 6 bez dalšího nastavení. Jednotlivé vrstvy budou přidány jako Class library. Tyto vrstvy jsou členěny a pojmenovávány na základě čisté architektury. Zároveň chceme zajistit jednotný směr závislostí v projektu pomocí DI, jednotné názvy a co největší redukci duplicity. O jaké vrstvy se tedy bude jednat? Základem je **Core** vrstva obsahující veškerou business logiku, následně **API** vrstva pro nastartování aplikace a komunikaci s ní. Poté vrstvy infrastruktury obsahující různé vnější závislosti, jako databáze, nebo RabbitMQ. Patří sem **DbStorage**, **EmailSender**, **RabbitMQConsumer**, **RazorTemplates** a nakonec testy (**Tests**).



Obr č.30 – ReviewMe solution

Platí, že Core nemá žádnou vnější referenci (ostatní vrstvy vidí na Core, ale ten ne na ně). Komunikaci probíhá na základě registrace DI kontejnerů za pomocí rozhraní.

9.1.1 ReviewMe.API

Jedná se o vstupní bod webové aplikace. Obsahuje třídu **Program**, ve které provádíme veškeré počáteční nastavení aplikace. Patří sem registrace *Swagger* pro dukumentaci API, poskládání *middleware pipeline*, nastavení logování (Serilog), vytvoření kontrolérů a *Cors* politik. Dále nastavení zabezpečení a registrace DI kontejnerů. Na tyto dvě se podíváme detailněji.

Zabezpečení

K zabezpečení aplikace je použito zabezpečovací schéma **JwtBearer**. Ten, kdo bude chtít komunikovat s API, musí tedy poskytnou token (bezpečnostní kontext) na základě kterého se provede autentizace a autorizace (může uživatel použít API a konkrétně tento koncový bod?). Veškeré nastavení pro token byly provedeny v kódu níže.

```
56 builder.Services.AddAuthentication(configureOptions: options =>
57     {
58         options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
59         options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
60     })
61     .AddJwtBearer(jwtBearerOptions =>
62     {
63         var authenticationSettings = authenticationSettingsSection.Get<AuthenticationSettings>();
64 
65         jwtBearerOptions.SaveToken = true;
66         jwtBearerOptions.TokenValidationParameters = new TokenValidationParameters
67         {
68             ValidateIssuerSigningKey = true,
69             IssuerSigningKey =
70                 new SymmetricSecurityKey(Encoding.UTF8.GetBytes(authenticationSettings.SecretKey
71             ClockSkew = TimeSpan.FromSeconds(authenticationSettings.JwtClockSkewInSeconds)
72         };
```

Obr č.31 – Registrace autentizace pomocí JWT tokenu

Na obr. č. 31 vidíme registraci autentizační služby, kdy po nastavení schémat udáváme, jak má být token čten. Ověřuje se jeho pravost pomocí tajného klíče, nastavujeme jeho životnost a říkáme, že se má uložit (aby uživatel nemusel zadávat token po každém zavolání API).

Registrace DI kontejnerů

Autorizaci poté nastavujeme pomocí speciální služby v *Core* vrstvě, čímž se dostáváme ke konfiguraci služeb.

```
86 DiConfigCore.ConfigureServices(builder.Services, builder.Configuration);
87 DiConfigRabbitMqConsumer.ConfigureServices(builder.Services, builder.Configuration);
88 DiConfigDbStorage.ConfigureServices(builder.Services, builder.Configuration);
89 DiConfigEmailSender.ConfigureServices(builder.Services, builder.Configuration);
90 DiConfigRazorTemplates.ConfigureServices(builder.Services, builder.Configuration);
```

Obr č.32 – Konfigurace služeb

Každá vrstva obsahuje statickou třídu s metodou **ConfigureServices**, která nám zaregistruje jednotlivé služby pro danou vrstvu.

Uvnitř jedné z nich můžeme vidět následující nastavení.

```
17  public static class DiConfigCore
18  {
        1 reference | Michálek Adam, 44 days ago | 3 authors, 3 changes | 3 work items
19      public static void ConfigureServices(IServiceCollection services, IConfiguration configuration)
20      {
21          var applicationSettingsSection = configuration.GetSection(key: "ApplicationSettings");
22          services.Configure<ApplicationSettings>(applicationSettingsSection);
23
24          services.AddAuthorization(options => options.AddReviewmePolicies());
25
26          services.AddScoped<IClaimsTransformation, RoleClaimsTransform>();
27          services.AddScoped<ICurrentUserService, CurrentUserService>();
28
29          services.AddScoped<IEmployeesService, EmployeesService>();
```

Obr č.33 – Konfigurace služeb v CORE

Po načtení nastavení registrujeme jednotlivé služby pomocí **Scoped**, **Singleton**, popřípadě **Transient**. Například na řádku 29 říkáme, že pokud třída bude v konstruktoru obsahovat **IEmployeeService**, poskytni jí danou implementaci **EmployeeService** (Singleton vrací vždy jednu instanci, Scoped vytvoří novou při každém požadavku na server). Pokud bychom tedy chtěli změnit implementaci této služby, nemusíme zasahovat přímo na konkrétní místo, ale pouze zde změníme závislost na jinou službu. Další obrovskou výhodou tohoto přístupu jsou jednotkové testy. Jednotlivé služby nyní můžeme jednoduše namockovat a izolovaně je otestovat.

Kontroléry

Ve složce *Controllers* se nachází všechny koncové body pro komunikaci s ReviewME. Jsou rozdělené do skupin podle business logiky na **AssessmentControler**, **EmployessControler**, **ReviewersController**, **ReviewerTasksController**, **SynchronizeDataController**. Na obr. níže můžeme vidět kontrolér pro operace se zaměstnanci.

```
5   [Route(template: "[controller]")]
6   [ApiController]
    1 reference | Juřička Tomáš, 81 days ago | 1 author, 2 changes | 1 work item
7   public class EmployeesController : Controller
8   {
9       private readonly IEmployeesService _employeesService;
10
        0 references | Juřička Tomáš, 81 days ago | 1 author, 1 change | 1 work item
11      public EmployeesController(IEmployeesService employeesService)
12      {
13          _employeesService = employeesService;
14      }
15
16      [Authorize(Policy = AuthorizationPolicies.Employee)]
17      [HttpGet]
        0 references | Juřička Tomáš, 81 days ago | 1 author, 2 changes | 1 work item
18      public IReadOnlyCollection<EmployeeResponse> Get()
19          => _employeesService.Get();
20
21      [Authorize(Policy = AuthorizationPolicies.Employee)]
22      [HttpGet(template: "{id:int}")]
        0 references | Juřička Tomáš, 81 days ago | 1 author, 2 changes | 1 work item
23      public EmployeeResponse Get(int id)
24          => _employeesService.Get(id);
```

Obr č.34 – EmployeesController

Na začátku pomocí atributu *ApiController a Route* říkáme, že se jedná o API s cestou skládající se z názvu daného kontroléru (/Employees/..). Poté vkládáme závislost na IEmployeeService (kam bude vložena EmployeeService implementace). Následně v metodách Get voláme příslušnou metodu ze služby a předáváme ji parametr z URI (Id). Nad metodami si můžeme všimnout atributu *HttpGet*, kterým deklarujeme typ v rámci http protokolu. Nad ním poté voláme atribut *Authorize* s určitou bezpečnostní politikou (pro typ uživatele). Ta je extrahována z JWT tokenu a pro další požadavky uložena. Tento prototyp kontroléru je pro další koncové body prakticky totožný.

Middlewares a další nastavení

V poslední řádě se v této vrstvě nachází vlastní middlewares pro efektivnější zpracování chyb a detailnější logování v rámci požadavků. V průběhu práce si můžeme všimnout, že se nepoužívají žádné try-catch bloky. Důvod je ten, že v rámci **ErrorHandlingMiddleware** byl vytvořen jeden tento velký blok pro obrovské množství výjimek (které se logují). Ten musí být logicky umístěn na začátku pipeline. V případě, že dojde k selhání, aplikace nespadne a výstupní chyba bude zapsána.

Dále jsou podstatné konfigurační nastavení aplikace (*appsettings.json*). Zde se nachází nastavení pro databáze (spojovací řetězce), nastavení RabbitMQ, Serilogu, Emailu a samotné aplikace.

9.1.2 ReviewMe.CORE a Tests

Vrstva, která kromě business logiky, obsahuje vlastní autorizační službu, výjimky, pomocníky a další nastavení. Rovněž obsahuje *Infrastructure* složku definující množinu veřejných rozhraní pro všechny vnější závislosti. Služba pro získání všech zaměstnanců (například) z databáze bude tedy závislá na tomto rozhraní, místo konkrétní implementace z Entity frameworku. Tu poté vytvoříme v další vrstvě (a zdědíme od našeho rozhraní) a v DI nastavení zaregistrujeme. Postup je poté stejný pro každou službu v rámci této vrstvy (ale i ostatních). Pro každou přidanou vlastnost v rámci business logiky byly přidány jednotkové testy, nacházející se zvlášť ve vrstvě. Zde poté využíváme naplno výhod čisté architektury, protože každou službu můžeme naplnit testovacími daty (namockovat) a tím pádem testovat pouze jednotky kódu v rámci business logiky (ne vnější závislosti).

Jakým způsobem je tedy v rámci Core členěn kód? Na obrázku můžeme vidět složkovou hierarchii.

```
ReviewMe.Core                         Infrastructures
  Dependencies                          IAssessmentReviewerRepository.cs
  Authorization                         IAssessmentsNotificationService.cs
  DomainEntities                        IAssessmentsRepository.cs
    EmailTemplateModels                 IBusinessLogicAuthService.cs
    Assessment.cs                       ICurrentUserService.cs
    AssessmentReviewer.cs               IEmailSender.cs
    EmailMessage.cs                     IEmailSendingService.cs
    Employee.cs                         IEmployeesRepository.cs
    IEntity.cs                          IGenericRepository.cs
  Enums                                 IRazorTemplateParsingService.cs
  Exceptions                            ISynchronizeData.cs
  Helpers                             Services
  Infrastructures                       CommandServices
  Services                              EmailNotificationServices
  Settings                              QueryServices
  DiConfigCore.cs                       DateTimeProvider.cs
```

Obr č.35 – Core vrstva

Základem jsou doménové entity, objekty v rámci business logiky ReviewME. Na obr. č.35 můžeme vidět jejich výčet. Infrastructure složka již byla zmíněna výše. Nejdůležitější je složka *Services* obsahující veškerou business logiku ReviewME. Této části se detailně zabývá kapitola 9.2, včetně pokrytí všech uživatelských požadavků. Zajímavá je poté složka *Authorization*, kde vytváříme vlastní autorizační role a politiky v rámci aplikace, skládáme bezpečnostní kontext (ClaimsPrincipal) a identifikujeme uživatele pomocí *WindowsIdentity*.

Autorizace

Autorizační část obsahuje statickou třídu nastavující autorizační politiky. Jedna je pro roli zaměstnance, druhá pro administrátora. Tato třída se volá z *DiConfigCore.cs*. Na následujícím obrázku můžeme vidět její kód.

```
public const string Employee = "Employee";
public const string SuperAdmin = "SuperAdmin";

1 reference | Juřička Tomáš, 84 days ago | 1 author, 1 change | 1 work item
public static void AddReviewmePolicies(this AuthorizationOptions options)
{
    options.AddPolicy(name: Employee, configurePolicy: policy =>
    {
        policy.RequireAuthenticatedUser();
        policy.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireClaim(RoleClaimType, params allowedValues: AuthRoleTypes.Employee.ToString());
    });

    options.AddPolicy(name: SuperAdmin, configurePolicy: policy =>
    {
        policy.RequireAuthenticatedUser();
        policy.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireClaim(RoleClaimType, params allowedValues: AuthRoleTypes.SuperAdmin.ToString());
    });
}
```

Obr č.36 – Registrace bezpečnostních politik

V podstatě zde pouze pro danou roli nastavujeme autentizační schéma a Claim, který se musí nacházet v bezpečnostním kontextu (ClaimsPrincipal). Ten poté aktualizujeme v rámci každého požadavku pomocí třídy *RoleClaimsTransform*. Ta musí dědit od *IClaimsTransformation* a přetížit metodu *TransformAsync*. Uvnitř metody nejprve musíme získat potřebná nastavení z konfiguračního souboru (appsettings), a to zejména tajný klíč,

podporované role a jejich doménové účty (abychom dokázali ověřit, jestli se daný uživatel nachází v AD skupině). Na základě toho poskládáme seznam Claims, vytvoříme z něj identitu a z té zase ClaimPrincipal. Aplikace má nyní uložený bezpečnostní kontext. Při vložení *JwtBearer* tokenu a zavolání koncového bodu poté ReviewME dokáže ověřit, jestli uživatel má, nebo nemá přístup pro daný zdroj. Tato logika umožňuje nejen jednoduše rozšířit povolené role, ale i vytvářet komplexní bezpečnostní politiku v případě potřeby.

Další

V poslední řádě Core obsahuje pomocníky (statické třídy provádějící konkrétní transformace s daty), vlastní výjimky, nastavení týkající se samotné aplikace včetně rolí a samozřejmě DI nastavení.

9.1.3 ReviewME.Infrastructure.DbStorage

Hlavním účelem této vrstvy je ukládání dat v relační databázi (SQL). Tyto data přijdou z RabbitMQ, popřípadě budou v rámci business logiky vytvořeny a uloženy. ReviewME si podle SOA (popřípadě Microservice) architektury ukládá pouze data, která nevyhnutelně potřebuje pro své fungování a na které má práva. Jako hlavní framework pro práci s databází byl použit populární *Entity framework* (a jeho potřebné varianty). Ten umožňuje pomocí přístupu *code-first* navrhnout v C# objekty a vztahy mezi nimi, ze kterých posléze bude vygenerována SQL databáze. Dále poskytuje jednoduchý a praktický systém migrací (v případě rozšíření databáze) a aktualizací, s ohledem na bezpečnost a efektivitu jednotlivých SQL dotazů. Při návrhu byl použit *Repository* vzor, který nám abstrahuje společnou funkcionalitu pro jednotlivé tabulky do třídy. Ty dědí od rozhraní (z core.infrastructure) a implementují metody pro jednotlivé operace nad konkrétní tabulkou. Tyto třídy opět musíme zaregistrovat v rámci DI, aby se do *Core* dostala tato konkrétní implementace. V případě změny databáze poté stačí zaměnit volání registrace dané databáze.

Další zajímavostí pro tuto vrstvu (stejně jako *RabbitMQConsumer*) je použití *Automapper* knihovny. Ta slouží pro mapování z jedné verze objektu (té databázové, např. EmployeeDo), do *Core* verze (Employee). Důvodem, proč každá vrstva má svoji verzi objektu, můžou být rozdíly v názvu a dále v existenci určitých vlastností, které se v každé vrstvě podle potřeby mohou lišit.

Jak vypadá tato vrstva můžeme vidět na obrázku níže.

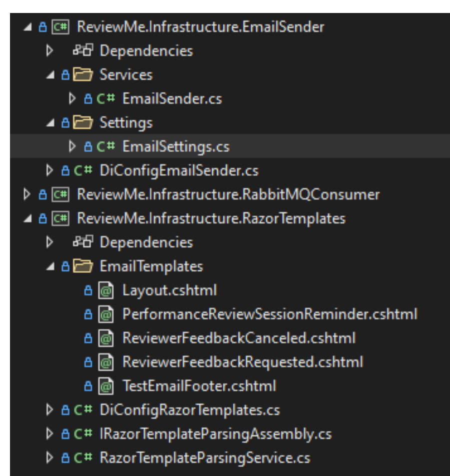
ReviewMe.Infrastructure.DbStorage
Dependencies
Comparers
DatabaseObjects
AssessmentDo.cs
AssessmentReviewerDo.cs
EmployeeDo.cs
Extensions
MapperExtensions.cs
Repository
AssessmentReviewerRepository.cs
AssessmentsRepository.cs
EmployeesRepository.cs
GenericRepository.cs
AutoMapProfile.cs
DiConfigDbStorage.cs
globalusings.cs
ReviewMeDbContext.cs

Obr č.37 – ReviewMe.Infrastructure.DbStorage

Na obr. č. 37 vidíme, že vrstva obsahuje kromě výše popsaných věcí *AutoMapProfile* pro nastavení mapování objektů. Následně obsahuje *ReviewMeDbContext*, velmi důležitý soubor pro nastavení Entity frameworku. Tato třídy musí dědit od *DbContext* třídy a obsahuje jednotlivé tabulky a nastavení pro prvotní konstrukci databáze (nastavení vztahů apod.).

9.1.4 ReviewME.Infrastructure.EmailSender a Razor templates

Jedná se o vrstvy poskytující zasílání emailu a skládání *Razor* šablon pro jednotlivé typy emailů. Jedná se o poměrně jednoduché vrstvy s několika službami, které využívají externí knihovnu, například *MailKit*. Notifikacím se detailně budeme věnovat v kapitole 9.3. Co vše vrstvy obsahují se můžeme podívat na obrázku níže.



