Zadání diplomové práce

Název:	Systém pre spravovanie žiadateľov o prácu
Student:	Bc. Filip Figuli
Vedoucí:	Ing. Michal Gubiš
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cieľom práce je vytvorenie jednotlivých častí informačného systému pre správu kandidátov o zamestnanie v spoločnosti Mönkemöller IT GmbH. Systém zjednoduší riadenie výberového konania uchádzačov od prijatia žiadosti až po obsadenie pozície.

- Na základne informácií od zadávateľa analyzujte možnosti implementácie nasledujúcich častí systému
    - Administrácia systému
    - Životný cyklus žiadosti
    - Emailový klient pre prijatie žiadosti cez elektronickú poštu
- Navrhnite a implementujte serverové časti s využitím technológie ASP.NET Core a existujúcej proprietárnej knižnice
- Navrhnite a implementujte časti užívateľského rozhrania s využitím technológie Blazor WebAssembly
- Otestujte jednotlivé časti tak, aby sa overilo splnenie funkčných požiadaviek zadávateľa
- Nasaďte implementáciu v prostredí Microsoft Azure

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 15. prosince 2023 v Praze.

Diplomová práca

SYSTÉM PRE SPRAVOVANIE ŽIADATEĽOV O PRÁCU

Bc. Filip Figuli

Fakulta informačních technologií
Katedra softvérového inžinierstva
Vedúci: Ing. Michal Gubiš
9. mája 2024

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2024 Bc. Filip Figuli. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu: Figuli Filip. *Systém pre spravovanie žiadateľov o prácu.* Diplomová práca. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Poďakovanie vii

Vyhlásenie viii

Abstrakt ix

Zoznam skratiek x

Úvod 1

1 Predstavenie výberového konania a existujúcich problémov 3
1.1 Prijímací proces v Mönkemöller IT GmbH . . . 3
1.2 Problémy aktuálnej implementácie procesu . . . 4

2 Analýza 5
2.1 Doménový model . . . 5
2.2 Špecifikácia požiadaviek . . . 6
2.2.1 Všeobecné požiadavky . . . 6
2.2.2 Funkčné požiadavky . . . 7
2.2.3 Prípady použitia . . . 7
2.3 Model obchodných procesov . . . 8
2.4 Existujúce riešenia . . . 10
2.4.1 Konkurencia . . . 10
2.4.2 Motivácia . . . 10

3 Všeobecný návrh riešenia 11
3.1 Naplnenie všeobecných požiadaviek . . . 11
3.2 Technológie . . . 12
3.2.1 Výber jazyka . . . 12
3.2.2 Užívateľské rozhranie . . . 14
3.2.3 Server . . . 16
3.2.4 Spôsob uloženia dát . . . 17
3.2.5 Proprietárny framework Swift . . . 19
3.2.6 Výber technológií . . . 20
3.3 Návrhové vzory . . . 21
3.3.1 GRASP . . . 21
3.3.2 Viacvrstvová architektúra . . . 22
3.4 Architektúra . . . 22
3.4.1 Populárne architektonické vzory . . . 22

3.4.2 Výber architektúry . . . 24
3.5 Rozdelenie do modulov . . . 25

4 Emailový klient 27
4.1 Výzvy . . . 27
4.2 Návrh . . . 28
4.2.1 Databázový model . . . 30
4.3 Implementácia . . . 30
4.3.1 Synchronizácia . . . 30
4.3.2 Užívateľské rozhranie . . . 34
4.4 Testovanie . . . 36

5 Životný cyklus žiadosti 39
5.1 Výzvy . . . 39
5.2 Návrh . . . 40
5.2.1 Databázový model . . . 42
5.3 Implementácia . . . 42
5.4 Testovanie . . . 46

6 Administrácia systému 49
6.1 Výzvy . . . 49
6.2 Návrh . . . 49
6.2.1 Databázový model . . . 52
6.3 Implementácia . . . 52
6.4 Testovanie . . . 56

7 Nasadenie 59
7.1 Konfigurácia Azure . . . 59
7.2 Registrácia aplikácie . . . 60
7.3 Nasadenie aplikácie . . . 61

8 Možné vylepšenia 63

Záver 65

Obsah príloh 71

Zoznam obrázkov

2.1 Doménový model zachytávajúci vzťahy medzi hlavnými entitami 6
2.2 Model obchodných procesov 9

4.1 Databázový model emailového klienta 30
4.2 Užívateľské rozhranie emailového klienta 34

5.1 Užívateľské rozhranie žiadosti o prácu 40
5.2 Databázový model žiadosti o zamestnanie 42

6.1 Užívateľské rozhranie administrácie 50
6.2 Užívateľské rozhranie administrácie priečinkov 50
6.3 Dialóg pre úpravu konfigurácie priečinkov 51
6.4 Databázový model administrácie 52

Zoznam výpisov kódu

4.1 Ukážkový kód využitia Graph API pre získanie emailov užívateľa 28
4.2 Ukážkový kód použitia delta query 28
4.3 Kód pre synchronizáciu priečinku 31
4.4 Kód pre paralelnú synchronizáciu zmien 32
4.5 Zjednodušená verzia funkcie pre synchronizáciu 33
4.6 Ukážka kódu na rozpoznávanie pôvodu správy 34
4.7 Ukážka kódu pre vytváranie HTML obsahu emailu 35

5.1 Ukážka kódu triedy spravujúcej operácie nad aktivitami 43
5.2 Ukážka kódu základnej triedy pre deskriptor aktivít 44
5.3 Ukážka kódu využitia jazyku C# pri generovaní obsahu 44
5.4 Ukážka kódu deskriptor triedy pre prechod medzi fázami 44
5.5 Ukážka kódu základnej triedy pre prechod medzi fázami 45

6.1 Ukážka kódu komponenty AmsManagePagePart 53
6.2 Ukážka kódu rekurzívnej funkcie pre načítavanie stromu priečinkov 53
6.3 Ukážka kódu funkcie zodpovednej za sprostredkovanie dát priečinkov z Graph API na klienta 54
6.4 Ukážka kódu funkcie zodpovednej za uloženie zmien v priečinkoch do databázy 55
6.5 Ukážka kódu funkcie zodpovednej za aktualizáciu mien priečinkov v systéme 56

V prvom rade by som chcel poďakovať vedúcemu práce Ing. Michalovi Gubišovi za jeho trpezlivosť, cenné rady a čas, ktorý mi venoval pri vedení tejto diplomovej práce. Ďalej chcem poďakovať spoločnosti Mönkemöller IT GmbH za jedinečnú príležitosť sa podieľať na vývoji projektu Applicants Management System. Na záver by som chcel vyjadriť svoju hlbokú vďačnosť rodine a priateľom, ktorí boli mojim neochvejným zdrojom podpory a inšpirácie po celú dobu môjho štúdia.

Vyhlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. Ďalej prehlasujem, že som s Českým vysokým učením technickým uzavrel licenčnú zmluvu o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona. Táto skutočnosť nemá vplyv na ust. § 47b zákona č. 111/1998 Sb. o vysokých školách.

V Prahe dňa 9. mája 2024

Abstrakt

Diplomová práca sa zaoberá vývojom informačného systému pre spravovanie žiadateľov o prácu vo firme Mönkemöller IT GmbH. Aktuálny výberový proces sa spolieha na provizórne vytvárané spisy, komunikáciu cez externé aplikácie a časovo náročnú administratívu. Väčšina týchto činností sa dá spojiť v jednoduchom užívateľskom rozhraní, ktoré zjednotí formát spracovávaných informácií a zefektívni priebeh žiadosti.