Obr č.38 – ReviewMe.Infrastructure.EmailSender a RazorTemplates

EmailSender *vrstva*

Kromě objektu pro nastavení (nahrané z appsettings) se zde nachází **EmailSender** služba. Ta poskytuje metodu pro zaslání emailu pomocí předaných parametrů (předmět, tělo, příjemci a další). Z důvodu jednoduchého použití byla vybrána knihovna *MailKit*.

RazorTemplates vrstva

Tato vrstva obsahuje vlastnoručně vytvořené šablony pro jednotlivé emaily přicházejících v různých okamžicích více lidem. Důvod umístění šablon do speciální vrstvy je opět jejich možná nahraditelnost jinými šablonami, popřípadě jiným vnějším poskytovatelem. Na obrázku č. 38 můžeme vidět šablonu *Layout,* což je v podstatě základ každého emailu (hlavička, patička, logo, aj). Uvnitř se poté nachází zástupný řetězec, který je poté nahrazen konkrétní šablonou. Z důvodu různých operací s .cshtml razor soubory byla přidána služba *RazorTemplateParsingService,* pro vytvoření finálního .html řetězce a získání zabudovaného zdroje (konkrétní šablony).

9.1.5 ReviewME.Infrastructure.RabbitMQConsumer

Poslední přidaná vrstva zajištuje kontinuální aktualizaci dat v ReviewME. Rovněž umožňuje provést synchronizaci dat v případě dlouhodobějšího výpadku (RPC požadavek). Hlavním požadavkem pro tuto vrstvu byla znuvupoužitelnost (pro další služby v systému) a důraz byl kladen rovněž na zabezpečení dat. Služba, odposlouchávající data ze sběrnice, je vždy spuštěna po zapnutí ReviewMe aplikace. Toho je dosaženo děděním od *IHostedService* a (jako ostatní služby ve vrstvách) registrací hned při startu. Jak je tedy vlastně vrstva členěna?

Obr č.39 – ReviewMe.Infrastructure.RabbitMqConsumer

Nejdůležitější službou je *RabbitMqConsumerHostedService,* která spustí celý proces. V případě, že je schopna navázat spojení, iniciuje proces průběžného čtení dat z dané Exchange (v případě že ne, tak to zkouší nadále). Jednotlivé příchozí zprávy jsou poté zpracovány podle klíče a přesměrovány dále ke zpracování na příslušný *Handler* (např. handler pro rozšíření databáze). Kromě pomocníků, vlastních objektů a dalších nastavení obsahuje službu pro kontrolu stavu (z důvodu bezpečnosti) a pro synchronizaci.

Více o této vrstvě v kapitole 9.4.

9.2 Business logika

Tato kapitola se věnuje implementaci uživatelských požadavků, popsaných v kapitole č.6. Nebudou zde zobrazeny všechny, ale jen ty, které byly nějakým způsobem důležité či zajímavé. Jak už bylo řečeno, veškerá business logika aplikace se nachází v *ReviewMe.Core.Services.* Uvnitř se poté rozděluje podle druhu služby (*CommandServices*, *QueryServices, a další*). V neposlední řadě se zde nachází služby abstrahující knihovny pro práci s datumem a časem, aby v jednotkových testech poté mohly být nahrazeny testovacími daty.

Začneme tedy jednodušším požadavkem, na kterém si ukážeme, jak jsou obecně služby implementovány.

9.2.1 Výpis všech zaměstnanců

Celková struktura všech služeb je velmi podobná. V první řadě je vytvořeno rozhraní obsahující metody pro danou službu.

```
3  public interface IEmployeesService
4  {
5      IReadOnlyCollection<EmployeeResponse> Get();
6
7      EmployeeResponse Get(int id);
8  }
```

Obr č.40 – IEmployeeService

Na obr. č .40 můžeme vidět službu pro zaměstnance. Byly potřeba dvě metody, jedna pro vrácení jednoho zaměstnance podle id (řádek 7), druhá pro vrácení všech zaměstnanců (řádek 5). Z kódu si můžeme všimnout, že veškeré odpovědi jsou mapovány na speciální *Response* objekt, který poté při požadavku API vrací. Rovněž vidíme, že s listy je v rámci celého řešení pracováno (když je to žádoucí) jako s *IReadOnlyCollection.* Jak z názvu vypovídá, jedná se o kolekci pouze pro čtení.

```
3  public class EmployeesService : IEmployeesService
4  {
5      private readonly IEmployeesRepository _employeesRepository;
6
7      public EmployeesService(IEmployeesRepository employeesRepository)
8      {
9          _employeesRepository = employeesRepository;
10     }
11
12     public IReadOnlyCollection<EmployeeResponse> Get()
13         => _employeesRepository // IEmployeesRepository
14             .Get() // IReadOnlyCollection<Employee>
15             .Where(employee => employee.IsActive) // IEnumerable<Employee>
16             .Select(ToEmployeeResponse) // IEnumerable<EmployeeResponse>
17             .ToList(); // List<EmployeeResponse>
```

Obr č.41 – EmployeeService

Dále můžeme vytvořit samotnou službu *EmployeeService*. Ta samozřejmě dědí od rozhraní výše a implementuje obě metody (na obr. č.41 je zobrazena pouze jedna). V první řádě musíme do třídy vložit externí závislost přes rozhraní. Na obr. č.41 poté můžeme vidět registraci DI pro vložení závislost na konkrétní implementaci. Následuje samotná logika aplikace. Pomocí proměnné repositáře přistoupím k metodě *Get* a následně jsou pomocí LINQ vyfiltrováni pouze aktivní zaměstnanci, převedeni na *Response* entitu (nacházející se ve stejné složce) a vráceni jako list. Tímto způsobem byla psána většina služeb. K transformaci nad daty se používal výhradně LINQ, což vede ke kratšímu a čitelnějšímu zápisu. Nevýhodou je horší debugging, ale s trochou snahy je i on dostačující.

Jak vypadá databázová metoda *Get* pro získání dat?

```
3  internal class EmployeesRepository : IEmployeesRepository
4  {
5      private readonly ReviewMeDbContext _db;
6      private readonly IMapper _mapper;
7
8      public EmployeesRepository(ReviewMeDbContext db, IMapper mapper)
9      {
10         _db = db;
11         _mapper = mapper;
12     }
13
14     public IReadOnlyCollection<Employee> Get()
15         => _db.Employees // DbSet<EmployeeDo>
16             .Include(navigationPropertyPath: employee => employee.Assessments) // IIncludableQuer
17             .OrderBy(employee => employee.SurnameFirstName) // IOrderedQueryable<EmployeeDo>
18             .ToList() // List<EmployeeDo>
19             .Map<IReadOnlyCollection<Employee>>(_mapper); // IReadOnlyCollection<Employee>
```

Obr č.42 – EmployeeRepository

V první řadě musíme injektovat závislost na DB kontext a poté *IMapper*. V metodě na obr. č.42 (řádek 14) je nad tabulkou zaměstnanci volána metoda *Include* pro jednotlivé ohodnocení (databázový *join*), abychom k nim přes zaměstnance mohli přistupovat. Poté dojde k seřazení podle jména a převedení na list. Nakonec je provedeno mapování na list *Core* zaměstnanců a výstup je vrácen. Tato metoda je opět prototyp toho, jak vypadají ostatní metody v repositáři. Často se liší větším počtem *Include,* jinou transformací dat, ale v podstatě jsou velmi podobné. Pravidlem bylo, aby data z repositáře byla co nejobecnější.

9.2.2 Otevření a práce s ohodnoceními

Základem aplikace jsou jednotlivé ohodnocení (assessment). Jedná se o entity vytvořené nad konkrétním zaměstnancem (proto na něj mají referenci), obsahující datumy, stav ohodnocení a list dalších zaměstnanců, kteří mají provádět ohodnocení. Všechny tyto informace vloží administrátor do formulářů a odešle na jeden z koncových bodů. Jejich počet se váže k počtu stavů, kterých může dané ohodnocení nabýt.

Jaké to tedy jsou stavy?

Open – ohodnocení je otevřené, čeká se na posbírání zpětné vazby od zaměstnanců

Closed – ohodnocení bylo zavřeno, ať už manuálně administrátorem, či automaticky po vypršení doby sbírání zpětné vazby

Deleted – ohodnocení je smazáno, například z důvodu odchodu zaměstnance

Na obr. č.43 níže vidíme rozhraní pro naši službu.

```
3  public interface IAssessmentsService
4  {
5      void OpenAssessment(int employeeId, OpenAssessmentRequest request);
6      void UpdateAssessment(int employeeId, UpdateAssessmentRequest request);
7      void CloseAssessment(int employeeId);
8      void DeleteAssessment(int employeeId);
9  }
```

Obr č.43 – IAssessmentService

Pokud nepočítáme aktualizaci, metody odpovídají každému stavu.

```
public void OpenAssessment(int employeeId, OpenAssessmentRequest request)
{
    var now :DateTimeOffset = _dateTimeProvider.Now();
    var assessment = new Assessment
    {
        AssessmentState = AssessmentState.Open,
        AssessmentDueDate = request.AssessmentDueDate,
        PerformanceReviewDate = request.PerformanceReviewDate,
        EmployeeId = employeeId,
        CreatedByEmployeeId = CurrentEmployeeId,
        CreatedAt = now,
        AssessmentReviewers = request.Reviewers // IReadOnlyCollection<int>
            .Distinct() // IEnumerable<int>
            .Select(reviewer :int => new AssessmentReviewer
            {
                EmployeeId = reviewer
            }) // IEnumerable<AssessmentReviewer>
            .ToList() // List<AssessmentReviewer>
    };
```

Obr č.44 – OpenAssessment část 1

V první části metody skládáme *Assessment* objekt z příchozí zprávy. Kromě standardních vlastností přiřazujeme i objekt *AssessmentReviewers.* Jedná se o třetí hlavní entitu v ReviewMe (a i poslední, pokud se nepočítají emailové). Slouží pro uchování vztahu jednotlivých hodnotitelů přiřazených k danému ohodnocení. Tato entita je ve vztahu one-to-many k entitě zaměstnanec. Ten tedy vždy zná seznam všech svých hodnotitelů a rovněž ohodnocení. Vztah mezi *Assessment* a *AssessmentReviewers* je poté rovněž one-to-many, protože k danému ohodnocení může být přiřazeno více hodnotitelů.

Jak vypadá druhá část metody?

```
if (request.AssessmentDueDate <= now)
    throw new ErrorTypeException(ErrorType.GeneralRequestValidation,
        message: $"Assessment due date cannot be in past ({assessment.AssessmentDueDate})!");

if (request.PerformanceReviewDate <= assessment.AssessmentDueDate)
    throw new ErrorTypeException(ErrorType.GeneralRequestValidation,
        message: $"Performance review date cannot be before Assessment due date  (Assessment Due

if (_assessmentsRepository.Get(employeeId, AssessmentState.Open) is not null)
    throw new ErrorTypeException(ErrorType.GeneralRequestValidation,
        message: $"Employee '{employeeId}' already has an open assessment!");

_assessmentsRepository.Add(assessment);

_assessmentsNotificationService.NotifyOnOpenAssessment(
    employeeId,
    assessment.AssessmentReviewers.Select(assessmentReviewer => assessmentReviewer.EmployeeId)
```

Obr č.45 – OpenAssessment část 2

Ve druhé části (na obr. č.45) provádíme několik validací datumů a dále testujeme, jestli náhodou zaměstnanec nemá otevřené ohodnocení (může jich mít více smazaných či zavřených, ale nesmí mít otevřené). Poté pomocí repositáře přidáme ohodnocení do databáze. V poslední řádě musíme poslat notifikaci všem hodnotitelům, danému zaměstnanci a dalším lidem. Zavoláním metody *NotifyOnOpenAssessment* a předáním parametrů, toho docílíme. Více o této službě v kapitole 9.3.

Aktualizace ohodnocení vypadá poté velmi podobně. Liší se pochopitelně jen databázová a notifikační metoda (a pár dalších eventualit). Zavření a smazání je poté velmi jednoduché, ohodnocení je nalezeno v databázi, změněno (stav) a aktualizováno. V případě smazání je navíc poslán další email.

9.2.3 Provedení ohodnocení a další operace

Po otevření ohodnocení je nyní nutné posbírat zpětnou vazbu od zaměstnanců (kolegů hodnoceného člověka). Služba poskytující tuto logiku se jmenuje *ReviewerTaskService.* Uživatel tedy vyplní dané ohodnocení, které bude uloženo do databáze. Rovněž bude schopen poslat jen dočasný nástřel a vrátit se k němu později. V poslední řádě může být ohodnocení odmítnuto, s podmínkou poskytnutí důvodu.

```
3  public interface IReviewerTaskService
4  {
5      GetReviewerTasksResponse Get();
6      GetReviewerTaskResponse Get(int assessmentId);
7      void Decline(int assessmentId, DeclineRequest request);
8      void Draft(int assessmentId, DraftRequest draftRequest);
9      void Submit(int assessmentId, SubmitRequest submitRequest);
0  }
```

Obr č.46 – IReviewerTaskService

Na obr. č.46 můžeme vidět rozhraní se všemi dostupnými operacemi. FE pochopitelně vyžadoval i zobrazení jednotlivých úkolů, proto se zde nachází metoda pro vrácení jednoho, nebo všech. Jednotlivé úkoly se rovněž můžou nacházet ve stavu, měnícího se podle zavolané metody. Jaké to jsou stavy?

Declined – hodnotitel odmítl hodnotit s poskytnutím důvodu

Drafted – hodnotitel si uložil rozpracované ohodnocení (ale ještě ho neodeslal)

Reviewed – hodnotitel dokončil a odeslal ohodnocení

Vidíme, že stavy opět odpovídají jednotlivým koncovým bodům. Metody jsou opět velmi podobné, proto bude ukázaná pouze jedna. Na obrázku níže můžeme vidět metodu pro dokončení ohodnocení.

```
98   public void Submit(int assessmentId, SubmitRequest submitRequest)
99   {
100      var assessmentReviewer = _assessmentReviewerRepository.Get(GetCurrentEmployeeId(),
101
102      CheckAssessmentReviewer(assessmentReviewer, assessmentId);
103      CheckAssessmentReviewerState(assessmentReviewer);
104
105      assessmentReviewer.AssessmentReviewerState = AssessmentReviewerState.Reviewed;
106      assessmentReviewer.Feedback = submitRequest.Feedback;
107      assessmentReviewer.AreasForImprovements = submitRequest.AreasForImprovements;
108
109      _assessmentReviewerRepository.Update(assessmentReviewer);
110  }
```

Obr č.47 – ReviewerTaskService – Submit

V první řadě si z databáze vytáhneme již zmiňovaný objekt *AssessmentReviewer* (zaměstnanec, co má hodnotit zaměstnance s otevřeným ohodnocením). Následují verifikační metody stavů, datumů a dalších dat. Posléze přiřadíme data tykající se samotného hodnocení a zavoláme databázovou metodu pro aktualizaci. Metoda pro *Draft* je prakticky stejná, kromě jiného stavu. U *Decline* se navíc provádí dodatečné kontroly a zpětná vazba je nahrazena důvodem odmítnutí. Metody pro vrácení všech úkolu jsou poté velmi jednoduché, v podstatě pouze získáme přes *AssessmentReviewer* všechny úkoly (u každého ohodnocení jeden) a transformujeme je na speciální objekt. Ten vrátíme na výstup.

9.2.4 Další služby

Mezi další služby patří například *ReviewersService*, která získává výčet kolegů pro daného zaměstnance. Tento list je poté zobrazen při vytváření ohodnocení s možností zvolit vhodného hodnotitele. Služba bude velmi specifická pro každou firmu. V případě této práce budou použita testovací data, z další služby – ColleagueService. Ta by v praxi mohla, jako součást systému, posílat data do sběrnice, o které by ReviewMe měla zájem.

9.3 Systém zasílání notifikací

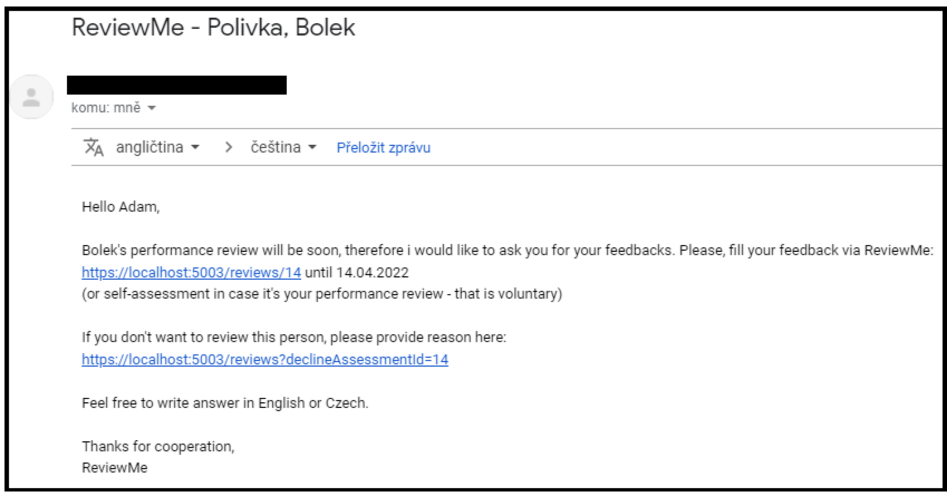
Jak už bylo řečeno, uživatel bude převážně vstupovat do aplikace přes příchozí email, který ho informuje o dané akci (ohodnocení, PR apod..) včetně užitečných informací a poskytne mu odkaz do FE části aplikace. Předpokládalo se, že notifikace budou používány i dalšími službami v systému, proto byla snaha o obecnost a znuvupoužitelnost.

9.3.1 Šablony, tvorba a poslání emailu

V kapitole 9.1.4 bylo popsáno základní rozvržení komponentů a jakým způsobem se pracuje se šablonami. V podstatě notifikace můžeme rozdělit na tři části. Samotné posílání emailů. Šablony pro jednotlivé emaily, rozdílné podle akce. Notifikační business logika pro jednotlivé akce (kdy a s jakými daty se email pošle). Na obr. č.45 vidíme, že emaily se posílají pouze při administrátorských operacích s ohodnoceními (vytvoření, modifikace, smazání). Tomu také odpovídají tři druhy šablon, které bylo potřeba vytvořit (a navíc rodičovská a testovací šablona). Jaké to jsou?

ReviewerFeedbackRequested

Šablona pro email informující zaměstnance o možnosti ohodnotit jeho kolegu. Na obrázku č.48 můžeme vidět finální email.



Obr č.48 – Notifikační email pro ohodnocení

Předmět emailu obsahuje jméno hodnoceného kolegy a zdrojovou aplikaci. Tělo poté krátkou zprávu informující uživatele, že tento kolega bude mít PR do určitého datumu. Dále se zde nachází dva odkazy na FE ReviewMe. Jeden do sekce pro hodnocení, druhý otevře dialog pro odmítnutí s poskytnutím důvodu. Html kód byl tvořen pomocí Razor šablon (.cshtml souborů).

ReviewerFeedbackCanceled

Jednoduchá šablona pro případ, kdy zpětná vazba už není potřeba (ohodnocení bylo smazáno, popřípadě nevyplněno do určitého data). Stejně jako v předchozím případě ji dostanou pouze hodnotitelé.

PerformanceReviewSessionReminder

Tato šablona je určena pro projektové manažery, HR, a hodnoceného člověka. Na základě ní se může domluvit přesné datum schůzky (popřípadě upravit, aby vyhovovalo všem stranám). HR zde navíc vidí i list uživatelů, kteří byli vybrání pro hodnocení. Jedná se o velmi jednoduchou šablonu.

Layout

Rodičovská šablona definuje styly, hlavičku, patičku a pozadí. Rovněž obsahuje zástupný symbol, za který bude vložena jedna z šablon výše. Tím se zajistí, že všechny budou mít společný design. Abychom tuto operaci mohli jednoduše provést, byla ve vrstvě šablon přidána *RazorTemplateParsingService.*

O službě *EmailSender* byla zmínka v kapitole č.9.1.4. V podstatě se jedná o integraci pro vnější službu pro zasílání emailů. Abychom však dokázali jednoduše spojit logiku šablon výše, se službou zasálání emailů, musela být vytvořena poslední pomocná komponenta s názvem *EmailSendingService*. Ta, kromě výše zmíněných závislostí, obsahuje dvě metody. Jednu veřejnou pro zasílání emailu. Druhou privátní pro poskládání výsledného emailu z šablon. Můžeme ji vidět na obrázku níže.

```
34    private async Task<EmailMessage> GetEmailMessage<T>(string razorTemplate, T model, string subject,
35        IEnumerable<string> recipientsTo,
36        IEnumerable<string> recipientsCc,
37        IEnumerable<string> recipientsBcc)
38    {
39        var layout :string = await _razorTemplateParsingService.GetEmbeddedStaticTemplate("Layout.cshtml", folderName: "EmailTemplates");
40
41        var body :string = await GetBodyFromEmailRazorTemplate(razorTemplate, model);
42
43        var emailMessage = new EmailMessage
44        {
45            To = recipientsTo,
46            Cc = recipientsCc,
47            Bcc = recipientsBcc
48        };
49
50        if (_applicationSettings.UseTestEmailAddresses && _applicationSettings.TestEmailAddresses.Count > 0)...
68
69        emailMessage.Subject = subject;
70        emailMessage.Content = layout
71            .Replace(oldValue: @"{{tableBody}}", newValue: body); // string
72
73        return emailMessage;
74    }
```

```
76    private async Task<string> GetBodyFromEmailRazorTemplate<T>(
77        string razorTemplate, T model)
78        => await _razorTemplateParsingService.RenderAsync(razorTemplate, model);
79    }
```

Obr č.49 – GetEmailMessage metoda

Vidíme, že metoda přijímá všechny potřebné argumenty pro zaslání emailu. Ty jsou poté vloženy do *EmailMessage* objektu. Jako první na řádku 39 získáme rodičovský layout, poté načteme konkrétní email (jako řetězec) na základě šablony a modelových dat (které budou v šabloně nahrazeny). Po konstrukci objektu na řádku 50 ověříme, jestli se jedná o testovací

email. Pokud ano, bude poslán pouze lidem zapsaným v konfiguraci. Následně přidáme předmět a tělo, které bude *layout* s nahrazeným zástupným symbolem danou šablonou s daty.

Následně se ve veřejné metodě výše zavolá služba pro zaslání emailu na základě výše uvedených dat. Tato spojovací služba je poté injektována do notifikační služby, která bude mít pouze jednu externí závislost pro email.

9.3.2 Notifikační služba

Středobodem je tedy notifikační služba. Jejím hlavním cílem je získat data pro šablonu emailu, a to primárně z databáze. Jedná se o seznam příjemců, podobu předmětu a další informace pro tělo. Tyto data bylo někdy obtížnější získat v požadované formě, proto služba také obsahuje různé transformace nad daty. Co vše služba implementuje?

```
public interface IAssessmentsNotificationService
{
    Task NotifyOnOpenAssessment(int employeeId, IReadOnlyCollection<int> reviewers);
    Task NotifyOnUpdateAssessment(int employeeId, IReadOnlyCollection<int> reviewers, IReadOnlyCollection<int> canceledReviewers);
    Task NotifyOnDeleteAssessment(int employeeId, IReadOnlyCollection<int> canceledReviewers);
}
```

Obr č.50 – IAssessmentNotificationService

Na obr. č.50 vidíme metody, které odpovídají jednotlivým krokům při práci s ohodnocením. Všechny metody přijímají id hodnocené osoby. Následně list (id) jednotlivých hodnotitelů. V případě aktualizace a smazání také list s hodnotiteli, jejichž zpětná vazba již není potřeba. Na obrázku níže si poté můžeme prohlédnout první část metody *NotifyOnOpenAssessment*.

```
30  public async Task NotifyOnOpenAssessment(int employeeId, IReadOnlyCollection<int> reviewers)
31  {
32      var employee = _employeesRepository.Get(employeeId);
33      var assessment =
34          employee.Assessments.FirstOrDefault(assessment => assessment.AssessmentState == AssessmentState.Open);
35      if (assessment == null)
36      {
37          _logger.LogError(message: "Cannot send email because assessment for employee: {employeeId} is not open", employeeId);
38          return;
39      }
40      var reviewMeUrl :IConfigurationSection? = _configuration.GetSection(key: "ReviewMeUrl");
41      var (recipientEmails :IReadOnlyCollection<string>, recipientNames :IReadOnlyCollection<string>) = GetRecipientsLoginAndName(reviewers);
42      var subjectReviewerReminder :string = "ReviewMe - " + employee.SurnameFirstName;
43      var recipients :IEnumerable<(First,Second)> = recipientEmails.Zip(recipientNames);
44
45      foreach (var (to :string, name :string) in recipients)
46      {
47          await SendTo(EmailTemplate.ReviewerFeedbackRequested, recipients: new List<string> { to },
48              subjectReviewerReminder, new ReviewerFeedbackRequestedModel
49          {
50              ReviewerName = name,
51              AssessedPersonName = employee.SurnameFirstName,
52              FeedbackUrl = reviewMeUrl["FeedbackUrl"] + "/" + assessment.Id,
53              AssessmentDueDate = assessment.AssessmentDueDate,
54              DeclineUrl = reviewMeUrl["DeclineUrl"] + assessment.Id
55          });
56      }
```

Obr č.51 – Metoda NotifyOnOpenAssessment část 1

V první řádě je z databáze podle id vytažen hodnocený zaměstnanec. Ten obsahuje list všech svých ohodnocení, z toho jedno bude otevřené (což bylo provedeno v metodě, která volá tuto). Toto ohodnocení si uložíme do proměnné, protože z něj budou brány data. Důležitý je poté řádek 41, kdy pro každého hodnotitele musíme z databáze získat jeho jméno a email (v tomto případě tako typ tuple). Dále z konfigurace získáme další údaje, vytvoříme předmět a transformujeme data do podoby vhodné pro průchod cyklem. Následně musíme poslat dva druhy emailu, jeden pro hodnotitele a druhý pro PR a další specifické osoby. První sada emailu je posílána na řádku 45 (obr. č.51). Procházíme jednotlivé hodnotitele a posíláme jim email se zvolenou šablonou a daty, které jsme získali v předchozích krocích.

Druhá sada emailu je posílána specifickým osobám, jak bylo zmíněno výše. Na obr. č.52 můžeme vidět zbytek metody.

```
58      var emailDomain :string? = _configuration.GetSection( key: "EmailDomain").Value;
59      var prSessionRecipients = new List<string>
60      {
61          employee.Login + emailDomain,
62          _currentUserService.UserNameWithoutDomain + emailDomain,
63      };
64
65      var subjectPerformanceReview :string = employee.SurnameFirstName + " - " + "Performance review";
66
67      foreach (var prSessionRecipient :string in prSessionRecipients)
68      {
69          await SendTo(EmailTemplate.PerformanceReviewSessionReminder, recipients: new List<string> { prSessionRecipient },
70              subjectPerformanceReview, new PerformanceReviewSessionReminderModel
71          {
72              ReviewersNames = recipientNames,
73              IsAdmin = prSessionRecipient.Contains(_currentUserService.UserNameWithoutDomain),
74              AssessmentUrl = reviewMeUrl["AssessmentUrl"] + "/" + employeeId
75          });
76      }
```

Obr č.52 – Metoda NotifyOnOpenAssessment část 2

Na řádku č.59 sestavíme list osob, kterým má dojít email o budoucím PR. Prozatím jsou zde pouze dvě osoby, list však může být jednoduše rozšířen. Následně na řádku 67 procházíme tento list a posíláme email přesně jako v předchozím případě, se správnou šablonou. Do modelu nyní musíme vložit informaci o tom, jestli přihlášený uživatel je administrátor.

Vidíme, že se jedná pouze o získání a manipulaci s daty pro zaslání emailu. Zbylé metody jsou poté prakticky stejné, pouze se místo PR emailu posílá email pro zrušení ohodnocení.

9.4 RabbitMQ subscriber

Poslední kapitola BE se věnuje modulu pro průběžnou aktualizaci dat, popřípadě pro celkovou synchronizaci. Jak už bylo vysvětleno, RabbitMq poskytuje vlastní .NET integraci, kterou je možné stáhnout jako NuGet. S knihovnou se pracuje dostatečně, ale stále vyžaduje poměrně dost práce při nastavování jednotlivých Exchange, front a modulů pro zpracování zpráv s ohledem na bezpečností politiky. Proto byla snaha najít open-source knihovnu abstrahující velké množství výše zmíněných nastavení. Dalším požadavkem byla snadná integrace s architekturou (nejlépe možnost injektovat službu se zvoleným nastavením), jednoduché ovládání, škálovatelnost a zachování přizpůsobitelnosti. Volbou se stala *RabbitMQ.Client.Core.Dependency*. Jak tedy registrujeme tuto službu a zařídíme, aby se spouštěla okamžitě při zapnutí aplikace a běžela po celou její dobu? Odpovědí je hostovaná služba.

9.4.1 Registrace RabbitMq knihovny a hostovaná služba

Před samotnou implementací bylo potřeba nastavit RabbitMq službu (účty, bezpečnostní politiky, zdroje, práva), viz kapitola 8.2. Výhodou knihovny je, že veškeré nastavení vkládáme do konfiguračního souboru a nastavujeme v *DiConfigRabbitMqConsumer.* Jak tedy tento soubor vypadá vidíme na obrázku níže.

```
18          var rabbitMqSection = configuration.GetSection(key: "RabbitMq");
19          var exchangeSection = configuration.GetSection(key: "RabbitMqExchange");
20
21          services.AddHostedService<RabbitMqConsumerHostedService>();
22          services.AddSingleton<IRabbitMqStateService, RabbitMqStateService>();
23
24          services.AddRabbitMqClient(rabbitMqSection)
25              .AddConsumptionExchange(Constants.ExchangeName, exchangeSection)
26              .AddSingleton<IManageDatabaseService, ManageDatabaseService>()
27              .AddAsyncMessageHandlerSingleton<UpdateMessageHandler<Employee, EmployeeRo>>("public.Update.*")
28              .AddAsyncMessageHandlerSingleton<AddMessageHandler<Employee, EmployeeRo>>("public.Insert.*")
29              .AddAsyncMessageHandlerSingleton<DeleteMessageHandler<Employee, EmployeeRo>>("public.Delete.*")
30              .AddAsyncMessageHandlerSingleton<SynchronizationMessageHandler<Employee, EmployeeRo>>(
31                  "public.GetAllEmployeesResponse");
32
33          services.AddSingleton<ISynchronizeData, SynchronizeDataService>();
34
35          services.AddAutoMapper(configAction: cfg :IMapperConfigurationExpression => { cfg.AddExpressionMapping(); },
36              AppDomain.CurrentDomain.GetAssemblies());
37      }
```

Obr č.53 – DiConfigRabbitMqConsumer

V první řadě musíme načíst konfigurační soubory, které obsahují veškeré nastavení, odpovídající naší architektuře. Autor v dokumentaci uvádí všechny možnosti a přepínače pro většinu operací, které RabbitMq poskytuje. V prvním konfiguračním souboru jsou obecné informace pro bezpečné navázání spojení, ve druhém specifické nastavení pro danou komunikaci.

Jak vypadá tento konfigurační soubor? Na obr. č.54 je zobrazen.

```
33  "RabbitMq": {
34    "HostName": "localhost",
35    "Port": "5672",
36    "UserName": "ReviewMe",
37    "Password": ,
38    "VirtualHost": "HrPortalData"
39  },
40  "RabbitMqExchange": {
41    "Type": "topic",
42    "Durable": true,
43    "AutoDelete": false,
44    "RequeueFailedMessages": true,
45    "RequeueTimeoutMilliseconds": 60000,
46    "RequeueAttempts": 15,
47    "Queues": [
48      {
49        "Name": "EmployeeDataQueue",
50        "RoutingKeys": [ "public.#" ]
51      }
52    ]
53  },
```

Obr č.54 – Konfigurační soubor

V sekci **RabbitMq** se v první řadě nachází adresa a port, na kterém běží naše služba (v tomto případě localhost). Následně jsou potřeba přihlašovací údaje pro naši aplikaci, aby ji RabbitMq dokázal autentizovat. Pro autorizaci je přiložen také virtuální host, který ovlivní viditelnost zdrojů pro naši aplikaci.

V sekci **RabbitMqExchange** poté nastavujeme typ Exchange (topic), jeho vlastnosti (nebude se mazat a bude uchován v paměti) a chování při výskytu chyby (má se zpráva dávat znovu do fronty? Jak dlouho se má čekat? Kolik pokusů?). Vzhledem k tomu, že používáme topic oprávnění, další sekce obsahuje výčet front, včetně povolených klíčů (všechny zprávy, které jsou public). Pokud fronta neexistuje, bude vytvořena. Tyto nastavení jsou dostatečná pro úspěšné získávání zpráv.

Registrace hostované služby

Po načtení konfigurací musíme zaregistrovat další potřebné služby. V první řadě to musí být hostovaná služba, jejichž úkolem je zahájit čtení zpráv z fronty hned při zapnutí aplikace. Základní službou, nutnou pro komunikaci s RabbitMq v rámci naší knihovny, je *IQueueService.* Ta a další (logger, konfigurace) musí být injektována do hostované služby, která dědí od *IHostedService* rozhraní. Díky tomu musíme poskytnou implementaci pro dvě metody. První je událost při spuštění aplikace, druhá při jejím ukončení.