Práca analyzuje aktuálnu implementáciu výberového procesu a požiadavky kladené na systém. Na základe analýzy vytvára návrh všeobecného riešenia potrebných technológií a výberu architektúry. Funkčné požiadavky sú adresované v osobitných moduloch, ktoré sa venujú analýze, návrhu, implementácií a testovaniu jednotlivých aspektov požadovanej funkcionality.

Najväčší prínos prichádza v modernizácií a zjednodušení výberového konania firmy Mönkemöller IT GmbH. Systém centralizuje dáta, minimalizuje potrebu manuálnej administratívy a zvyšuje efektivitu celého procesu.

Kľúčové slová multitenant, Graph API, Microsoft Cloud, Microsoft Azure, ASP.NET Core, Blazor, WebAssembly, PWA, PostgreSQL

Abstract

This diploma thesis explores the development of an information system for managing job applicants at Mönkemöller IT GmbH. The current selection process depends on provisionally created files, communication via external applications and time-consuming administration. Most of these activities can be integrated into a simple user interface, which will unify the format of the processed information and make the recruitment process more efficient.

The work analyzes the current implementation of the selection process and the requirements imposed on the system. Based on the analysis, a proposal for a general solution involving necessary technologies and the selection of architecture is developed. Functional requirements are addressed in separate modules, each devoted to the analysis, design, implementation and testing of various aspects of the required functionality.

The greatest contribution comes in the modernization and simplification of the recruitment process at Mönkemöller IT GmbH. The system centralizes data, minimizes the need for manual administration and increases the efficiency of the entire process.

Keywords multitenant, Graph API, Microsoft Cloud, Microsoft Azure, ASP.NET Core, Blazor, WebAssembly, PWA, PostgreSQL

Zoznam skratiek

PWA	Progressive web app
HTML	HyperText Markup Language
REST	Representational State Transfer
ORM	Object–relational mapping
HTML	HyperText Markup Language
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
API	Application programming interface
XML	Extensible Markup Language
JSON	JavaScript Object Notation
SQL	Structured Query Language
GRASP	General Responsibility Assignment Software Pattern
MVC	Model-View-Controlle
MVP	Model-View-Presente
MVVM	Model-View-Viewmodel
SOAP	Simple Object Access Protocol
DNS	Domain Name System
SSH	Secure Shell Protocol
CPU	Central processing unit
RAM	Random-access memory
URI	Uniform Resource Identifier

Úvod

Úspech spoločnosti je priamo úmerný profesionalite ľudí, ktorí v nej pracujú. Pútavá a prosperujúca firma si ľahko získa pozornosť zaujímavých kandidátov. Výber správneho tímu je kľúčom k víťazstvu. Otázkou zostáva, ako rozpoznať toho najvhodnejšieho kandidáta?

Práve z tohto dôvodu sú zavádzané rozsiahle náborové procesy, aby zabezpečili vysokú úroveň odbornosti a kvality tímu. Problém tohoto procesu spočíva v jeho priebehu. Proces zvyčajne zahŕňa rozsiahle debaty medzi už existujúcimi členmi tímu, improvizované záznamy o uchádzačoch a neefektívne, časovo náročné, administratívne postupy. Snaha o získanie ideálneho kandidáta sa často stretáva so stratou tých s najväčším potenciálom ku konkurencií, pretože je proces výberu príliš pomalý a neorganizovaný.

Týmto nedostatkom trpí mnoho firiem a Mönkemöller IT GmbH nie je výnimkou. Napriek dlhoročným skúsenostiam a vysokej kvalite výberového procesu postráda riadenie úkonov, fixnú štruktúru a efektívne spracovanie. V dôsledku toho sa čas potrebný na vybavenie jednotlivých žiadostí neprimerane predlžuje a zamestnanci nedokážu udržať krok s požiadavkami.

S príchodom moderných technológií vzrástol aj záujem o automatizáciu. Mnohé úkony, ktoré sa kedysi vykonávali manuálne, dnes preberajú informačné systémy, ktoré dáta efektívne spracúvajú a organizujú. Stali sa neoddeliteľnou súčasťou bežného života každého človeka.

Takýto prístup sa dá aplikovať aj na výberový proces firmy Mönkemöller IT GmbH, čo bolo motiváciou pre vznik tejto práce. Cieľom vzniknutého riešenia je adresovať problémy spojené s výberovým konaním a zefektívniť jeho priebeh. Softvér zadefinuje flexibilný a efektívny spôsob realizácie výberového procesu bez potreby udržiavania stavu žiadosti v improvizovaných záznamoch. Vďaka intuitívnemu užívateľskému rozhraniu sa eliminuje závislosť na neprispôsobených nástrojoch. Priebeh žiadosti bude dôkladne dokumentovaný v organizovaných formátoch vhodných pre archiváciu a budúce použitie. Užívatelia budú môcť vykonávať svoje úlohy bez nadbytočnej potreby využívať externé komunikačné kanály.

Kapitola 1

Predstavenie výberového konania a existujúcich problémov

V tejto kapitole je popísaná problematika výberového konania uchádzačov o zamestnanie vo firme Mönkemöller IT GmbH. Obsahuje stručný opis aktuálneho postupu, ktorým sa žiadosť vybavuje a vysvetlenie problémov, ktoré sú s týmto procesom spojené.

1.1 Prijímací proces v Mönkemöller IT GmbH

Firma Mönkemöller IT GmbH ako súčasť svojho programu na prijímanie nových zamestnancov zverejňuje pracovné ponuky na rôznych platformách. Potenciálni záujemcovia môžu, na emailovú schránku na to určenú, odosielať svoje životopisy s prípadným sprevádzajúcim textom a motivačným listom.

Jednotlivé žiadosti sú spracovávané zamestnancom z ľudských zdrojov, ktorý ich roztriedi zodpovedným osobám. V prípade, že sú informácie neúplné, zamestnanec kontaktuje žiadateľa formou pre-interview, kde sa všetko chýbajúce doplní. Po zozbieraní potrebných informácií, technický vedúci prevezme žiadosť a zhodnotí spôsobilosť kandidáta na danú pozíciu.

Nádejní kandidáti ďalej pokračujú do interview, ktoré môže prebiehať buď fyzicky, alebo formou videohovoru. Cieľom je overiť pravdivosť doložených informácií a zhodnotiť celkový dojem z kandidáta. Nasleduje preverenie praktických vedomostí cez implementačnú úlohu. K úlohe dostáva každý kandidát spätnú väzbu. V prípade spokojnosti technického vedúceho s riešením úlohy je tomuto kandidátovi ponúknutá pozícia vo firme.

Je nutné podotknúť, že nie všetky kroky procesu musia byť vykonané pri každom uchádzačovi. Napríklad v prípade, keď sa uchádzač zaujíma o viaceré pozície naraz, alebo už v minulosti bol súčasťou výberového procesu, nie je nutné počiatočné kroky na získavanie informácií opakovať. Rovnako, ak zamestnanec zodpovedný za preverenie znalostí uzná za vhodné, že nie je potrebné vykonať niektoré kroky, ako napríklad vypracovanie praktickej úlohy, je možné ich preskočiť.

1.2 Problémy aktuálnej implementácie procesu

Veľká časť problému s výberovým konaním pramení z toho, že firma nemá na spracovanie žiadne automatizované nástroje. Schránka zodpovedná za prijímanie emailov týkajúcich sa žiadosti, musí byť osobne kontrolovaná na dennej báze zamestnancom z ľudských zdrojov. Žiadostí prichádzajú desiatky, každá z nich musí byť zredukovaná na potrebné informácie. Tieto informácie nemajú štandardizovaný formát a sú uložené len v lokálnych súboroch, ktoré si zamestnanec vytvára počas spracovávania. Súbory nie sú centralizované, takže ich je potrebné zdieľať medzi zainteresovanými osobami pomocou komunikačných kanálov ako je email, alebo Microsoft Teams.