Metody můžeme vidět na obr. č.55 níže.

```
public Task StartAsync(CancellationToken cancellationToken)
{
    var rabbitMqSettingsSection = _configuration.GetSection("RabbitMq");
    var hostName = rabbitMqSettingsSection["HostName"];
    int.TryParse(rabbitMqSettingsSection["Port"], out var port);
    var delay = _configuration.GetValue<int>("TryToConsumeDelay");

    TryStartConsuming(hostName, port, delay);

    return Task.CompletedTask;
}
private void TryStartConsuming(string hostName, int port, int delay)
{
    Task.Run(() =>
    {
        while (IsConsuming == false)
        {
            if (PingHost(hostName, port))
            {
                try
                {
                    _queueService = GetIQueueService();
                    _queueService?.StartConsuming();
                    _logger.LogInformation("RabbitMq started !");
                    IsConsuming = true;
                }
```

Obr č.55 – Hostovaná služba

V první řadě jsou vytaženy konfigurační nastavení z důvodu získání portu, adresy a doby čekání při neschopnosti se spojit s RabbitMq Node. Tyto informace jsou předány funkci *TryStartConsuming.* Problémem, nezbytným k vyřešení, bylo chování aplikace v situaci, kdy se nemůže připojit k RabbitMq službě. Byla tedy implementována již zmíněná funkce, které periodicky posílá ping na danou adresu. V případě úspěšného kontaktu, je získána implementace QueueService a následně je průběžné čtení dat zahájeno. V případě nedostupnosti služby uspíme celý proces a poté započne nový pokus. Při ukončení aplikace je pouze ukončeno čtení dat. Tento způsob ověřování dostupnosti služby se ukázal jako stabilní. V případě neschopnosti čtení při běhu aplikace se o opětovné navázání spojení stará samotná knihovna.

Registrace RabbitMq a komponent pro zpracování zpráv

Kromě registrace stavových (a dalších služeb) musíme zaregistrovat samotného RabbitMq (připojení), Exchange a jednotlivé komponenty pro zpracování zpráv. To můžeme vidět na obr. č.53 na řádku 24. V první řádě přidáme *RabbitMqClient.* Ten se stará o připojení, vytvoření kanálu, zabezpečení apod. Následuje *AddConsuptionExchange,* pro způsob zpracování zpráv. Jako parametr pro obě služby předáme dané konfigurace popsané výše. Následuje služba *IManageDatabaseService*, protože v rámci komponent pro zpracování zpráv potřebuje závislost na databázi. V poslední řadě to jsou samotné komponenty.

Jedná se o službu *AddAsyncMessageHandlerSingleton,* přidanou pro každý druh operace s tabulkou. Jako argument je poté přiložen klíč, aby bylo zřejmé, který typ zprávy bude zpracován kým. Jako generický argument musíme přiložit daný *handler*, který očekává dva vnořené generické parametry. Jedná se o typy pro mapování z RabbitMq vrstvy do Core vrstvy. V poslední řadě je přidána služba pro synchronizaci dat a tím je nastavení dokončené. Jak jsou implementované jednotlivé komponenty pro zpracování zpráv?

9.4.2 Komponenty pro zpracování zpráv

Logika uvnitř těchto komponent je duplicitní, proto byla vytvořena výchozí jednotka se společnou funkcionalitou, kterou můžeme vidět na obr. č.56.

```
8  public class HandlerBase<TCoreType, TRabbitMqType>
9      where TCoreType : IEntity
10     where TRabbitMqType : IEntityRo
11 {
12     protected static IReadOnlyCollection<TCoreType> DeserializeAndMapData(
13         BasicDeliverEventArgs eventArgs,
14         IMapper mapper)
15     {
16         var message = Encoding.UTF8.GetString(eventArgs.Body.Span);
17         var deserializedMessage = JsonSerializer
18             .Deserialize<IReadOnlyCollection<TRabbitMqType>>(message);
19         var coreObject = mapper.Map<IReadOnlyCollection<TCoreType>>(deserializedMessage);
20         return coreObject;
21     }
```

Obr č.56 – HandlerBase

V první řadě si můžeme všimnout typu, který kromě generických parametrů pro mapování obsahuje dvě typové omezení. Následuje metoda pro deserializaci a mapování objektu. Na řádku 16 získáme z příchozího objektu zprávu s daty ve formátu JSON. Následně ji deserializujeme na objekt typu získaného z generiky a namapujeme z RabbitMq formátu na Core formát. Toto je jádro celého procesu.

Každá jednotlivá komponenta musí dědit od výše zmíněné třídy a také od *IAsyncMessageHandler*, která vyžaduje implementaci metody *Handle* s parametrem došlé zprávy. Dále je nutná závislost na objekt pro databázové operace a udržování stavu (např. jestli se nějaký zdroj blokován).

```
29     public async Task Handle(BasicDeliverEventArgs eventArgs, string matchingRoute)
30     {
31         try
32         {
33             if (_rabbitMqStateService.IsBlockedBecauseOfDataSynch)
34             {
35                 _logger.LogError(Messages.InsertBlockBecauseOfDataSynchErrorMessage);
36                 throw new Exception(Messages.InsertBlockBecauseOfDataSynchErrorMessage);
37             }
38             var coreObject = DeserializeAndMapData(eventArgs, _mapper);
39             await _manageDatabase.AddDataAsync(coreObject);
40         }
```

Obr č.57 – AddMessageHandler

Na obr. č.57 můžeme vidět, že pokud RabbitMq právě není blokován kvůli synchronizaci, zavoláme metodu z rodiče popsanou výše a příchozí data uložíme do databáze. Proces pro aktualizaci a smazání dat je prakticky totožný. Rozdílem je právě synchronizace.

9.4.3 Synchronizační služba

Co když se aplikace z důvodu poruchy dostane do nekonzistentního stavu? Administrátor pravděpodobné bude muset provést synchronizaci. Ta je v RabbitMq řešena pomocí RPC. Tento proces bude více vysvětlen v kapitole č.11. Z pohledu ReviewMe bylo potřeba vytvořit službu, která pošle (ne odebírá) požadavek do Exchange a poté v handleru očekává odpověď. Celý proces je synchronní.

```
20  public void SendRequestToRefreshTable(TableName tableName)
21  {
22      var requestCorrelationId :Guid = _rabbitMqStateService.Block();
23
24      var props :IBasicProperties = SetBasicProperties(tableName, requestCorrelationId);
25
26      _queueService.Send(
27          Encoding.UTF8.GetBytes(s: string.Empty),
28          props,
29          Constants.ExchangeName,
30          GetRequestRoutingKey(tableName));
31  }
```

Obr č.58 – SynchronizeDataService – SendRequestToRefreshData

Na obr. č. 58 můžeme vidět jedinou metodu v synchronizační službě, přijímající jako parametr název tabulky pro synchronizaci. Ta na řádku 22 musí vygenerovat jedinečné id pro požadavek, dále nastavit základní vlastnosti zprávy (klíč pro frontu očekávající odpověď a další.). V posledním kroku použijeme *QueueService* pro odeslání zprávy do Exchange. Jako argument poskytneme jeho jméno, dané vlastnosti a klíč pro požadavek (který mikro služba očekává). Tělo může být prázdné.

```
var responseCorrelationId :string? = eventArgs.BasicProperties.CorrelationId;
Guid.TryParse(responseCorrelationId, out var responseCorrelationIdGuid);

if (_rabbitMqStateService.DoesRequestMatchResponse(responseCorrelationIdGuid))
{
    var coreObjects :IReadOnlyCollection<TCoreType> = DeserializeAndMapData(eventArgs, _mapper);
    await _manageDatabaseService.AddNewDataAndUpdateExistingAsync(coreObjects);

    _rabbitMqStateService.UnBlock(responseCorrelationIdGuid);
}
else
{
    _logger.LogError(Messages.RequestDoesntMatchResponseErrorMessage);
```

Obr č.59 – SynchronizationMessageHandler

Na obr. č. 59 je vidět zpracování odpovědí. Pokud se id shodují (požadavek odpovídá odpovědi), zavoláme metodu synchronizace a odblokujeme RabbitMq pro standardní čtení.

10 REVIEWME FE ČÁST

Jedná se o samostatně nasazený Blazor (webassembly) projekt, který pro každou akci volá BE API popsané v předchozí kapitole. Jedním ze základních předpokladů byla jednoduchost (není zde prakticky žádná logika) a nahraditelnost za jiný (např. modernější) FE. Z toho důvodu zde nebude ukázáno vše, pouze podstatné části. Zároveň zde nebudou představeny téměř žádné vizuální výstupy (ty budou shrnuty v kapitole č.12).

10.1 Vrstvy a jejich popis

Samotný projekt je rozdělen na tři části. První je společný model pro FE a BE, aby byla možná komunikace. Druhá jsou jednotlivé Blazor komponenty členěné do atomického designu. Poslední je samotná aplikace, kterou můžeme samostatně spustit (vstupní bod).

10.1.1 ReviewMe.Models

Na obrázku č.60 můžeme vidět souborovou hierarchii pro danou vrstvu.

```
ReviewMe.Models
  Dependencies
  Assessments
    OpenAssessmentRequest.cs
    SaveAdditionalFeedbackRequest.cs
    UpdateAssessmentRequest.cs
  Employees
    Employee.cs
  Enums
    AssessmentReviewerState.cs
  Reviewers
  ReviewerTasks
```

Obr č.60 – ReviewMe.Models

V jednotlivých složkách jsou objekty, které očekáváme od BE jako návratovou hodnotu při různých operacích, popř. které jsou FE posílány jako požadavek. Například při zavolání koncového bodu pro získání zaměstnance je nám vrácen v JSON formátu a automaticky namapován na daný objekt v modelech.

10.1.2 ReviewMe.Frontend

Vstupní bod pro aplikaci, obsahující jednotlivé stránky, služby, statické zdroje a testovací data. Rovněž se zde nachází nastavení frameworku *Tailwind*, který umožňuje nepoužívat CSS při pokročilém stylování. Místo toho jsou styly psány přímo do HTML tagů. To na první pohled může působit chaoticky a nepřehledně, ale zvyšuje to rychlost stylování a

eliminuje některé nevýhody CSS. S využitím Blazor komponentů se mnohonásobně zvýšila čitelnost tohoto zápisu, který bude představen v dalších částech.

Na obrázku č.61 můžeme vidět souborovou hierarchii FE vrstvu.

Obr č.61 – ReviewMe.Frontend

Ve složce *wwwroot* se nachází statické zdroje, jako konfigurační soubory (adresa na BE API, přepínač pro použití testovacích dat), knihovny pro fonty, JavaScript scripty, index.html a další.

Složka *Mock* obsahuje objekt poskytující testovací data pro celou aplikaci. Toto bylo během vývoje velmi užitečné, protože při designu nějakého formuláře není potřeba komunikovat s reálným API. Tato služba je tedy při startu injektována místo pravých služeb, pokud je to nastaveno v konfiguračním souboru.

Středobodem jsou poté *Pages.* Zde se nachází všechny stránky naší aplikace na jednotlivých koncových bodech. Uvnitř, jak bude ukázáno dále, se pomocí služby zavolají data z BE API a předají se jako parametr do komponenty, která je vykreslí.

Zmíněné služby (*Services*) jsou vidět na prostřední a pravé straně obrázku. Jejich základem je http klient, pomocí kterého provedeme požadavek na server, získáme výstup a provedeme navigaci. Můžeme vidět, že služby odpovídají jednotlivým akcím na BE.

Program.cs kromě spuštění aplikace obsahuje funkce pro registraci závislostí na služby výše (popřípadě testovacích služeb), nastavení lokálního uložiště pro JWT token apod.

10.1.3 ReviewMe.Components

Zde najdeme veškeré Blazor komponenty poskládané do atomického designu. Více o AD je zmíněno v kapitole 2.2. Jednotlivé komponenty byly vytvářeny od velmi obecných (atomy), po specifické šablony (templates), ze kterých byly po předání argumentů vykresleny finální stránky. Velmi časté bylo použití funkčních delegátů v rámci argumentů.

Na obrázku č.62 níže můžeme vidět složkovou strukturu pro tuto vrstvu.

```
ReviewMe.Components                    Molecules
  Dependencies                           Modal
  Atoms                                  EmployeeCard.razor
    Alert                                Field.razor
    Button                               Table.razor
    DatePicker                         Organisms
    Accordion.razor                      Page.razor
    EmployeeImage.razor                Templates
    LabeledValue.razor                   AssessmentPageTemplate.razor
    OutsideElementClick.razor            EmployeesPageTemplate.razor
    RadioInput.razor                     ReviewerTasksTemplate.razor
    SectionHeader.razor                  ReviewerTaskTemplate.razor
    SelectInput.razor                  _Imports.razor
    TableCell.razor                    Utilities.cs
```

Obr č.62 – ReviewMe.Components

Vidíme, že vrstva obsahuje čtyři složky pro komponenty v rámci atomického designu (Pages se nachází v předchozí vrstvě). Zajímavý je soubor *Utilities,* který nám umožňuje další způsob, jak znovu používat jednotlivé komponenty.

V další kapitole si ukážeme implementaci stránky pro ohodnocení z pohledu kódu.

10.2 Implementace stránky pro ohodnocení

Implementace všech stránek je principiálně prakticky totožná. Rozdílem je odlišné poskládání komponent do finální podoby. V této kapitole se tedy primárně zaměříme na stránku pro ohodnocení daného zaměstnance. Rovněž bude objasněn způsob vytváření znovu používatelných komponent, práce s *Tailwind* frameworkem a utility.

Začněme tedy se službou pro získání dat z BE API a vytvořením stránky pro ohodnocení.

10.2.1 Získání dat a vytvoření stránky

Získání dat bude vloženo do stejného typu služby, které byly dělány na BE. Základem je tedy rozhraní, které obsahuje prakticky stejné metody, jako na obr. č.43 (kapitola o ohodnoceních). Jak vypadá implementace takové služby vidíme na obr. č.63 níže.

```csharp
public async Task OpenAssessment(int employeeId,
    (DateTimeOffset assessmentDueDate,
    DateTimeOffset performanceReviewDate,
    IReadOnlyCollection<int> reviewers) assessmentData)
{
    var (assessmentDueDate, performanceReviewDate, reviewers) = assessmentData;
    var request = new OpenAssessmentRequest
    {
        AssessmentDueDate = assessmentDueDate,
        PerformanceReviewDate = performanceReviewDate,
        Reviewers = reviewers
    };
    try
    {
        await _httpClientService // IHttpClientService
            .PostJsonAsync($"Assessments/employee/{employeeId}/Open", request); // Task
        _navigationManager.NavigateTo("reviews/employees");
```

Obr č.63 – FE – AssessmentService – OpenAssessment

Služba potřebuje závislost na http klient a navigační manager. V metodě výše jako argument očekáváme všechny data potřebné pro otevření ohodnocení. Ty následně umístíme do společného objektu *OpenAssessmentRequest*. Nakonec v try-catch bloku zavoláme metodu *PostJsonAsync* s danou URI a tělem. Požadavek je poslán, po úspěšném dokončení je uživatel přesměrován na list zaměstnanců.

Tato služba bude injektována do stránky *AssessmentPage* (společně s ostatníma, které jsou potřeba). Jejím základem je specifická instance *AssessmentPageTemplate a* parametry jsou nejen klasické objekty, ale také delegáty na různé funkce (akce). Výstup je viditelný na obr. č.64.

```
8  <AssessmentPageTemplate
9      EmployeeTask="EmployeeTask"
10     ReviewersTask="ReviewersTask"
11     ReviewersFeedbackTask="ReviewersFeedbackTask"
12     OnAssessmentOpen="async assessmentData :(assessmentDueDate,performanceReviewDate,…)
13                         => await AssessmentService.OpenAssessment(Id, assessmentData)"
14     OnAssessmentUpdate="async assessmentData :(assessmentDueDate,performanceReviewDate,…)
15                         => await AssessmentService.UpdateAssessment(Id, assessmentData)"
16     OnAssessmentClose="async () => await AssessmentService.CloseAssessment(Id)"
17     OnAssessmentDelete="async () => await AssessmentService.DeleteAssessment(Id)"
```

Obr č.64 – AssessmentPage

Kromě dalších různých informací jsou zde předány jednotlivé akce ze služby výše, volány v asynchronních lambda metodách. Ostatní data jsou poté získána při inicializaci komponenty.

Jak ale vypadá přímo tato šablona? Jedná se o poměrně velký soubor, jehož rodičovským tagem je organismus Page (ten obsahuje vždy viditelné horizontální menu). Následně je celý rozdělen na HTML sekce podle dané funkce. Na obr. č.65 níže můžeme vidět jednu ze sekcí.

```
<div class="reviewme-p-5 reviewme-space-y-5">
    <LabeledValue Icon="fa-calendar-alt" Title="Assesment due date" Value="@(employee.AssessmentDueDate ==
    <LabeledValue Icon="fa-users" Title="Newly added reviewers" Value="@NewlyAddedReviewersCount.ToString(
    <div class="reviewme-flex reviewme-flex-col reviewme-justify-center reviewme-items-center reviewme-gap
        @if (employee.HasOpenAssessment is false)
        {
            <Button OnClick="OpenAssessment" Variant="ButtonVariant.PinkSolid" Size="ButtonSize.Regular">
                Open assessment
            </Button>
        }
        else
        {
```

Obr č.65 – AssessmentPageTemplate

Jedná se o sekci s plovoucím menu v pravé dolní části obrazovky, obsahující akce s ohodnocením. Je zde viditelné použití Tailwind syntaxe. Ta je zapisována do tříd s předponou ReviewMe. Jedná se o krátké příkazy, které jsou při sestavení nahrazeny CSS (ReviewMe-pt-2 znamená například nastavení vrchního padding na 0.5rem). Na obr. dále vidíme atomy pro label, použití razor syntaxe pro psaní c# kódu a tlačítko s předanou akcí

(rovněž atom), variantou pro vykreslení a velikostí. Tímto způsobem je poté designován i zbytek souboru.

Důležité bylo vytváření malých znovu používatelných komponent jako tlačítko, či label. Jak vypadá jejich implementace?

```
2  <button @onclick="OnClick"
3          disabled="@Disabled"
4          class="@Utilities.ClassNames("reviewme-text-sm  reviewme-font-medium  reviewme-ro
5                         Variant == ButtonVariant.BlueSolid ? "reviewme-bg-blue-500 reviewm
6                         Variant == ButtonVariant.BlueOutline ? "reviewme-bg-white reviewme
7                         Variant == ButtonVariant.PinkSolid ? "reviewme-bg-red-500 reviewme
8                         Variant == ButtonVariant.WhiteOutline ? "reviewme-text-blue-900 ho
9                         Size == ButtonSize.Small ? "reviewme-py-1 reviewme-px-2" : "",
10                        Size == ButtonSize.Regular ? "reviewme-py-2 reviewme-px-4" : "")">
11     @ChildContent
12 </button>
```

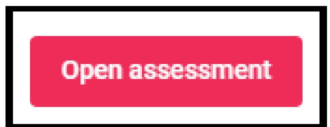
Obr č.66 – Button komponenta

Na obr. č.66 vidíme, že základem komponenty je HTML tlačítko. Tomu jako argument předáme danou akci. Pro vložení designu je použita utility, které předáme výchozí nastavení a nadefinujeme všechny varianty a velikosti (opět pomocí Tailwindu). Ty si uživatel posléze vybírá pomocí výčtového typu.

```
14 @code {
15
16     [Parameter]
17     [EditorRequired]
18     public ButtonVariant Variant { get; set; }
19
20     [Parameter]
21     [EditorRequired]
22     public ButtonSize Size { get; set; }
23
24     [Parameter]
25     [EditorRequired]
26     public RenderFragment ChildContent { get; set; } = default!;
27
28     [Parameter]
29     public EventCallback OnClick { get; set; }
```

Obr č.67 – AssessmentPageTemplate kód sekce

V sekci pro kód (obr.č.67) jsou připraveny jednotlivé vlastnosti, s atributy označujícími parametr, který je povinný. Pro změnu designu se zde nachází varianta a velikost (s příslušnými výčty), kliknutí musí být *EventCallback* a zajímavostí je *ChildContext*, který umožní vložit do tlačítka vlastní fragment pro vykreslení (např. celou div sekci). Na obr. č.68 níže můžeme vidět výsledné tlačítko z obrázku č.65.



Obr č.68 – Open assessment button

11 EMPLOYEEMICROSERVICE ČÁST

Poslední implementační část bude zaměřena na mikro službu EmployeeMicroservice, jejíž hlavním cílem je periodické zasílat data do RabbitMq. Tyto data můžou přicházet z externího zdroje (portálu pro správu zaměstnanců), například pomocí REST API, jiného zprostředkovatele zpráv, popřípadě průběžným čtením (pooling) z databázového *View*. V rámci práce bude použita poslední varianta. Jedním z důvodů, proč je mikro služba důležitá, je zabezpečení dat. Jak bylo vysvětleno v kapitole 8.2.2, zprávy jsou posílány s klíči, obsahujícími úroveň zabezpečení (public, private, protected). Mikro služba ví a má právo rozhodnout, které zprávě bude přiřazena, jaká úroveň. Tím je zajištěno, že ostatní služby v systému budou moci odposlouchávat pouze data, na které mají práva.

Důvodem implementace této služby je primárně vytvořit prototyp odesilatele zpráv (Publisher) stejně, jako v případě ReviewMe BE byl vytvořen prototyp pro odběratele zpráv (Subscriber). Obě komponenty se poté můžou znovupoužít v každé službě pro komunikaci v rámci systému. Tento typ asynchronní komunikace je ve většině případech dostačující (i z pohledu rychlosti). Pokud je rychlost prioritou, je vždy možné použít RPC (synchronní komunikaci), který byl rovněž implementován. Sekundární důvod je ukázka vytvoření jednoduché mikro služby a napojení do systému.

Kód této mikro služby z velké části vychází z BE ReviewMe. Repositáře, notifikace a RabbitMq komponenty (včetně všech použitých knihoven) jsou založeny na stejných principech. Proto v této kapitole nebudou více rozepisovány.

11.1 Vrstvy a jejich popis

Rozložení vrstev je prakticky totožné jako na BE části ReviewMe. Vrstva pro **API** obsahuje pouze jeden koncový bod, sloužící pro synchronizaci dat z externí služby. Její implementace se nachází v **Core** vrstvě, stejně jako jediná entita zaměstnanec. Druhá a poslední služba poskytuje načtení a uložení testovacích dat. Další v řadě jsou tři **Infrastructure** vrstvy, jedna pro vlastní databázi mikro služby, druhá obsahující pouze *View* z externí databáze a třetí je RabbitMq Publisher. Poslední vrstvou je **Jobs**. Jedná se o úlohy v pozadí spuštěné souběžně s aplikací, které kontrolují změnu v databázovém View a následně jí propíší do vlastní databáze.

Na obrázku níže můžeme vidět jednotlivé vrstvy.

Search Solution Explorer (Ctrl+;)
Solution 'EmployeeMicroservice' (6 of 6 projects)
EmployeeMicroservice.API
EmployeeMicroservice.API.Core
EmployeeMicroservice.Infrastructure.DbStorage
EmployeeMicroservice.Infrastructure.DbStorageHR
EmployeeMicroservice.Infrastructure.RabbitMQProducer
EmployeeMicroservice.Jobs

Obr č.69 – EmployeeMicroservice

Kromě odesilatele zpráv jsou vrstvy velmi malé. V následující kapitole se podíváme na synchronizaci dat.

11.2 Synchronizace dat

Data v mikro službě je potřeba udržovat aktuální. Toho je dosaženo pravidelnou synchronizací databáze EmployeeMicroservice s daty v databázovém pohledu (View). Ten nám při změně dat v externí databázi poskytne pohled na data, která můžeme vzít a pře uložit. Aktualizaci dat můžeme vynutit pomocí API, ale většina bude prováděna automatizovaně v podobě úlohy v pozadí, která volá stejnou službu, jako API. Jak je tato služba implementována?

V podstatě velmi jednoduše, získáme aktuální data jak z naší databáze, tak z pohledu na tu externí. Služba tedy potřebuje reference na obě databáze a rovněž na RabbitMq. Následně jsou procházeni existující zaměstnanci a porovnávání podle id z databázového pohledu. Pokud zde nejsou v databázi, jsou přidání, pokud se liší v nějaké vlastnosti, jsou aktualizování. Jestli zcela chybí, je jejich záznam odstraněn. Souběžně jsou přidáváni do odpovídajících listů, aby mohli být předání odesilateli zpráv. To můžeme vidět na obrázku níže.

```
if (insertedEmployees.Any())
{
    const string routingKeyInsertEmployees = "public.Insert.Employees";
    await _messageSender // IMessageSender
        .SendMessage(body: JsonSerializer.Serialize(insertedEmployees), routingKeyInsertEmployees);
}
```

Obr č.70 – ManageService

Kód je velmi přímočarý. Vytvoříme konstantu s klíčem pro veřejná data. Následně zavoláme metodu pro odeslání zprávy s parametrem serializovaného listu nových zaměstnanců a daným klíčem. Jak vypadá implementace odesilatele zpráv se dozvíme v další kapitole.

11.3 RabbitMq publisher

Vrstva pro zasílání zpráv má velmi podobnou strukturu jak odběratel na BE. Stejně jako u něj je potřeba hostovaná služba potřebná pro zahájení činnosti vrstvy ihned při startu mikro služby. Veškeré nastavení je opět vloženo do konfiguračního souboru a nastaveno v rámci souboru pro injektování závislostí, který můžeme vidět níže.

```
18        var rabbitMqSection = configuration.GetSection(key: "RabbitMq");
19        var exchangeSection = configuration.GetSection(key: "RabbitMqExchange");
20
21        services.AddRabbitMqClient(rabbitMqSection)
22            .AddConsumptionExchange(RabbitMqConstants.ExchangeName, exchangeSection)
23            .AddAsyncNonCyclicMessageHandlerSingleton<SynchRequestHandler<Employee, EmployeeRo>>(
24                "public.GetAllEmployees")
25            .AddHostedService<HrPortalMicroserviceHostedService>();
26
27        services.AddScoped<IMessageSender, MessageSender>();
```

Obr č.71 – RabbitMQProducerDiConfig

V první řadě vytáhneme konfigurační soubor, a vložíme ho jako parametr metodě pro přidání RabbitMq klienta na řádku 21. Následně zavoláme metodu *AddConsumptionExchange*, která přidá (popřípadě aktivuje) Exchange pro zasílání zpráv. Po vložení hostované a *MessageSender* služby (řádek 25 a 27) je nyní možné odesílat zprávy. V případě RPC požadavku musí služba rovněž odposlouchávat, proto je na řádku 23 přidána komponenta pro zpracování zpráv.

Na obr.č.72 níže vidíme službu pro odesílání zpráv.

```
7   internal class MessageSender : IMessageSender
8   {
9       private readonly IQueueService _queueService;
10
11      public MessageSender(IQueueService queueService)
12      {
13          _queueService = queueService;
14      }
15
16      public async Task SendMessage(string body, string key)
17      {
18          await _queueService.SendJsonAsync(
19              body,
20              RabbitMqConstants.ExchangeName,
21              key); // Task
22      }
```

Obr č.72 – MessageSender

Implementace je velmi jednoduchá. V podstatě jen injektujeme závislost na *QueueService,* nejdůležitější objekt při práci s RabbitMq pomocí zvolené knihovny. V metodě pro zaslání zpráv (řádek 16) poté pošleme zprávu ve formátu JSON s danými parametry.

Poslední složkou je komponenta pro zpracování RPC požadavků. Základní myšlenkou tohoto procesu je, že ReviewMe BE zašle zprávu do Exchange se speciálním klíčem a jedinečným Id. To je velmi důležité, podle něj můžeme přiřadit jednotlivé požadavky k odpovědím. Zvolená podoba klíče je rovněž podstatná a ovlivní, jakým řadičem bude zpráva zpracována. Posledním parametrem je jméno fronty, ve které má BE očekávat odpověď. Mikro služba poté přijme požadavek, vytvoří novou zprávu s požadovanými daty včetně předání požadovaných vlastností (zmíněných výše). Nakonec ji zakóduje a odešle pomocí služby popsané výše.

Jak vypadá implementace komponenty pro zpracování zpráv?

```
30  public Task Handle(BasicDeliverEventArgs eventArgs, string matchingRoute, IQueueService queueService)
31  {
32      using var scope = _serviceProvider.CreateScope();
33      var genericRepositoryIDS =
34          scope.ServiceProvider // IServiceProvider
35              .GetRequiredService<IGenericRepositoryHRService>();
36
37      var replyProps :IBasicProperties? = queueService.ConsumingChannel.CreateBasicProperties();
38      replyProps.CorrelationId = eventArgs.BasicProperties.CorrelationId;
39      replyProps.ReplyTo = eventArgs.BasicProperties.ReplyTo;
40
41      var entities :IReadOnlyCollection<TCoreType> = genericRepositoryIDS.GetAllRecords<TCoreType>();
42
43      var rabbitMqEntities = _mapper.Map<IReadOnlyCollection<TRabbitMqType>>(entities);
44
45      var rabbitMqEntitiesSerialized :string = JsonSerializer.Serialize(rabbitMqEntities);
46
47      var encodedBody :byte[] = Encoding.UTF8.GetBytes(rabbitMqEntitiesSerialized);
48      queueService.Send( encodedBody, replyProps, RabbitMqConstants.ExchangeName, routingKey: replyProps.ReplyTo);
```

Obr č.73 – SynchRequestHandler

V první řade (na obr. č.73) je potřeba získat službu pro práci s databází mikro služby, aby bylo možné obsloužit požadavek. Poté, na řádku 37, vytvoříme objekt pro uložení metadat. Do něj vložíme korelační Id a název fronty pro odpověď. Poté na řádku 41 získáme všechny záznamy z tabulky na základě typu, získaného podle příchozího klíče. Ty jsou namapovány, serializovány, kódovány a poslány.

12 TESTOVÁNÍ, VYHODNOCENÍ A POUŽITÍ APLIKACE

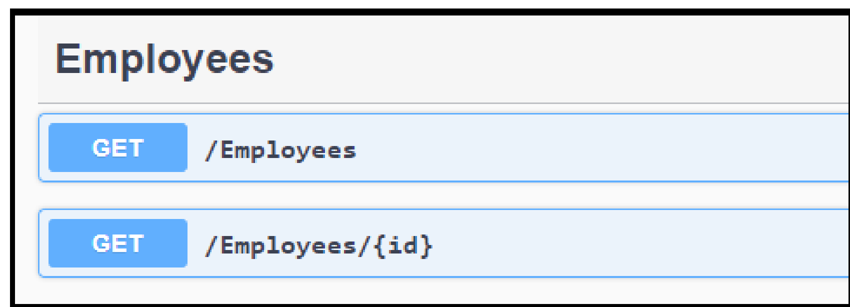
Poslední kapitola je zaměřena na výstupy práce. Jednotlivé části zde budou zobrazeny, vyhodnoceny a otestovány. Testování z pohledu uživatele probíhalo manuálně, ověřením jednotlivých požadavků. Při vývoji poté hrály velkou roli jednotkové testy, tvořené pro každou přidanou business logiku. V rámci kapitol bude zobrazeno splnění jednotlivých uživatelských požadavků. Závěr kapitoly je poté věnován potenciálu aplikace a jejího použití v praxi, včetně všech vytvořených služeb a komponent.

Nejdůležitější částí práce je BE část ReviewMe. Začneme tedy jí.

12.1 ReviewMe BE

Výstupem této části je samostatně nasazená služba, poskytující veřejné API. Obsahuje veškerou logiku pro zadávání zpětné vazby (zaměstnanec), zobrazení úkolu pro ohodnocení, vytváření ohodnocení (admin), zobrazení listu zaměstnanců, synchronizace a notifikace. Jádro aplikace je rovněž nezávislé na externích poskytovatelích, kteří můžou být jednoduše nahrazeni bez nutnosti přepsat stávající logiku. Rovněž byla snaha tvořit aplikaci v rámci SOA (kombinace s Microservice) architektury, komunikující v systému pomocí zprostředkovatele zpráv. Bylo provedeno příslušné nastavení a vytvořena požadovaná vrstva pro odposlouchávání dat a provedení synchronizace. Všechny zmíněné části musely být otestovány.

Do služby pro vizuální zobrazení API byl přidán *Swagger*. Ten umožňuje jednoduše posílat požadavky na dané API z přehledného UI. Vzhledem k samostatnosti BE, byla tímto způsobem testována veškerá business logika. V první fází se připravily data v databázi. Poté byl poslán požadavek na daný koncový bod s předpřipraveným tělem. Odpověď se musí rovnat požadovanému výstupu. Na obr. č.74 můžeme vidět koncové body pro zaměstnance.



Obr č.74 – Koncové body pro zaměstnance

Vidíme, že veškeré operace s aplikací jsou zde seřazené do sekcí, podle kontrolérů.

Zároveň pro každý koncový bod můžeme vidět jeho typ (GET, POST, PUT, DELETE) a cestu, včetně parametrů. Po rozkliknutí (např. Employees) dále můžeme vidět kódy odpovědí a tělo, které vyžaduje, nebo vrací. Zároveň můžeme kliknout na tlačítko *Try it out* a poslat náš požadavek. Na obr. č.75 můžeme vidět výstup po zavolání dané operace.