Na vybavenie žiadosti je potrebná pomerne intenzívna komunikácia v rámci tímu. Pracovník z ľudských zdrojov musí osloviť jednotlivých účastníkov výberového konania, synchronizovať medzi nimi stretnutia a dodať každému potrebné podklady pre vykonanie ich úlohy. Keďže sa proces skladá z niekoľkých po sebe nasledujúcich fáz, musí sa takáto synchronizácia opakovať. Z každej fázy je potrebné reportovať výstup, od ktorého potom závisí ďalší postup.

Orchestrácia toľkých úkonov s dôslednou dokumentáciou celého procesu je extrémne časovo náročná. Pri väčšom množstve žiadostí v úzkom časovom období nastáva veľký komunikačný šum, ktorý celý proces ešte viac spomaľuje a komplikuje. A ak aj sa žiadosti stíhajú vybavovať, nie je možné rozumne a prehľadne vybavené žiadosti archivovať tak, aby sa v nich dalo jednoducho vyhľadávať. Napríklad v prípade, keď sa rovnaký záujemca uchádza opätovne o rovnakú, alebo aj inú otvorenú pozíciu, sa väčšia časť procesu môže preskočiť. Niekedy je potrebné zistiť dôvody prečo bola predchádzajúci žiadosť neúspešná a na základe toho upraviť priebeh celého procesu.

Kapitola 2

Analýza

Úlohou tejto kapitoly je analyzovať problematiku výberového konania. Ako prvá je predstavená doména so základnými entitami, ktoré vo výberovom konaní figurujú. Ďalej sa tu nachádza obchodný model procesov, ktorý popisuje prijímací proces zachytávajúc kroky od prijatia žiadosti až po obsadenie pozície, resp. odmietnutie kandidáta. Nasleduje zhrnutie špecifikovaných požiadaviek na systém tak, aby riešil problémy spomínané v predošlej sekcii. Kapitolu uzavrie prehľad už existujúcich riešení a motivácia za vytváraním tejto práce.

2.1 Doménový model

Cez vstupný kanál do systému prichádzajú emaily ako reakcie na otvorené pozície vo firme. Tieto emaily sú často sprevádzané priloženými súbormi ako napríklad životopis, motivačný list a tak ďalej. Z nich následne vznikajú jednotlivé žiadosti. Ku každej žiadosti existuje žiadateľ s údajmi potrebnými pre budúci kontakt. V rámci každej fázy životného cyklu žiadosti, prebieha množstvo aktivít, za ktoré sú zodpovedné rôzne priradené osoby. Nasleduje krátky prehľad entít, ktoré sú hlavnými aktérmi vo výberovom procese s diagramom zobrazujúcim vzťahy medzi nimi.

Vstupný kanál - Kde sa žiadateľ dozvedel o otvorenej pozícii.

Otvorená pozícia - Pozícia, o ktorú sa kandidát uchádza.

Email - Informácie zaslané kandidátom.

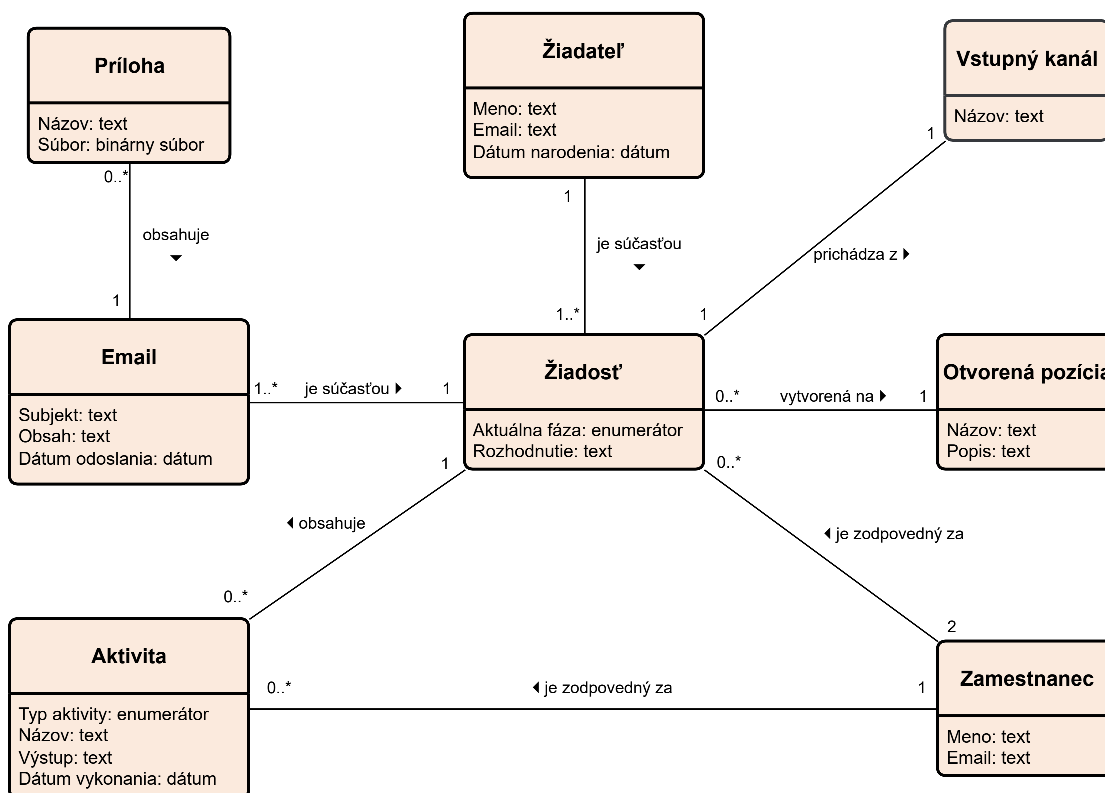
Prílohy - Názov a binárna reprezentácia súborov priložených ku emailu.

Žiadosť - Entita reprezentujúca žiadosť.

Žiadateľ - Kontaktné informácie reprezentujúce žiadateľa.

Aktivita - Činnosť naplánovaná ako súčasť procesu žiadosti.

Zamestnanec - Zamestnanec podieľajúci sa na vybavení žiadosti.



Obr. 2.1 Doménový model zachytávajúci vzťahy medzi hlavnými entitami

2.2 Špecifikácia požiadaviek

Cieľom tejto sekcie je vymedziť hranice systému a zachytiť, aké obmedzenia sú na systém kladené. Transformuje abstraktné potreby zadávateľa na fixne vymedzené všeobecné a funkčné požiadavky. Všeobecné požiadavky ovplyvňujú voľbu návrhu a architektúry systému, zatiaľ čo funkčné požiadavky špecifikujú, aké funkcie má systém poskytovať.

2.2.1 Všeobecné požiadavky

N1: Dostupnosť cez web ako PWA - Užívateľské rozhranie systému bude dostupné cez moderný webový prehliadač a to ako progresívna webová aplikácia.

N2: Nasadenie v cloud prostredí - Aplikácia musí byť kompatibilná s cloudovým prostredím Microsoft Azure.

N3: Multitenant režim - Systém musí podporovať režim viacerých tenantov. To znamená, že bude schopný obsluhovať viacero spoločností, alebo ich častí, paralelne. Dáta tenantov sa nesmú pretínať.

N4: Technológie - Vývoj prebehne na platforme .NET v jazyku C#. Pre implementáciu musí byť využitá proprietárna knižnica Swift s pomocou frameworku ASP.NET Core a jeho súčasťou Blazor.

2.2.2 Funkčné požiadavky

F1: Synchronizácia dát z Microsoft Cloud - Systém bude schopný pravidelne aktualizovať zoznam prichádzajúcich emailov na základe zmien zaznamenaných v nakonfigurovanej poštovej schránke Microsoft Cloud.