Curl

```
curl -X 'GET' \
  'https://localhost:5001/Employees' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:5001/Employees
```

Server response

Code Details

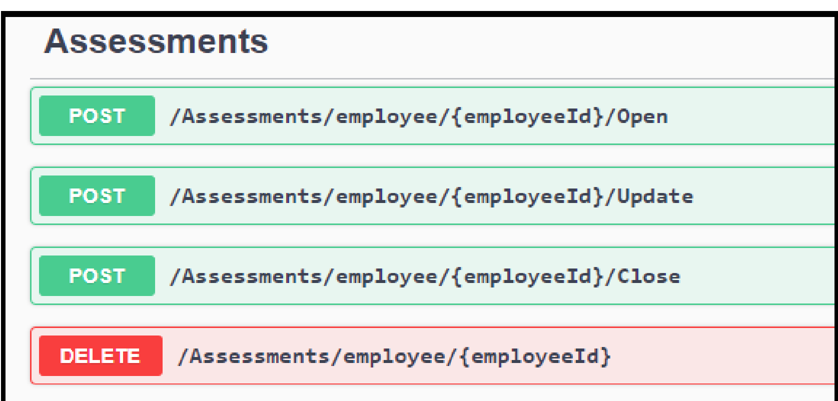
200 Response body


```
  {
    "id": 110,
    "surnameFirstName": "Polivka, Bolek",
    "department": "DotNet",
    "location": "Ostrava",
    "position": "Developer",
    "imageSrc": "https://localhost/persons/login/polivka/photo",
    "teamLeaderName": "",
    "performanceReviewMonth": "0001-01-01T00:00:00+00:00",
    "assessmentDueDate": "0001-01-01T00:00:00+00:00",
    "performanceReviewDate": "0001-01-01T00:00:00+00:00",
    "hasOpenAssessment": false,
    "additionalFeedback": ""
  },
  {
    "id": 115,
    "surnameFirstName": "Wolfeschlegelsteinhausenbergerdorff, Hubert",
```


Obr č.75 – Výstup koncového bodu Employees

Kromě samotného požadavku vidíme list všech zaměstnanců z databáze. Jedná se pochopitelně o testovací data, která ale už dorazila do služby zprostředkovatelem zpráv. V této fázi už se tedy simuluje celkový koloběh dat, od získání z externí služby a odeslání do sběrnice, po příchod do ReviewMe. List obsahuje všechna data, které by FE část mohla potřebovat při zobrazení listu zaměstnanců. Tímto způsobem byl testován zbytek aplikace.

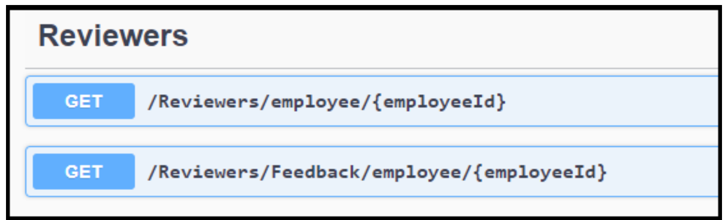
Na obrázku č.76 vidíme koncové body pro operace s ohodnocením.



Obr č.76 – Koncové body pro ohodnocení

Jednotlivé metody odpovídají těm, popsaným v BE části. Důležité bylo umožnit volání metod jenom v určitém pořadí (není žádoucí volat smazání ohodnocení, když nebylo ještě otevřeno). Kromě zavření ohodnocení, každá z metod musí jako vedlejší efekt poslat emailovou notifikaci. Hlavní je poté validace, modifikace a uložení dat do DB. To vše bylo potřeba ověřit.

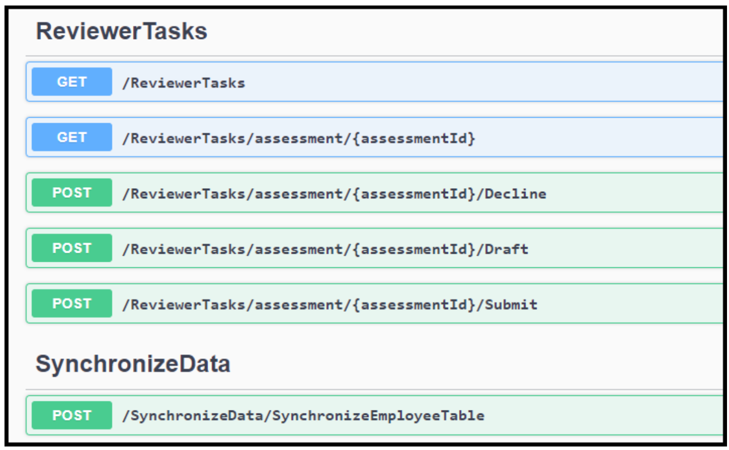
Další v pořadí jsou koncové body pro získání kolegů daného zaměstnance (obr. č.77), kteří budou požádání o hodnocení.



Obr č.77 – Koncové body pro získání kolegů

První případ vrátí kolegy podle id zaměstnance (vráceny jsou testovací data), druhý vrátí list zpětných vazeb od kolegů.

Poslední jsou koncové body pro hodnocení kolegů z pohledu zaměstnance (obr č.78). Ty, jak jsme viděli v BE části, jsou poměrně jednoduché. V podstatě pouze validují vstup, změní příslušný stav a uloží do databáze. Rovněž zde záleží na pořadí, testování bylo tedy podobné jako v případě vytváření ohodnocení. Kromě akcí musí být pochopitelně možnost získat seznam všech rozpracovaných úkolů.



Obr č.78 – Koncové body pro hodnocení a synchronizaci

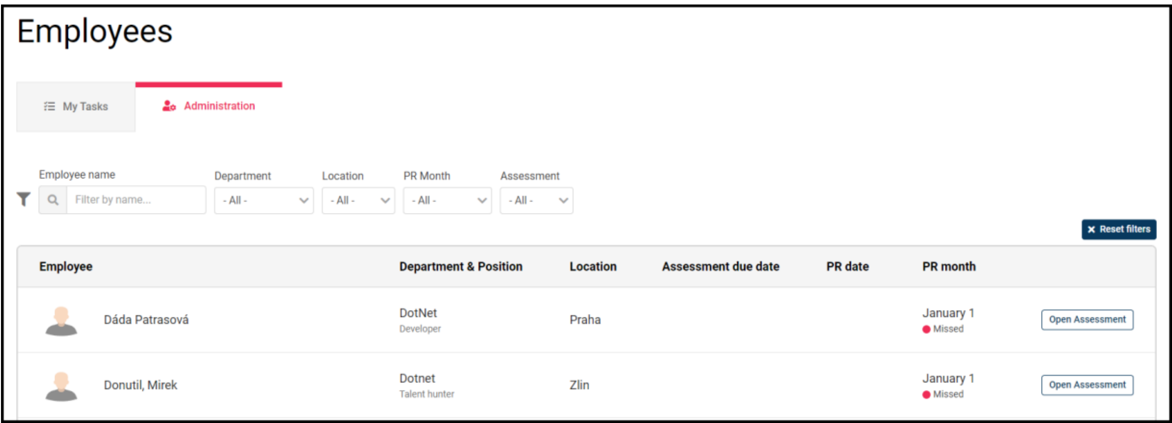
Průchod celou aplikací (včetně zbývajících požadavků) bude dále ukázán na FE části.

12.2 ReviewMe FE

Jak bude vypadat výsledný uživatelský zážitek při používání aplikace? Na to se pokusíme zodpovědět v této kapitole. Začneme v administrátorské sekci.

12.2.1 List zaměstnanců

Po tom, co administrátor klikne na příslušné menu, bude přesměrován do sekce obsahující list zaměstnanců, jehož součástí jsou kromě dodatečných informací i operační tlačítka, a to pro otevření a editaci ohodnocení. Nad listem se poté nachází filtrační možnosti podle zvolených dat. Výsledek můžeme vidět na obr. č. 79.



Obr č.79 – List zaměstnanců

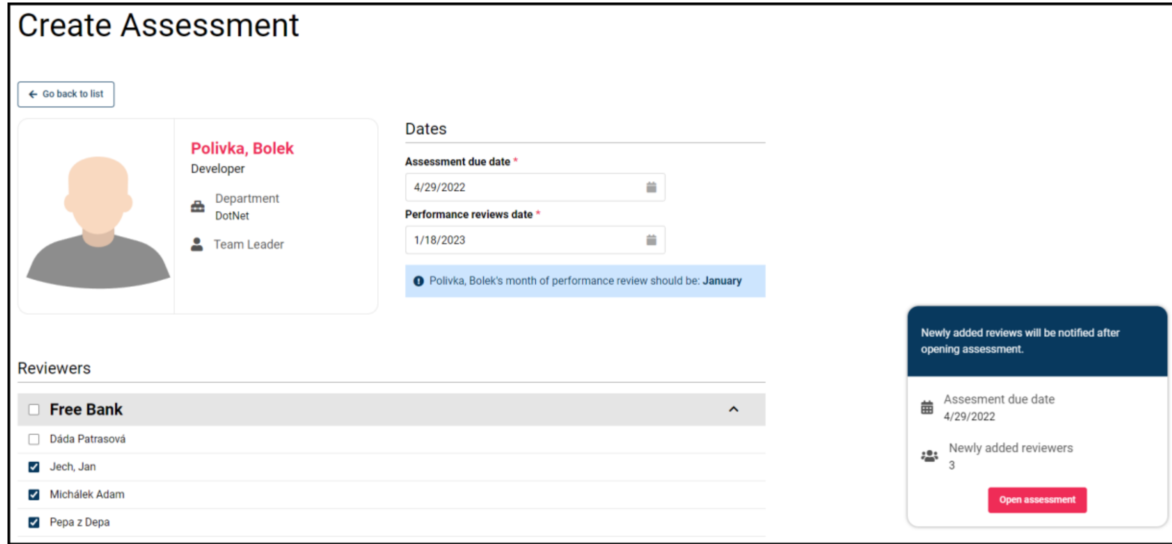
UI a UX bylo konzultováno s lidmi z praxe. Cílem byla jednoduchost a přehlednost. Administrátor tedy přijde do této sekce, a v prvním kroku potřebuje vyhledat uživatele, u kterého je nutné posbírat ohodnocení, protože se mu například blíží PR. Po nalezení klikne na tlačítko pro otevření ohodnocení (popřípadě editaci) na pravé straně obrazovky. Tím se dostáváme do sekce pro vytváření ohodnocení.

12.2.2 Vytváření ohodnocení

Kromě vytváření zde můžeme provádět další operace s ohodnocením. Vždy záleží, v jakém stavu jej otevřeme (např. v případě editace budou menší rozdíly).

Stránka kromě základních informací o zaměstnanci obsahuje formuláře pro dva datumy. Jeden je pro PR, druhý uzávěrka pro sbírání zpětné vazby. Platí pro ně různá validační pravidla, které vynucuje sám FE, před zasláním na BE. Dolní část poté obsahuje seznam projektů daného zaměstnance a v každém je list jeho kolegů. Administrátor poté na základě interní znalosti označí kolegy, kteří mají provést hodnocení. V pravé dolní části je operační

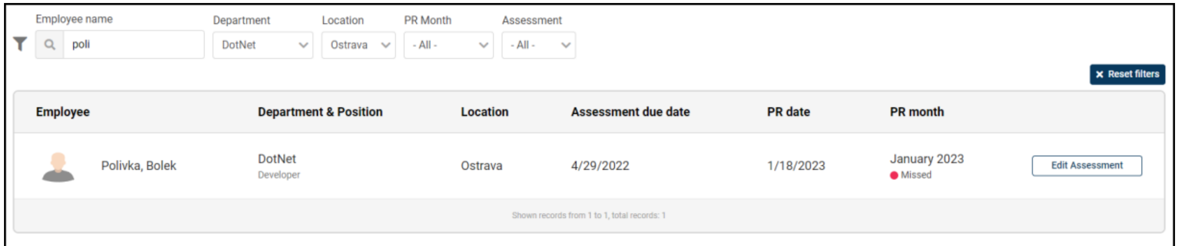
panel, zobrazující stav, datum, počet hodnotitelů a tlačítko pro provedení operace. Na obrázku č. 80 můžeme vidět vytvoření ohodnocení.



Obr č.80 – Vytváření ohodnocení

Může zde vzniknout otázka, proč se žádost o ohodnocení neposílá všem dostupným kolegům (což by zautomatizovalo celý proces). Odpovědí je velikost teamů. Průměrný zaměstnanec pracuje pravidelně s 5 až 10 kolegy, kteří jsou schopní provést realistické ohodnocení. Ostatní kolegové by ho neprovedli, a pokud ano, tak ne přesně (pravděpodobně). Přehlcení zaměstnanců žádostmi o ohodnocení by poté mohlo vést k averzi a vyhýbání.

To, co se moc nepovedlo, je panel v pravé části obrazovky. Problémem je hlavně responzivita a obecně jeho umístění mohlo být promyšleno lépe. Po kliknutí na tlačítko pro otevření ohodnocení s vyplněnými daty jsme přesměrováni zpátky na list zaměstnanců.

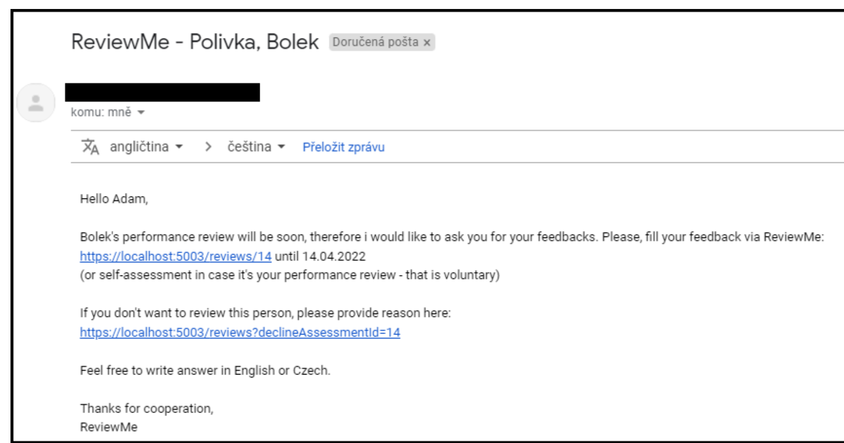


Obr č.81 – Vyfiltrovaný zaměstnanec s otevřeným ohodnocením

Na obr. č. 81 vidíme vyfiltrovaného zaměstnance, na kterého bylo v předchozím kroku otevřeno ohodnocení. Kromě vyplněných datumů se změnila podoba tlačítka, které nyní umožní editovat ohodnocení.

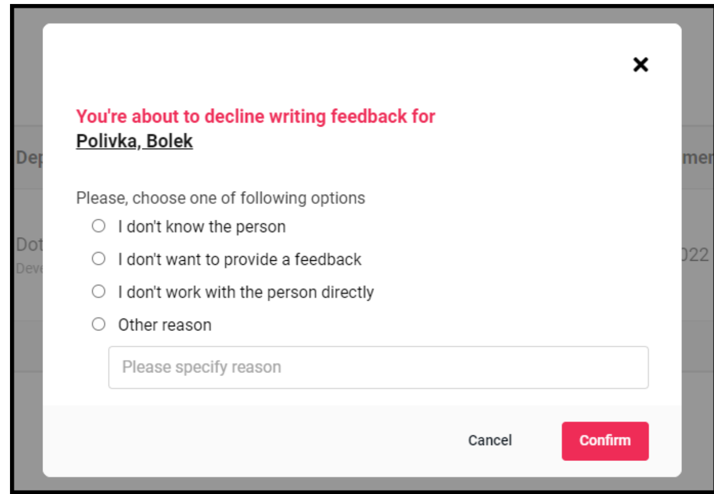
12.2.3 Zaslané notifikační emaily

Jak bylo popsané v předešlých kapitolách, po otevření ohodnocení (popřípadě editaci a smazání) jsou poslány notifikační emaily. Jejich návrh byl velmi podstatný, protože to bude vstupní bod do aplikace pro většinu zaměstnanců. Podoba a obsah emailů byly rovněž konzultovány. Na obrázku č. 82 můžeme vidět email s výzvou pro podání zpětné vazby na kolegu.



Obr č.82 – Notifikační email pro ohodnocení

Hlavička obsahuje zdrojovou aplikaci a hodnoceného člověka. Kromě osobního pozdravu tělo obsahuje tři podstatné informace. První je datum do kdy můžeme hodnotit. Druhý je odkaz přímo do aplikace, kde můžeme vyplnit ohodnocení. Třetí je dialog pro zamítnutí ohodnocení s poskytnutím důvodu. Po přechodu na odkaz se nám zobrazí následující dialog (obr. č.83).

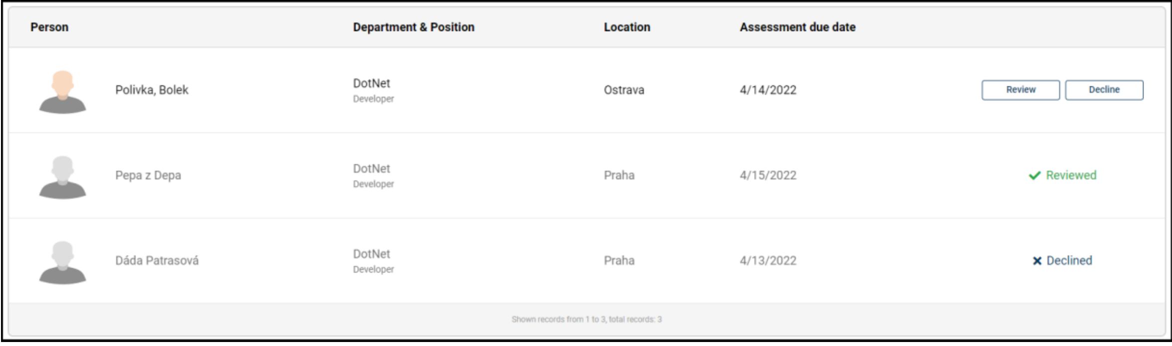


Obr č.83 – Dialog pro odmítnutí ohodnocení

Po vybrání důvodu jsme přesměrování do aplikace.