F2: Vytváranie žiadosti z emailu - Po prijatí emailu z Microsoft Cloud zamestnanec vytvorí novú žiadosť o zamestnanie.

F3: Prepájanie rôznych emailov ku žiadosti - Žiadosť bude obsahovať emailovú komunikáciu, ktorá s ňou súvisí.

F4: Zmena stavu žiadosti - Žiadosť bude prechádzať viacerými fázami počas svojho životného cyklu. Bude možné postupovať v procese dopredu aj dozadu.

F5: Zamietnutie žiadosti - Žiadosť môže byť v ľubovolnej fázi zamietnutá v prípade, že uchádzač nespĺňa potrebné kvality.

F6: Vytvorenie aktivity súvisiacej so žiadosťou - Súčasťou životného cyklu každej žiadosti je plánovanie rozličných aktivít, ktoré prebiehajú v konkrétnej fáze. Tieto aktivity sú priradené k príslušným zodpovedným osobám v systéme. Každá aktivita je zakončená textovým výstupom a číselným hodnotením kandidáta.

F7: Administrácia dát v systéme - V systéme budú konfigurovateľné dáta. Každý tenant si môže nastaviť vlastné vstupné kanály, otvorené ponuky, či priečinky pre synchronizáciu.

2.2.3 Prípady použitia

UC1: Nastavenie priečinku pre synchronizáciu - Administrátor rozhodne, ktoré priečinky sa majú do systému synchronizovať z emailovej schránky Microsoft Cloud.

UC2: Vytvorenie žiadosti z emailu - Zamestnanec si zobrazí v emailovom klientovi novú žiadosť o zamestnanie a na základne jej detailov vyplní potrebné údaje pre inicializáciu žiadosti.

UC3: Prepojenie rôznych emailov ku žiadosti - V emailovej schránke sa nachádza email, ktorý predstavuje pokračovanie konverzácie s uchádzačom o zamestnanie. Zamestnanec si vyhľadá príslušnú žiadosť a email k nej priradí.

UC4: Zmena stavu žiadosti - Pre aktuálnu fázu žiadosti sú splnené všetky potrebné aktivity. Zamestnanec presunie žiadosť do ďalšej fázy výberového konania.

UC5: Zamietnutie žiadosti - Uchádzač nespĺňa niektoré z kritérií potrebných na úspešné absolvovanie výberového procesu. Zamestnanec žiadosť zamieta.

UC6: Vytvorenie aktivity súvisiacej so žiadosťou - Zamestnanec podľa aktuálnej fázy, v akej sa žiadosť nachádza, organizuje príslušné udalosti a prideľuje ich zodpovedným osobám.

UC7: Administrácia dát v systéme - Užívateľ zodpovedný za administráciou dát v systéme môže konfigurovať vstupné kanály, otvorené pozície a priečinky na synchronizáciu pre svojho tenanta.

2.3 Model obchodných procesov

Pokračuje krátky popis krokov výberového procesu nasledovaný modelom obchodných procesov. Kroky nie sú povinné, je možné ich za istých okolností preskočiť.

Spracovanie žiadosti - Žiadosti prichádzajú v neštandardizovanom formáte. Je potrebné z nej vybrať dôležité informácie a inicializovať žiadosť v systéme.

Doplnenie chýbajúcich informácií - V prípade, že informácie neboli kompletné zamestnanec z ľudských zdrojov naplánuje pre-interview, kde sa dopýta potrebné detaily a doplní ich do žiadosti.

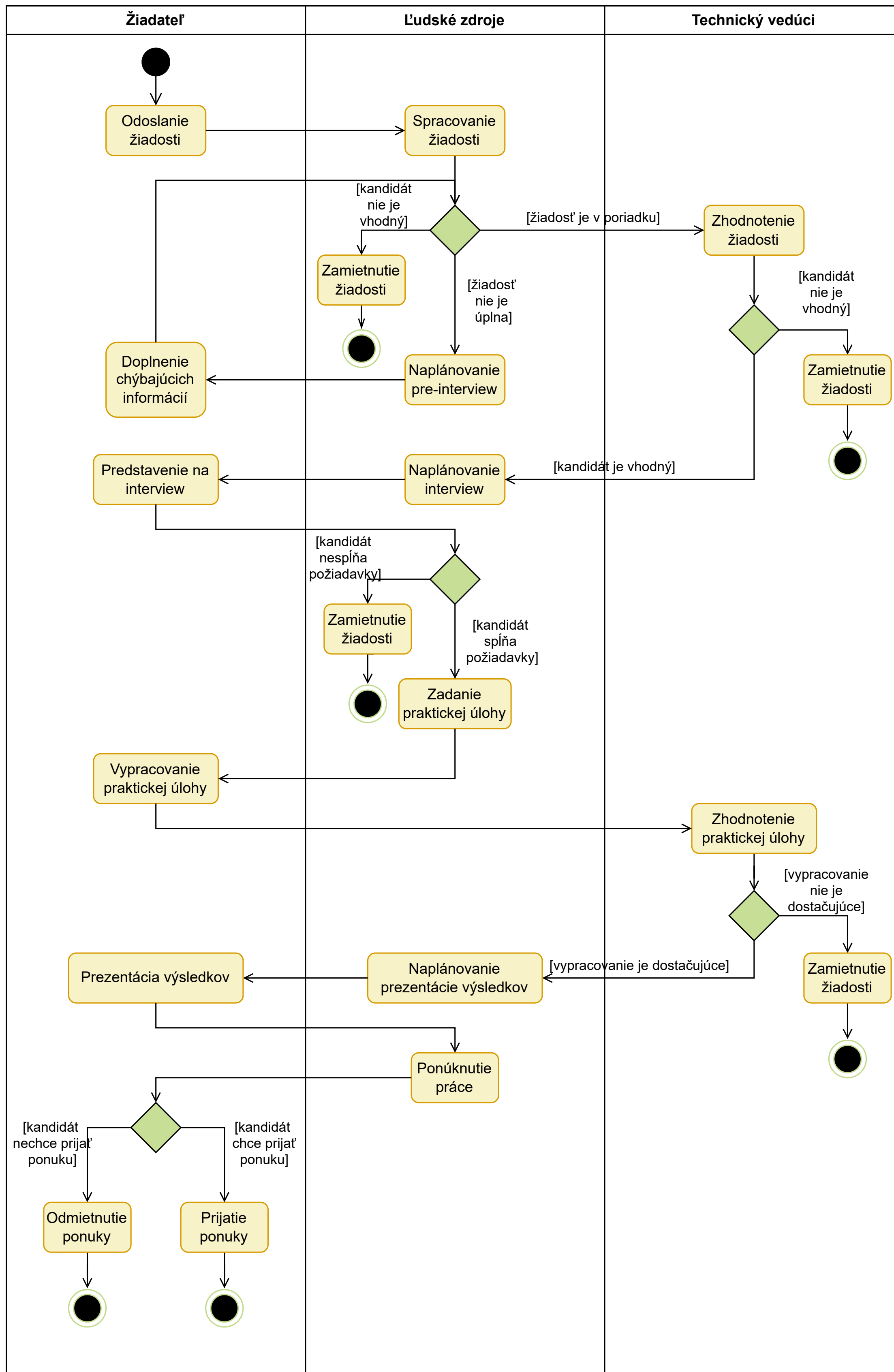
Zhodnotenie žiadosti - Skompletizovanú žiadosť dostáva na starosti technický vedúci. Zhodnotí schopnosti uchádzača a jeho predpoklady byť vhodným kandidátom na danú pozíciu. Žiadosť sa v kladnom prípade posúva ďalej na oddelenie ľudských zdrojov, kde sa naplánuje interview.

Predstavenie na interview - Úlohou interview je overiť pravdivosť doložených informácií a zistiť aké ma uchádzač predstavy o spolupráci. Zamestnanec rozhovor dokumentuje a na konci zhodnotí svoj subjektívny dojem z uchádzača.

Zadanie praktickej úlohy - Kandidát úspešný v interview dostáva na vypracovanie praktickú úlohu. Cieľom úlohy je preveriť znalosti v kontexte pozície o ktorú sa kandidát uchádza.

Zhodnotenie praktickej úlohy - Vypracovanie úlohy dostáva na zhodnotenie technický vedúci. Posúdi technické zdatnosti kandidáta a spíše krátku recenziu kódu, aby kandidát dostal plnohodnotnú spätnú väzbu.

Prezentácia výsledkov - Spätná väzba od technického vedúceho je odprezentovaná zamestnancom z ľudských zdrojov počas stretnutia. Ak uchádzač dosiahol stanovené štandardy kvality riešenia, dostane ponuku práce.



Obr. 2.2 Model obchodných procesov

2.4 Existujúce riešenia

Pred začatím vývoja je dôležité zistiť aktuálny stav na trhu, prípadne či už neexistuje riešenie, ktoré by klientovi poskytlo schodné podmienky. Táto sekcia predstaví populárne alternatívy a vysvetlí motiváciu za tvorbou tejto práce.

2.4.1 Konkurencia

BambooHR je platforma poskytujúca množstvo rôznych funkcionalít na podporu oddelenia pre ľudské zdroje v podniku. Jej hlavným cieľom je poskytnúť nástroje na spracovanie dát a produkovanie rôznych výročných správ. Napomáha adaptácii nových zamestnancov a sleduje ich pokrok. Pre už zabehnutých zamestnancov poskytuje podporu pre manažment formou kalkulácie miezd, zjednodušuje proces vyplácania platov a sleduje výkon jednotlivých zamestnancov.

Workable sprostredkúva priestor, kde sa firma môže svojim potenciálnym záujemcov príťažlivým spôsobom odprezentovať. Ponúka podporu pre tvorbu rôznych transparentov s pracovnými ponukami priamo na ich platforme. Takto vytvorené ponuky sú následne automaticky zdieľané do populárnych sociálnych sietí. Firma nemusí ponuky len vytvárať môže rovnako vyhľadávať priamo kandidátov a osloviť ich cez Workable. Umožňuje nastaviť vlastný schvaľovací proces, ktorý je možné synchronizovať v rámci tímu. Po ukončení výberového procesu sú informácie združované do súhrnných správ, ktoré firme pomáhajú identifikovať odkiaľ prichádzajú najlepší kandidáti, alebo aké sú potenciálne medzery vo výberovom procese.

Greenhouse cez detailne nakonfigurované formuláre pomáha zachytiť nevyhnutné informácie od potenciálnych kandidátov. Takto vytvorené ponuky zdieľa medzi potenciálnych záujemcov. Má podporu pre vysoko granulovaný výberový proces s rôznymi právami v rámci tímu čím napomáha lepšej spolupráci. Celý proces sprevádza ľahko použiteľné užívateľské rozhranie, ktoré zjednodušuje všetky potrebné úkony, ktoré sú súčasťou výberového konania. Umožňuje zbierať dodatočné informácie formou automatických dotazníkov, z ktorých generuje prehľadné súhrnné správy ako spätnú väzbu celého procesu.

2.4.2 Motivácia

Všetky tri diskutované riešenia sú zamerané na podporu oddelenia pre ľudské zdroje, no nie priamo na asistenciu počas výberového procesu. Ponúkajú rôznorodé funkcie pre tvorbu súhrnných správ, manažment zamestnancov a vyhľadávanie kandidátov.

Spoločnosť Mönkemöller IT GmbH disponuje existujúcimi nástrojmi pre vykonávanie týchto činností. Ich výberový proces funguje správne a nie je potrebné ho prispôsobovať. Hlavné nedostatky spočívajú v náročnom spracovaní aplikácií a zložitej komunikácii medzi členmi tímu. Pre zachovanie výberového procesu je potrebné implementovať riešenie na mieru, ktoré by celý proces zjednodušilo a automatizovalo. Implementácia diskutovaných riešení by si vyžiadala úpravu celého procesu pre integráciu do platformy a kompletnú reštrukturalizáciu oddelenia pre ľudské zdroje.

Kapitola 3

Všeobecný návrh riešenia

Táto kapitola nadväzuje na predchádzajúcu analýzu. Kým analýza popisuje, čo je úlohou práce, návrh vysvetľuje, ako to je možné dosiahnuť. Cieľom je vybrať vhodnú architektúru a rozhodnúť o nevyhnutných návrhových krokoch. Návrh jednotlivých modulov bude podrobnejšie spracovaný v ďalších kapitolách.
Najskôr prebehne zhrnutie všeobecných požiadaviek nasledované výberom technológií. Ďalej budú v krátkosti spomenuté dôležité návrhové vzory, ktoré pomôžu pri výbere architektúry. Kapitolu zakončí rozdelenie práce na menšie moduly a rozdelenie projektu do balíčkov.

3.1 Naplnenie všeobecných požiadaviek

Skôr, ako začne diskusia nad výberom technológií či architektúry, je nevyhnutné objasniť, aké kritéria musí riešenie spĺňať. Z všeobecných požiadaviek v analýze vyplynulo, že užívateľské rozhranie musí byť dostupné cez moderný webový prehliadač, systém musí podporovať režim mutlitenant a nasadenie sa bude realizovať v cloudovom prostredí Microsoft Azure.

Progresívna webová aplikácia je druh aplikácie vytvorenej pomocou webových technológií, ktorá na prvý pohľad pripomína natívnu aplikáciu [1]. Tradičné webstránky sa nenachádzajú v zariadení užívateľa, ale sú navštevované pomocou webového prehliadača. Ten zaistí kompatibilitu ako aj potrebnú funkcionalitu pre chod stránky. Tým získava stránka schopnosť fungovať na ľubovolnom systéme, ktorý má podporu pre nejaký webový prehliadač. To však za cenu úplného zapuzdrenia v prehliadači, čím stráca väčšinu výhod natívnych aplikácií, ako je napríklad prístup ku súborom, offline režim, alebo aj aktualizovanie stavu aplikácie na pozadí. Stránka sa načíta v užívateľskom rozhraní prehliadača, čiže nefiguruje ako samostatná aplikácia. Progresívna webová aplikácia kombinuje vlastnosti oboch prístupov a tým obchádza problémy s nimi spojené. Pri navštívení takejto stránky dostáva užívateľ výzvu, či chce, aby sa stiahla do jeho systému. Takto stiahnutá aplikácia aj naďalej potrebuje prehliadač na svoj chod, ale ten už figuruje len ako abstraktná vrstva bez užívateľského rozhrania, ktorú užívateľ nevníma. Stiahnutie aplikácie predstavuje prebratie všetkých zdrojov od serveru, tak ako by to bolo pri načítaní stránky,

ale miesto toho, aby bola v dynamicky alokovanej pamäti prehliadača, sa aplikácia ukladá do lokálneho úložiska a teda môže pracovať v offline režime.

Toto kritérium má výrazný dopad na spôsob implementácie aj na výber samotných technológií. Vzhľadom k tomu, že sa jedná o webstránku, bude potrebné implementovať osobitne serverovú časť systému a potom užívateľské rozhranie. Tieto dve časti typicky spolu komunikujú pomocou série požiadaviek a odpovedí, cez ktoré si vymieňajú potrebné dáta. Pre dosiahnutie zdanlivo plynulého chodu aplikácie je potrebné využiť niektorý z moderných frameworkov, ktorý zabezpečí prevedenie užívateľského rozhrania ako webovú aplikáciu, nie statickú stránku.