12.2.4 List úkolů pro ohodnocení

Používání aplikace z pohledu zaměstnance (hodnotitele) je velmi jednoduché. Ten může provést pouze dvě akce, zobrazit si list svých úkolů (včetně historie) a provést hodnocení. Na obrázku č. 84 můžeme vidět zmíněný list.



Person	Department & Position	Location	Assessment due date	
Polivka, Bolek	DotNet Developer	Ostrava	4/14/2022	Review Decline
Pepa z Depa	DotNet Developer	Praha	4/15/2022	✓ Reviewed
Dáda Patrasová	DotNet Developer	Praha	4/13/2022	× Declined

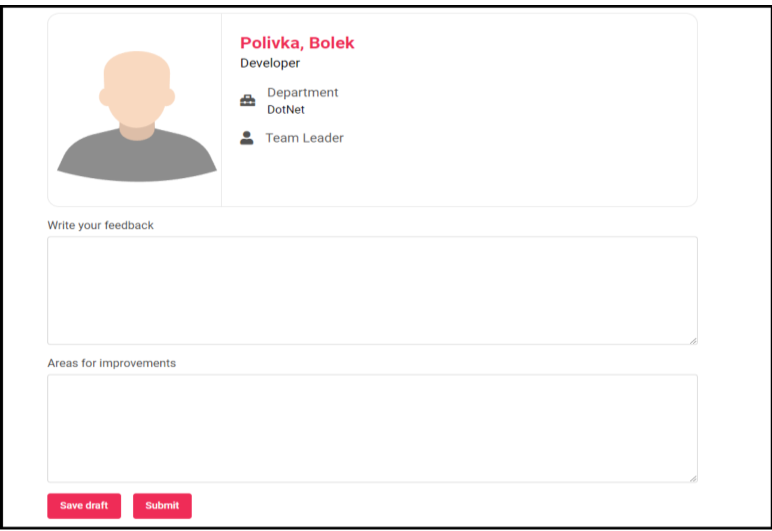
Shown records from 1 to 3, total records: 3

Obr č.84 – List hodnotících úkolů s historií

Jedná se o podobný list jako na obr. č. 81 (s odlišnými daty). Obsahuje pouze informace, které jsou relevantní pro provedení hodnocení, tedy jméno, pozice, a hlavně finální datum. Operační panel obsahuje možnost jít hodnotit, popřípadě zamítnout (zde se otevře stejný dialog jako na obr. č.83). Kolegové, kteří jsou již ohodnocení, zamítnutí, popřípadě vypršel termín pro hodnocení, jsou šedí. Důvod je poté zobrazen místo tlačítek.

12.2.5 Vyplnění zpětné vazby

Na obr č. 85 můžeme vidět velmi jednoduchou stránku pro vyplnění ohodnocení.

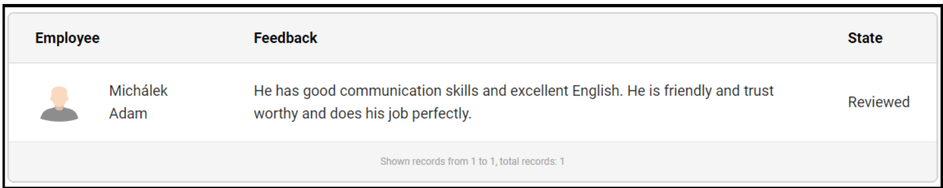


Obr č.85 – Provedení ohodnocení

Zahrnuje pouze dva textové formuláře a tlačítko pro odeslání, popřípadě pozdní vyplnění.

12.2.6 Zobrazení ohodnocení administrátorem

Po sesbírání zpětné vazby je nutné ji vyhodnotit (popř. shromáždit) před PR schůzkou. Proto byla do sekce pro jednotlivé ohodnocení (u konkrétního zaměstnance) přidána tabulka zobrazující zpětnou vazbu. Na obr. č. 86 můžeme vidět její podobu.



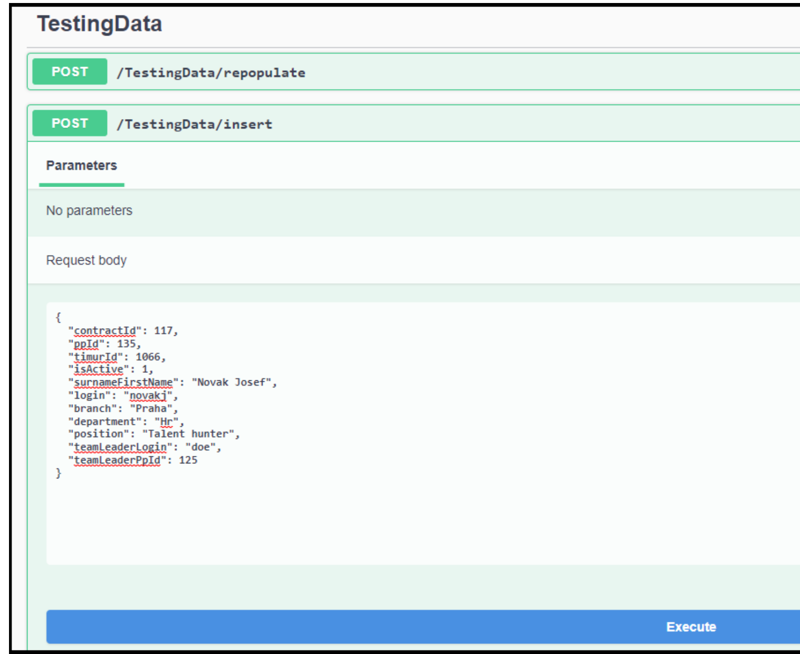
Obr č.86 – Zobrazení zpětné vazby

Tabulka obsahuje všechny zaměstnance, kteří měli provést ohodnocení. To potom můžeme vidět ve sloupci *Feedback*. Ohodnocení může být poměrně dlouhé, proto se zobrazuje pouze jeho zkrácená verze. Po kliku na něj si můžeme načíst celou jeho podobu (otevře se dialog). Kromě jména (*Employee*) zaměstnance je zde stav hodnocení (*State*). V případě odmítnutí se důvod vloží do sloupce pro zpětnou vazbu.

12.3 Testování distribuce dat

Kritická část pro testování byla distribuce dat v systému pomocí RabbitMq (v rámci SOA architektury). Bylo potřeba zodpovědět otázky ohledně spolehlivosti, a hlavně bezpečnosti dat uživatelů (i v rámci GDPR). Do této sekce rovněž spadá vytvořená mikro služba fungující jako zdroj externích dat, kterým přidá bezpečnostní kontext pro další distribuci. Testovací proces byl podstatně zjednodušen nástroji v rámci RabbitMq balíčku. Jedná se primárně o UI Management plugin a další. Rovněž bylo vyvinuto několik testovacích služeb, které simulují určité situace.

Jako ukázku můžeme nasimulovat standardní situaci, kdy tabulka v externí databázi bude rozšířena o zaměstnance. Ten by poté měl být reflektován v databázovém pohledu, ze kterého v pravidelných intervalech čte komponenta naší mikro služby. Pokud detekuje změnu, načte ji, uloží do databáze a pošle do systému s daným bezpečnostním klíčem. Na obr. č. 87 můžeme vidět koncový bod simulující vložení dat do externí databáze.



Obr č.87 – Mikro služba – testování distribuce dat

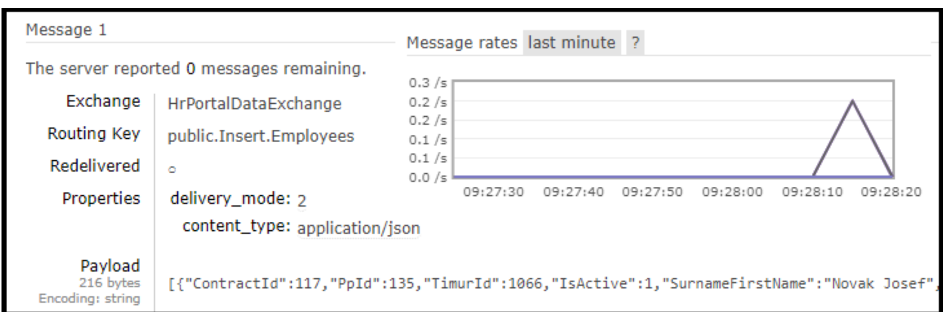
Vidíme, že byly přidány dva koncové body. V tomto testovacím případu zkusíme vložit jediného zaměstnance, který by měl projít přes celý systém. Kromě metod na obrázku mikro služba obsahuje také koncový bod pro synchronizaci s databázovým pohledem externí služby.

Po vložení můžeme vidět, že Job detekuje změnu a aktualizuje databázi mikro služby. Na obr. č. 88 vidíme výpis z logu.

```
09:34:07 INF] PlusPortal synchronization was finished. Inserted employees: 1. Updated employees 0. Deleted employees 0
09:34:07 INF] A new message received with deliveryTag 1.
09:34:07 INF] Message processing finished successfully. Acknowledge has been sent with deliveryTag 1.
```

Obr č.88 – Mikro služba – výpis logů při aktualizaci a zaslání zprávy

Následně dojde k odeslání zprávy dále do systému. Po vypnutí služeb odposlouchávající tuto zprávu ji můžeme rovněž detekovat a zobrazit v UI RabbitMq (obr č. 89).



Obr č.89 – RabbitMq – zachycení odeslané zprávy

Na obrázku vidíme všechny potřebné informace o zprávě, jako klíč, tělo apod. S ní rovněž můžeme různě manipulovat a testovat další stavy i v rámci zabezpečení (např. reakce při zaslání do uzlu, který nemá právo číst).

Posledním článkem řetězce je odposlech zprávy ReviewMe službou (obr. č. 90).

```
[09:43:57 INF] RabbitMq started consuming massages !
[09:43:57 INF] RabbitMq started consuming massages !
[09:43:57 INF] A new message received with deliveryTag 1.
[09:43:57 INF] A new message received with deliveryTag 1.
[09:43:57 INF] Executed DbCommand (40ms) [Parameters=[@p0='1
 09:43:57 INF] Message processing finished successfully. Ack
 09:43:57 INF] Message processing finished successfully. Ack
```

Obr č.90 – ReviewMe BE – zachycení odeslané zprávy

Ta příjme zprávu (klíče se rovnají), zpracuje ji, uloží do databáze a pošle odpověď s potvrzením doručení.

Podobným způsobem byly poté testovány i další situace, příkladem může být celková synchronizace dat.

12.4 Použitelnost aplikace v praxi

Poslední kapitola se věnuje použitelnosti aplikace v praxi. To byl také jeden z hlavních cílů práce. Celý proces, od sběru požadavků, přes výběr technologií a návrh architektury, po samotnou implementaci a testování byl konzultován odborníky z praxe. Nemělo by se tedy stát, že některá část je naprosto nepoužitelná.

BE část ReviewMe je rozhodně použitelná v praxi. Business logika aplikace je prakticky kompletní a uživatelské požadavky byly splněny. Jednotlivé části jsou jednoduše rozšiřitelné. Pokud by například byl požadavek o rozšíření možností při vyplnění zpětné vazby, výsledná práce by nezabrala moc času. Jedním z důvodů je znovu používatelnost většiny komponent a snaha udržet čistotu kódu a přehlednost celého řešení. To je zároveň nezávislé na externích službách, což je důležitý faktor pro dlouhodobou udržitelnost. Pokud by firma nebyla spokojena s notifikacemi, popřípadě pravidelnou aktualizací dat (požadavky se můžou v každé firmě lišit), může jednoduše stávající řešení upravit.

FE část by potřebovala nějaké úpravy z hlediska UI a UX, obecně je ale použití jednoduché a přímočaré, což byl účel. Díky nahraditelnosti FE je rovněž pro každou firmu jednoduché si vytvořit vlastní design, případně upravit stávající.

Mikro služba už může být čistě specifická pro firemní systém. Ten často v případě podobné architektury potřebuje službu, zodpovědnou za distribuci dat z externích služeb (mimo firmu), včetně přidání bezpečnostního kontextu (jaká služba má práva na které data). To je důvodem vzniku této mikro služby. Její návrh je tedy velmi generický a většina součástí se bude hodit i v jiných službách (např. odesilatel zpráv).

Vložení všech služeb do SOA architektury byl krok, který rovněž velmi závisí na konkrétním firemním prostředí. Rozhodně to v krátkodobém horizontu vyžaduje více práce (zvláště při již existujícím systému), v dlouhodobém pak může pomoci se škálovatelností, bezpečností dat, jednotností komunikace a přehledností celkového řešení. Cenou je samozřejmě nastavování zprostředkovatele zpráv (RabbitMq), návrh bezpečnostních standardů a implementace Pub/Sub pro všechny strany, které chtějí komunikovat. V konečném důsledku navržené řešení může být velmi užitečné v praxi, jeho implementace však záleží na rozhodnutí firmy.

ZÁVĚR

Všechny cíle práce a jednotlivé body zadání se povedlo splnit. V teoretické části byla nastudována potřebná problematika, týkající se primárně prostředí pro vývoj, zabezpečení, čistého kódu, vývojových architektur, zprostředkovatele zpráv a DevOps. Tyto informace byly následně použity v každé fázi vývoje zadané aplikace, od získávání a zapisování uživatelských požadavků, po následnou implementaci a testování. Splnění požadavků bylo konzultováno s lidmi, kteří by aplikaci potenciálně mohli používat.

Hlavním výstupem práce je funkční aplikace pro hodnocení členů teamu, vložená do systému v rámci navržené SOA architektury. Tento krok byl velmi důležitý, protože se počítalo s rozšiřováním firemního ekosystému dalšími službami. Ty nyní budou mít jednotný a jednoduchý způsob, jak spolu efektivně sdílet prostředky a komunikovat. Důraz byl zároveň kladen na použitelnost v praxi. Samotná aplikace je poté rozdělená na FE a BE část a nasazena zvlášť. BE část je tedy samostatná služba, poskytující veřejné API pro komunikaci. Hlavním snahou při implementaci byla nezávislost na externích technologiích a zapouzdření business logiky tak, aby jádro aplikace zůstalo vždy zachováno, například při výměně databázového poskytovatele. Služba dále zahrnuje zasílání notifikací, které se ukázalo jako velmi podstatné, protože v praxi pro většinu zaměstnanců je vstupním bodem do aplikace právě email s odkazem na FE část, kde je provedeno hodnocení. Aktualizace dat pomocí zprostředkovatele zpráv (a příslušné vrstvy, která provede odposlech) byla další velmi důležitá část. Právě zde je řešena otázka zabezpečení dat a jejich distribuce v celém systému. Vzhledem k povaze aplikace je možnost nabízet ji jiným firmám, a to bez nutnosti použití FE části, která je rovněž součástí této práce. Při vytváření UI byl brán v potaz uživatelský zážitek, který byl konzultován s potenciálními koncovými uživateli z praxe.

Sekundárním, ale velmi důležitým cílem práce byl návrh firemní architektury, která se může skládat z velkého množství služeb. Architektura byla v podobě SOA filozofie aplikována na každou službu v systému. Další inspirací byla Microservice architektura. Na jejím základě je vytvořená mikro služba určená pro distribuci externích dat. Ta získává data z externího portálu, přiřazuje jim úroveň zabezpečení (v rámci ochrany dat) a posílá dále do zprostředkovatele zpráv.

Aplikace má mnoho prostoru pro rozšíření. Patří sem přidání dalších formulářů do sekce pro podání zpětné vazby, otázky na míru zadané administrátorem, sekce pro komplexní zprávu PR, rozšířené logování databáze a modul pro správu zaslané zpětné vazby.

Po zhodnocení vyplývá, že aplikace je použitelná v praxi, ať už samostatně, tak i v rámci navrženého systému.