Multitenant implementácia vyžaduje návrh systému, ktorý zabezpečí disjunktnú manipuláciu a správu dát. Každý tenant musí mať k dispozícii svoju vlastnú sadu bezpečnostných prvkov a nesmie byť možné čítať či upravovať dáta druhého tenanta.

Dosiahnutie takejto funkcionality spočíva v označovaní jednotlivých entít značkou unikátnou pre tenanta. Manipulácia s databázou bude možná len za podmienky, že požiadavka obsahuje spomínanú značku. Systém teda vždy vráti len tie dáta, ktoré tenant sám vytvoril. Ako dodatočnú vrstvu bezpečnosti budú citlivé informácie zašifrované kľúčmi, ktoré vzniknú počas pridávania tenanta do systému.

3.2 Technológie

Napriek tomu, že zákazník má pomerne presnú predstavu o tom, aké technológie sa majú pri implementácii použiť je dôležité diskutovať aj rôzne iné varianty pre kompletnosť návrhu. V prípade, že by sa našiel vhodnejší výber technológií, zákazník je otvorený ku konzultácii týchto zistení a prípadnej úprave zadania.

Táto sekcia sa pozrie jednotlivo na to, aké technológie sú dostupné v súlade s vymedzenými možnosťami všeobecných požiadaviek. Najskôr je potrebné zistiť, aké programovacie jazyky sú vhodné pre túto prácu. Zo všeobecných požiadaviek vyplynulo, že sa implementácia bude deliť na dve časti, a síce server realizovaný ako webové API s užívateľským rozhraním vo forme webovej aplikácie. Ďalej bude pokračovať diskusia nad spôsobom uloženia dát pre efektívne spracovanie a správu dát v disjunktnej forme pre režim multitenant. Sekciu zakončí argumentácia výberu technológií vzhľadom ku zisteným informáciám.

3.2.1 Výber jazyka

Programovacie jazyky je možné kategorizovať do rôznych skupín. Táto klasifikácia hlavne odráža metódu, ktorou jazyk dosahuje požadovaných výsledkov [2].

Procedurálne programovacie jazyky nasledujú špecifický postup. Kód sa vykonáva riadok po riadku v malých celkoch nazývaných procedúra. Využíva premenné, slučky a funkcie pre dosiahnutie chceného výsledku. Sú vhodné pre nízko úrovňové programovanie, kde chce mať programátor plnú kontrolu nad tým, ako sa kód vykonáva. Sú nevhodné pre implementáciu komplexných informačných systémov, kde nie je cieľom vysoká efektivita [3].

Funkcionálne programovacie jazyky sú deklaratívne. Ich využitie spočíva v skladaní funkcií, čím programátor vytvára stromy výrazov, ktoré priraďujú ku hodnotám výsledky [4].

Objektovo orientované programovacie jazyky sa sústredia na aplikáciu rôznych návrhových vzorov, udržateľnosť kódu a možnosť znovupoužitia. Rozdeľujú logické časti kódu do tried a objektov za cieľom zapuzdrenia. Využíva rôzne úrovne abstrakcie, ktorá zakrýva komplikované časti kódu a umožňuje programátorovi manipulovať len so zjednodušeným rozhraním [5].

Skriptovacie programovacie jazyky sú využívané primárne na automatizáciu opakujúcich sa úloh, ako podpora pre už existujúce systémy, alebo vytváranie dynamického obsahu na webstránke [2].

Logické programovacie jazyky narozdiel od spomínaných programovacích jazykov, ktoré poskytujú presné inštrukcie pre vykonávanie úloh počítačom, definujú sériu logických pravidiel, ktoré určujú rozhodovací proces počítača.

Logické a funkcionálne programovacie jazyky nie sú na potreby implementácie určené vzhľadom k tomu, že sa jedná o informačný systém, ktorý bude mať pomerne veľký rozsah a musí byť udržateľný v časovom horizonte niekoľkých rokov. Do tejto role rovnako nespadajú čisto procedurálne jazyky, ktorých najčastejším využitím sú rýchle a efektívne algoritmy, kde programátor využíva ich schopnosť pracovať s kódom na nízkej úrovni. Skriptovacie jazyky určené pre vývoj dynamických webstránok by mohli byť použité pre realizáciu užívateľského rozhrania. Implementácia serveru bude teda realizovaná objektovo orientovaným jazykom [2]..

Java prichádza s myšlienkou: "Write once, run anywhere". Jedná sa o vysoko úrovňový, objektovo orientovaný jazyk, ktorý je dizajnovaný tak, aby bol spustiteľný na skoro ľubovolnom zariadení. Java využíva prístup, ktorý sa snaží maximálne limitovať závislosti zatiaľ čo si zachováva rôznorodosť použitia. Kód sa kompiluje do inštrukčnej sady nazývanej bytecode. Bytecode vie následne vykonávať Java virtual machine bez ohľadu na využitú architektúru počítača [6].

Python vznikal s nadčasovým pohľadom na syntax kódu. Jeho dizajn kladie dôraz na čitateľnosť kódu využívaním špecifických pravidiel pri odsadzovaní jednotlivých riadkov.

Jedná sa o dynamicky typovaný jazyk, ktorý sa dá zaradiť medzi viaceré programovacie paradigmy. Kód vykonáva procedurálne s využitím objektov, obsahuje aj súčasti funkcionálneho programovania. Ako väčšina dnes používaných vysoko úrovňových jazykov, využíva garbage collector na princípe počítania jednotlivých referencií a následné uvoľňovanie nevyužitej pamäte v cykloch.

Python bol dizajnovaný, aby sa dal ľahko rozširovať. Je to jeden z dôvodov za jeho veľkým úspechom. Napriek tomu, že za bežných okolností sa vývoj v pythone nezameriava na vysokú optimalizáciu, programátor má možnosť rozšíriť svoje riešenie modulmi programovanými napríklad v jazyku C [7].

C# je objektovo-orientovaný programovací jazyk. Jeho hlavnými prednosťami sú typová bezpečnosť, spracovanie výnimiek, výrazy lambda a integrovaná podpora pre dotazy.

Pri kompilácii je jeho kód preložený do intermediate language, ktorý zostane zapuzdrený v tzv. assembly balíčku spolu so sprievodnými informáciami. Takto vytvorený assembly balíček môžeme načítať do common language runtime, ktorý vykoná finálny preklad do natívnych strojových inštrukcií.

Narozdiel od niektorých iných jazykov, C# oslobodzuje programátora od potreby vysporiadať sa s nízkoúrovňovými problémami ako napríklad správa pamäti. Tú zaň vykonáva garbage collector, ktorý automaticky pamäť uvoľňuje, keď ju už program nepotrebuje [8].

JavaScript si netreba zamieňať s Javou. Zatiaľ čo Java je plnohodnotný objektovo orientovaný jazyk, JavaScript pôvodne vznikol len ako skriptovací jazyk s účelom vytvárať dynamický obsah na vtedy čisto statických webových stránkach. Napriek tomu, že väčšina jeho využitia stále spočíva vo vykonávaní skriptov v rámci prehliadača, JavaScript sa stal pomerne populárnym vo svete programovania a dostal pozornosť aj od veľkých firiem ako Microsoft či Google.

Dnes sa už jedná o vysoko úrovňový jazyk kompilovaný za behu aplikácie. Využíva dynamické typovanie, objekty a rôzne iné funkcie veľkých jazykov ako je C# či Java. Jeho zaujímavou vlastnosťou je slučka udalostí, v ktorej sa zoraďujú jednotlivé udalosti čakajúce na vykonanie [9].