SEZNAM POUŽITÉ LITERATURY

[1] What is .NET Framework?: .NET and .NET Framework. Dotnet: microsoft [online]. [cit. 2022-04-16]. Dostupné z: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework

[2] What is .NET? Introduction and overview: .NET fundamentals. Docs: microsoft [online]. 11.3.2022 [cit. 2022 04 16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/core/introduction

[3] .NET Standard: .NET fundamentals. Docs: microsoft [online]. 26.1.2022 [cit. 2022-04-16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-1-0

[4] LUIJBREGTS, Barry. The .NET Ecosystem explained. Azurebarry [online]. 15.8.2017 [cit. 2022-04-21]. Dostupné z: https://www.azurebarry.com/dot-net-ecosystem-explained/

[5] BINIASZ, KYLE. .NET Core vs .NET Framework: How to Pick a .NET Runtime for an Application. Stackify: net-core-vs-net-framework [online]. 2022, 28.1.2022 [cit. 2022-04-16]. Dostupné z: https://stackify.com/net-core-vs-net-framework/

[6] ROTH, Daniel, Rick ANDERSON a Shaun LUTTIN. Overview to ASP.NET Core: ASP.NET Core. Docs: microsoft [online]. 26.3.2022 [cit. 2022-04-16]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0

[7] What's new in .NET 6: .NET fundamentals. Docs: microsoft [online]. 27.1.2022 [cit. 2022-04-16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-6#:~:text=NET%206%20is%20the%20fastest,tools%2C%20and%20better%20team%20collaboration

[8] What's new in C# 10: What's new. Docs: microsoft [online]. 11.3.2022 [cit. 2022-04-16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-10#record-structs

[9] ASP.NET Core Blazor: Web apps. Docs: microsoft [online]. 26.3.2022 [cit. 2022-04-16]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0

[10] ASP.NET Core Blazor hosting models: Web apps. Docs: microsoft [online]. 31.3.2022 [cit. 2022-04-16]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-6.0#:~:text=The%20Blazor%20Server%20hosting%20model%20offers%20several%20benefits%3A,NET%20Core%20APIs.

[11] CHAPSAS, Nick. Blazor server-side vs client-side (WebAssembly) | What should you choose?. Youtube [online]. 27.9.2019 [cit. 2022-04-16]. Dostupné z: https://www.youtube.com/watch?v=HFvPKmS2gig

[12] Key Security Concepts: Principal. Docs: microsoft [online]. 09/15/2021 [cit. 2021-11-21]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/security/key-security-concepts

[13] ROUSOS, Mike. Overview of ASP.NET Core authentication: Authentication concepts. Docs: microsoft [online]. 25-5-2021 [cit. 2021-11-22]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/?view=aspnetcore-6.0

[14] Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app. Docs: microsoft [online]. 20-9-2021 [cit. 2021-11-22]. Dostupné z: https://docs.microsoft.com/en-us/azure/active-directory/develop/web-app-quickstart?pivots=devlang-aspnet-core

[15] Simple authorization in ASP.NET Core. Docs: microsoft [online]. 19-6-2021 [cit. 2021-11-22]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/security/authorization/simple?view=aspnetcore-6.0

[16] Introduction to authorization in ASP.NET Core: Authorization types. Docs: microsoft [online]. 10-5-2021 [cit. 2021-11-22]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction?view=aspnetcore-6.0

[17] ANDERSON, Rick. Introduction to Identity on ASP.NET Core: Create a Web app with authentication. Docs: microsoft [online]. 14-9-2021 [cit. 2021-11-22]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-6.0&tabs=visual-studio

[18] MARTIN, Rober. The Clean Architecture. Blog: cleancoder [online]. 13.8.2012 [cit. 2022-04-17]. Dostupné z: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[19] A quick introduction to clean architecture: Web design. Freecodecamp: quick-introduction-to-clean-architecture [online]. 13.8.2018 [cit. 2022-04-17]. Dostupné z:

https://www.freecodecamp.org/news/a-quick-introduction-to-clean-architecture-990c014448d2/

[20] ZIEMOŃSKI, Grzegorz. Clean Architecture Is Screaming. Dzone: clean-architecture-is-screaming [online]. 7.3.2017 [cit. 2022-04-17]. Dostupné z: https://dzone.com/articles/clean-architecture-is-screaming

[21] FROST, Brad. Atomic design. Bradfrost: atomic-web-design [online]. 10.6.2013 [cit. 2022-04-17]. Dostupné z: https://bradfrost.com/blog/post/atomic-web-design/

[22] Vishalxvii. Functional Programming Paradigm. Geeksforgeeks: functional-programming-paradigm [online]. 31.8.2021 [cit. 2022-04-18]. Dostupné z: https://www.geeksforgeeks.org/functional-programming-paradigm/

[23] TYSON, Matthew. What is functional programming? A practical guide. Infoworld: what-is-functional-programming-a-practical-guide [online]. 1.4.2021 [cit. 2022-04-18]. Dostupné z: https://www.infoworld.com/article/3613715/what-is-functional-programming-a-practical-guide.html

[24] POPOVIC, Jovan. Functional Programming in C#. Codeproject: Functional-Programming-in-Csharp [online]. 10.6.2012 [cit. 2022-04-18]. Dostupné z: https://www.codeproject.com/Articles/375166/Functional-Programming-in-Csharp

[25] ČÁPKA, David. LINQ v C# .NET - Revoluce v dotazování. Itnetwork: csharp [online]. [cit. 2022-04-18]. Dostupné z: https://www.itnetwork.cz/csharp/kolekce-a-linq/c-sharp-tutorial-linq-dotazy

[26] HAMILTON, Thomas. Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE. Guru99: unit-testing-guide [online]. 12.2.2022 [cit. 2022-04-18]. Dostupné z: https://www.guru99.com/unit-testing-guide.html

[27] IBM Cloud Education. SOA (Service-Oriented Architecture). Ibm: soa [online]. 7.4.2021 [cit. 2022-04-18]. Dostupné z: https://www.ibm.com/cloud/learn/soa

[28] SarthakGarg. Service-Oriented Architecture. Geeksforgeeks: service-oriented-architecture [online]. 29.8.2021 [cit. 2022-04-18]. Dostupné z: https://www.geeksforgeeks.org/service-oriented-architecture/

[29] RAJPUT, Dinesh. Service-oriented architecture (SOA). Oreilly: hands-on-microservices [online]. [cit. 2022-04-18]. Dostupné z:

https://www.oreilly.com/library/view/hands-on-microservices/9781789133608/936cb8b7-b10c-4093-9001-17fddcca0b30.xhtml

[30] Microservice vs monolitické: Rozdíl mezi mikroservisem a monolitickým. Cs.education: wiki [online]. [cit. 2022-04-18]. Dostupné z: https://cs.education-wiki.com/1235570-microservice-vs-monolithic

[31] WALKER, Alyssa. SOA vs Microservices: Key Differences explained with Examples: What is SOA?. Guru99: microservices-vs-soa [online]. 19.2.2022 [cit. 2022-04-18]. Dostupné z: https://www.guru99.com/microservices-vs-soa.html

[32] IBM Cloud Education. Message Brokers: What is a message broker?. Ibm: message-brokers [online]. 23.1.2020 [cit. 2022-04-20]. Dostupné z: https://www.ibm.com/cloud/learn/message-brokers#toc-message-br-oBdNX5GN

[33] What is a Message Broker?. Tibco: what-is-a-message-broker [online]. [cit. 2022-04-20]. Dostupné z: https://www.tibco.com/reference-center/what-is-a-message-broker

[34] RabbitMQ is the most widely deployed open source message broker. [online]. [cit. 2022-04-20]. Dostupné z: https://www.rabbitmq.com/

[35] OLAH, Gabor. An introduction to RabbitMQ – What is RabbitMQ?: What problem does RabbitMQ solve?. Erlang-solutions: an-introduction-to-rabbitmq [online]. 3.4.2020 [cit. 2022-04-20]. Dostupné z: https://www.erlang-solutions.com/blog/an-introduction-to-rabbitmq-what-is-rabbitmq/

[36] What can RabbitMQ do for you?. Features [online]. [cit. 2022-04-20]. Dostupné z: https://www.rabbitmq.com/features.html

[37] ALI, Asad. 4 Reliable RabbitMQ Hosting Platform for your Application. Geekflare: rabbitmq-hosting-platform [online]. 11.8.2020 [cit. 2022-04-20]. Dostupné z: https://geekflare.com/rabbitmq-hosting-platform/

[38] Topics. Rabbitmq: tutorial-five-dotnet [online]. [cit. 2022-04-20]. Dostupné z: https://www.rabbitmq.com/tutorials/tutorial-five-dotnet.html

[39] Co je DevOps?. Azure.microsoft: devops-overview [online]. [cit. 2022-04-20]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/what-is-devops/#devops-overview

[40] What is DevOps?. About.gitlab [online]. [cit. 2022-04-20]. Dostupné z: https://about.gitlab.com/topics/devops/

[41] What is CI/CD?. Redhat: what-is-ci-cd [online]. 31.1.2018 [cit. 2022-04-20]. Dostupné z: https://www.redhat.com/en/topics/devops/what-is-ci-cd

[42] What is Continuous Delivery?: Why continuous delivery?. Continuousdelivery [online]. [cit. 2022-04-20]. Dostupné z: https://continuousdelivery.com/

[43] What is Azure DevOps?. Docs: microsoft [online]. 9.2.2022 [cit. 2022-04-21]. Dostupné z: https://docs.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops

[44] Co je Azure Boards?. Docs: microsoft [online]. 18.4.2022 [cit. 2022-04-21]. Dostupné z: https://docs.microsoft.com/cs-cz/azure/devops/boards/get-started/what-is-azure-boards?view=azure-devops

[45] RabbitMQ Performance Measurements, part 2. Blog: rabbitmq [online]. 25.4.2012 [cit. 2022-04-21]. Dostupné z: https://blog.rabbitmq.com/posts/2012/04/rabbitmq-performance-measurements-part-2

[46] Authentication, Authorisation, Access Control. Rabbitmq: access-control [online]. [cit. 2022-04-21]. Dostupné z: https://www.rabbitmq.com/access-control.html

[47] Redis, Kafka or RabbitMQ: Which MicroServices Message Broker To Choose?. Otonomo [online]. [cit. 2022-04-21]. Dostupné z: https://otonomo.io/redis-kafka-or-rabbitmq-which-microservices-message-broker-to-choose/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API Application Programming Interface

BE Back-end

CD Continuous delivery

CI Continuous integration

DB Database

DI Dependency injection

ESB Enterprise service bus

FE Front-end

FP Functional programming

HOF High order functions

JS Javascript

JSON JavaScript Object Notation

LINQ Language-Integrated Query

REST Representational state transfer

MB Message broker

OOP Object-oriented programming

SOA Service-oriented architecture

SQL Structured Query Language

UI User interface

XML Extensible Markup Language

SEZNAM OBRÁZKŮ

Obrázek 1 .NET ekosystém 14
Obrázek 2 Counter – Blazor komponenta 16
Obrázek 3 Blazor a DOM komunikace – client side 17
Obrázek 4 Blazor a DOM komunikace – server side 18
Obrázek 5 Autentizace 19
Obrázek 6 ClaimsPrincipal hierarchie 20
Obrázek 7 Čistá architektura 22
Obrázek 8 Molekula 25
Obrázek 9 Immutabilita v C# - 1 28
Obrázek 10 Immutabilita v C# - 2 28
Obrázek 11 HOF funkce 29
Obrázek 12 LINQ kód 30
Obrázek 13 SOA architektura 32
Obrázek 14 Princip RabbitMQ 40
Obrázek 15 Princip Topic exchange 41
Obrázek 16 Devops fáze diagram 42
Obrázek 17 CI – build pipeline 44
Obrázek 18 CD release pipeline 45
Obrázek 19 Azure Devops 46
Obrázek 20 Schéma celkové architektury 52
Obrázek 21 Backlog pro ReviewMe 56
Obrázek 22 Úkoly pro ReviewMe 57
Obrázek 23 CI nastavení 57
Obrázek 24 CD nastavení stage 58
Obrázek 25 Nastavení serveru 59
Obrázek 26 RabbitMQ management UI 60
Obrázek 27 Users záložka 61
Obrázek 28 Exchange setup 61
Obrázek 29 Topic permissions 62
Obrázek 30 ReviewMe solution 63
Obrázek 31 Registrace autentizace pomocí JWT tokenu 64
Obrázek 32 Konfigurace služeb 64
Obrázek 33 Konfigurace služeb v CORE 65
Obrázek 34 EmployeesController 65

Obrázek 35 Core vrstva 67
Obrázek 36 Registrace bezpečnostních politik 67
Obrázek 37 ReviewMe.Infrastructure.DbStorage 69
Obrázek 38 ReviewMe.Infrastructure.EmailSender a RazorTemplates 69
Obrázek 39 ReviewMe.Infrastructure.RabbitMqConsumer 70
Obrázek 40 IEmployeeService 71
Obrázek 41 EmployeeService 71
Obrázek 42 EmployeeRepository 72
Obrázek 43 IAssessmentService 73
Obrázek 44 OpenAssessment část 1 73
Obrázek 45 OpenAssessment část 2 74
Obrázek 46 IReviewerTaskService 74
Obrázek 47 ReviewerTaskService – Submit 75
Obrázek 48 Notifikační email pro ohodnocení 76
Obrázek 49 GetEmailMessage metoda 77
Obrázek 50 IAssessmentNotificationService 78
Obrázek 51 Metoda NotifyOnOpenAssessment část 1 78
Obrázek 52 Metoda NotifyOnOpenAssessment část 2 79
Obrázek 53 DiConfigRabbitMqConsumer 80
Obrázek 54 Konfigurační soubor 81
Obrázek 55 Hostovaná služba 82
Obrázek 56 HandlerBase 83
Obrázek 57 AddMessageHandler 83
Obrázek 58 SynchronizeDataService – SendRequestToRefreshData 84
Obrázek 59 SynchronizationMessageHandler 84
Obrázek 60 ReviewMe.Models 85
Obrázek 61 ReviewMe.Frontend 86
Obrázek 62 ReviewMe.Components 87
Obrázek 63 FE – AssessmentService – OpenAssessment 87
Obrázek 64 AssessmentPage 88
Obrázek 65 AssessmentPageTemplate 88
Obrázek 66 Button komponenta 89
Obrázek 67 AssessmentPageTemplate kód sekce 89
Obrázek 68 Open assessment button 89
Obrázek 69 EmployeeMicroservice 91

Obrázek 70 ManageService 91
Obrázek 71 RabbitMQProducerDiConfig 92
Obrázek 72 MessageSender 92
Obrázek 73 SynchRequestHandler 93
Obrázek 74 Koncové body pro zaměstnance 94
Obrázek 75 Výstup koncového bodu Employees 95
Obrázek 76 Koncové body pro ohodnocení 95
Obrázek 77 Koncové body pro získání kolegů 96
Obrázek 78 Koncové body pro hodnocení a synchronizaci 96
Obrázek 79 List zaměstnanců 97
Obrázek 80 Vytváření ohodnocení 98
Obrázek 81 Vyfiltrovaný zaměstnanec s otevřeným ohodnocením 98
Obrázek 82 Notifikační email pro ohodnocení 99
Obrázek 83 Dialog pro odmítnutí ohodnocení 99
Obrázek 84 List hodnotících úkolů s historií 100
Obrázek 85 Provedení ohodnocení 100
Obrázek 86 Zobrazení zpětné vazby 101
Obrázek 87 Mikro služba – testování distribuce dat 102
Obrázek 88 Mikro služba – výpis logů při aktualizaci a zaslání zprávy 102
Obrázek 89 RabbitMq – zachycení odeslané zprávy 102
Obrázek 90 RabbitMq – ReviewMe BE – zachycení odeslané zprávy 103

SEZNAM PŘÍLOH

Příloha P I: Obsah vloženého zipu na CD

Příloha P 2: Popisné diagramy

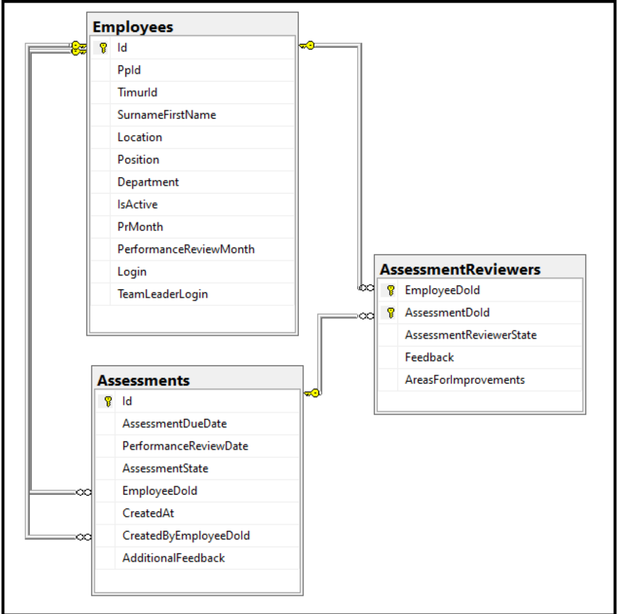
PŘÍLOHA P I: OBSAH VLOŽENÉHO ZIPU NA CD

Obsah:

- Elektronická verze diplomové práce
- Zdrojové soubory pro implementované řešení ve složce ReviewMe, ReviewMeFE a EmployeeDataMicroservice
- Návod pro nastavení a spuštění celého řešení

PŘÍLOHA P 2: POPISNÉ DIAGRAMY

- ER diagram pro ReviewMe aplikaci



ER diagram