3.2.2 Užívateľské rozhranie

Úlohou užívateľského rozhrania je prezentovať užívateľovi aktuálny stav dát a reagovať na jeho vstupy. Z všeobecných požiadaviek vyplýva, že užívateľské rozhranie bude zabezpečené webovou aplikáciou. Takáto aplikácia bude schopná fungovať na rôznych platformách, zároveň využije schopnosti natívnych aplikácií ako offline režim alebo aktualizovanie aplikácie na pozadí.

Na dosiahnutie tohto riešenia existuje množstvo rôznych technológií. Táto sekcia predstaví čitateľovi tie najznámejšie z nich, ktoré spolupracujú s vybranými jazykmi predošlej sekcie.

Vaadin je open-source platforma pre vývoj webových aplikácií v jazyku Java. Prináša Java web framework s rôznymi nástrojmi pre uľahčenie vývoja moderných webových užívateľských rozhraní.

Implementácia Vaadinu spočíva vo využití takzvaného Vaadin Flow. Vaadin Flow pomáha programátorom implementovať užívateľské rozhranie bez potreby priamo používať jazyk HTML, alebo JavaScript. Architektúra využíva tradičný model na strane serveru, ktorý umožňuje bezpečné spracovanie väčšiny operácií obchodnej logiky.

Klientská časť využíva na komunikáciu so serverom technológiu WebSocket, ktorá umožňuje dynamické aktualizovanie oboch strán. Na vykreslenie užívateľského rozhrania Vaadin poskytuje dva rôzne prístupy.

Vaadin dlhodobo využíva komponentovú architektúru. Každý element je navrhnutý ako samostatná funkčná komponenta. Tieto komponenty sa potom spájajú dokopy, čím vytvárajú celistvé a interaktívne užívateľské rozhranie.

Novšie verzie Vaadinu prichádzajú s novým prístupom vo forme Hilla frameworku. Klientskú časť aplikácie už negeneruje Java, ale nahrádza ju TypeScript. Tieto dve časti spolu komunikujú na preddefinovanom REST rozhraní [10].

Reflex ako čerstvý prírastok do sveta moderných webových aplikácií, prichádza s podobným prístupom, ale tentokrát v jazyku python. Využíva tiež komponentovú architektúru, ktorá zaobaľuje prvky JavaScript knižnice nazývanej React. Tak vytvára plynulé užívateľské rozhranie bez potreby hlbokej znalosti webových technológií.

S nástrojmi na skladanie užívateľského rozhrania prichádza aj množstvo šikovných a praktických pomôcok pre vývoj serveru. Programátor tu môže nájsť veľmi obľúbené Object-Relational Mapping (ORM), ktoré poskytuje prepojenie s ľubovolnou databázou. Vytvára objekty, ktoré v kóde odrážajú entity v databáze a uľahčuje tak prácu s dátami [11].

ASP.NET Core Blazor je frontendový webový framework poskytovaný platformou .NET. Dovoľuje programátorovi vyvíjať užívateľské rozhranie generované na strane klienta, alebo aj na strane serveru. Delí sa na dva základné typy Blazor Server a Blazor WebAssembly.

Blazor Server nevyužíva klasický prístup požiadavka-odpoveď ako bežné webstránky. Narozdiel od nich táto komunikácia prebieha cez technológiu SignalR s využitím protokolu WebSocket. Napriek tomu, že užívateľ má pocit, že internuje s progresívnou webovou aplikáciou, všetok kód je vykonávaný na serveri. Server reaguje na jednotlivé úkony užívateľa tým, že si porovnáva aktuálny stav stránky u klienta s jeho lokálne vygenerovaným stavom a posiela klientovi len rozdiel medzi nimi. Napriek vysokej optimalizácii takto vytvorená webstránka podlieha problému vyššej latencie a nie je schopná pracovať v offline režime.

Blazor WebAssembly využíva technológiu WebAssembly na vykonávanie C# kódu u klienta. To mu umožňuje špeciálny WebAssembly .NET runtime. Samotná stránka, jej kód a aj závislosti sa odošlú na klienta, kde už aplikácia pracuje nezávisle na serveri priamo v prehliadači. Takto vytvorená aplikácia je tiež známa ako Progresívna Webová Aplikácia, ktorá vie využívať rôzne výhody a možnosti podobne ako natívne aplikácie. Po tom čo si ju užívateľ stiahne do vlastného systému získa možnosť práce v offline režime, aktualizácie stavu na pozadí a odosielanie upozornení.

Blazor aplikácie sú založené na Razor komponentách. Jedná sa o prvky užívateľského rozhrania, ktoré v kóde figurujú ako klasické C# triedy. Razor komponenty pozostávajú zo zmesi HTML kódu a C# kódu, tiež známe ako Razor markup. Tieto prvky následne programátor dokopy kombinuje, a tým vzniká rozhranie aplikácie [12].

Angular predstavuje platformu pre vývoj webových aplikácií, ktorá je založená na jazyku JavaScript, respektíve na jeho rozšírení nazývanom TypeScript. Rovnako ako iné už spomenuté frameworky, aj Angular používa komponentovú architektúru k tvorbe užívateľského rozhrania za pomoci jednotlivých, plne funkčných elementov.

Stránka vzniká formou HTML šablóny, ktorá definuje štruktúru dokumentu, do ktorej Angular dynamicky vykresľuje komponenty. Komplexnú funkcionalitu webstránky obhospodarujú metódy deklarované v rámci tried komponent. Tie komunikujú so zvyškom aplikácie pomocou služieb, ktoré zapúzdrujú a vykonávajú potrebnú obchodnú logiku. V prípade, že aplikácia potrebuje externé dáta môže sa služba obrátiť na server pomocou HTTP požiadavky. Stránka reaguje na vstup od užívateľa pomocou udalostí, ktorým programátor predefinuje konkrétne správanie [13].

React je obľúbená knižnica jazyku JavaScript, ktorá sa používa na tvorbu dynamických a interaktívnych webových rozhraní. Je založená na komponentovej architektúre, ktorá umožňuje vytváranie opakovane použiteľných komponentov užívateľského rozhrania. Pre zvýšenie efektivity využíva Virtual Document Object Model, ktorý optimalizuje vykresľovanie minimalizovaním počtu aktualizácií v rámci dokumentu stránky. React vytvára v pamäti virtuálnu reprezentáciu stránky, na ktorej realizuje potrebné zmeny. Tieto zmeny následne optimalizuje len na tie nevyhnutné pre docielenie chceného výsledku, a tie potom aplikuje na skutočnú stránku. [14]

3.2.3 Server

Keďže klientská strana aplikácie bude operovať v systéme klienta, je nevyhnutné implementovať spôsob komunikácie so serverom. Štandardne sa pre bezstratovú komunikáciu medzi klientom a serverom využíva protokol TCP. Ten zabezpečí, že každý odoslaný paket je skontrolovaný a potvrdený na oboch stranách. Na štandardizáciu obsahu požiadaviek a odpovedí sa využíva protokol HTTP. Špecifikuje niekoľko rôznych verzií správy, ktoré si môže server s klientom vymeniť. Vďaka tomu obe strany vedia rozpoznať, čo druhá strana požaduje.

Táto časť poskytne prehľad technológií, ktoré by mohli byť relevantné pre implementáciu serverovej časti systému. Selekcia týchto technológií je založená na vybraných jazykoch a kompatibilite s technológiami, ktoré boli spomenuté v predchádzajúcej časti.

Springboot je open source framework pre vytváranie aplikácií, ktoré bežia pod Java virtual machine. Presnejšie sa jedná o sadu nástrojov pre podporu vývoja webových aplikácií alebo mikro servisov pomocou frameworku Spring. Automatizuje spravovanie závislostí a uľahčuje celý proces nasadenie softvéru ako osobitnej aplikácie [15].

Spring framework vznikol ako riešenie na radu problémov s ktorými sa bežne programátori potýkajú pri vývoji podnikových aplikácií. Skladá sa z niekoľkých vrstiev, ktoré spolupracujú na vytváraní informačného systému. Od mapovania databázových entít na objekty, cez správu dependency injection až po vystavenie ovládačov pre zachytávanie HTTP požiadaviek. V kontexte tejto práce môže byť využitie vo forme webového API, ktoré bude sprostredkovávať prepojenie s databázou a služby vykonávajúce obchodnú logiku aplikácie priamo nad dátami [16].

Django je open source webový framework postavený na jazyku python pre podporu pri vývoji udržateľných a bezpečných webstránok. Zaobstaráva väčšinu problémov s bežným vývojom systému, aby sa programátor mohol sústrediť na implementáciu svojho riešenia.

Jedná sa o kompletné riešenie pre webové stránky. Pomáha pri vytváraní servera, ktorý sa spája s databázou. Poskytuje podporu pre spracovanie HTTP požiadaviek a generovanie webového užívateľského rozhrania. Nie je však podmienkou využiť užívateľské rozhranie Djanga, je bežnou praxou ho kombinovať s modernými frameworkami pre vytváranie dynamických webových stránok, ako sú napríklad Angular či React [17].

ASP.NET Core predstavuje sadu nástrojov pre vývoj moderných webových aplikácií. Podobne, ako aj ostatné časti .NET platformy, umožňuje nasadenie na rôznych platformách, a je dostupný ako open-source.

Užívateľské rozhranie zastrešuje už spomínaný Blazor framework, ktorý je integráciou ASP.NET Core spolu s webovým API. ASP.NET Core. Podobne ako už spomínané technológie, uľahčuje prácu vývojárom tým, že redukuje potrebu zaoberať sa úplne základnými aspektmi a umožňuje im sústrediť sa na logiku aplikácie. Súčasťou je aj vstavaná podpora pre dependency injection, modulárne ovládače pre spracovanie HTTP požiadaviek a framework pre ORM mapovanie databázových entít do objektov.

Vývojári nie sú obmedzení voľbou operačného systému, keďže ASP.NET Core je kompatibilný s Windows, Linux a macOS. Architektúra je navrhnutá tak, aby bola vhodná pre jednoduché testovanie a efektívne nasadanie v cloudových službách, alebo v kontajneroch ako je Docker [18].

Node.js je populárny webový framework založený na jazyku JavaScript. Je to open-source platforma, ktorá je kompatibilná s viacerými operačnými systémami. Hlavným cieľom Node.js je umožniť vývojárom používať JavaScript mimo tradičného webového prehliadača. Využíva na to špeciálne navrhnuté JavaScript prostredie.

Pred vznikom Node.js bol JavaScript použiteľný výhradne v prostredí prehliadača. Ak chceli vývojári vytvoriť aplikáciu, ako napríklad webový server, museli sa obrátiť na iné programovacie jazyky. Node.js túto situáciu zmenilo a dnes ho množstvo veľkých firiem využíva na prevádzku svojich online platforiem [19].

Node.js získalo popularitu vďaka svojej jednoduchosti a efektívnosti. Programátori môžu v priebehu niekoľkých minúť pomocou správcu balíčkov, ako je napríklad npm, nainštalovať všetky potrebné závislosti do svojho projektu. Rovnakou cestou sa dajú využiť aj stovky rôznych knižníc, ktoré pre JavaScript boli vytvorené . Po napísaní niekoľkých riadkov kódu vzniká plne funkčný server. Navyše aplikácie vytvorené pomocou Node.js sú ľahko naladiteľné na rôznych hostingových platformách [20].

3.2.4 Spôsob uloženia dát

Databázy sú nevyhnutnou súčasťou informačných systémov. Predstavujú softvérovú komponentu pre uchovávania a konzistenciu dát. Existuje niekoľko druhov databáz, ktoré by mohli byť vhodné pre riešenie zadania. Táto časť má za cieľ poskytnúť prehľad o rôznych typoch databázových technológií a určiť, ktorá z nich najlepšie pokryje nároky softvéru. Nasleduje porovnanie dostupných databázových systémov, ktoré do danej technológie spadajú [21].

Dokumentové databázy sa považujú za takzvané NoSQL databázy. To znamená, že miesto tradičného spôsobu ukladania dát vo formáte riadkov a tabuliek, dáta sú uložené vo flexibilnejšom formáte dokumentov.

Dokument predstavuje záznam v databáze. Záznam zvyčajne obsahuje nejaký objekt zapísaný vo formáte JSON, alebo XML. Ďalej sa dokumenty, ktoré majú podobný obsah, zoskupujú do zbierok. Pre udržiavanie konzistentného obsahu sa využívajú preddefinované validačné schémy, ktoré zabezpečia, že sa do systému nedostanú neúplné dáta. Databázový engine poskytuje API rozhranie pre prístup k dátam, ktorý môže informačný systém využívať.

Hlavnými výhodami dokumentových databáz je ich flexibilný pohľad na dáta. Kým relačné databázy vyžadujú fixne dohodnuté vzťahy a štruktúru dát, dokumenty nevyžadujú úplnú zhodu entít pre zoskupenie v zbierke, ak sa to nevynúti validačnou schémou. Takto uložené dáta sa potom dajú v prípade potreby ľahko horizontálne škálovať medzi viaceré databázové enginy [22].

Relačné databázy je typ databázy, ktorá uchováva dáta organizované v riadkoch a stĺpcoch, ktoré tvoria tabuľky. Samotný názov databáz vychádza z unikátnej vlastnosti vytvárať medzi jednotlivými entitami väzby, tiež známe ako relácie. Každý zápis v databáze je zvyčajne sprevádzaný unikátnym identifikátorom. Tento identifikátor si môže zapamätať iný zápis v tabuľke čím vzniká medzi nimi väzba.

Pre štandardizáciu komunikácie medzi rôznymi relačnými databázami sa využíva jazyk SQL. Umožňuje vytvárať takzvané query, ktoré reprezentujú jednu špecifickú otázku na databázový engine. SQL má niekoľko kľúčových slov, ktoré môže programátor kombinovať do komplexných queries.

Striktný prístup k dátam, napriek očividne nižšej efektívnosti, prináša veľa výhod. Všetky zmeny prebiehajú počas jednej transakcie, takže buď sa úspešne vykonajú všetky naraz, alebo žiadna. To zaručuje aj neustále konzistentný stav databázy. V prípade viacerých požiadaviek v rovnaký čas, sú požiadavky zoradené podla istých pravidiel a vykonávajú sa nadväzne na seba. Navyše databázový engine vie rozpoznať či jednotlivé transakcie do seba zasahujú a prípadne ich môže spustiť súčasne [23].

Grafové databázy uchovávajú dáta v grafových štruktúrach a radia sa zvyčajne medzi NoSQL databázy. Ich hlavným cieľom je zachytávať veľmi komplexné relačné dáta.

Základ grafových databáz spočíva v teórii grafov. Entity reprezentujú jednotlivé uzly a vzťahy medzi nimi odzrkadľujú hrany. Uzly môžu obsahovať dodatočné vlastnosti. Jednotlivé grafy sa väčšinou dodatočne organizujú do podgrafov, ktoré vypovedajú informácie o danej skupinke entít.

Najväčšou výhodou grafových databáz je ich schopnosť vyjadriť komplexné vzťahy medzi entitami. Sú schopné efektívne spracúvať otázky cielené na prepájanie entít na základe ich spoločných vlastností [24].

Zadanie vyžaduje pomerne zložitú dátovú architektúru obsahujúcu množstvo entít, ktoré budú podliehať špecifickým pravidlám. Táto štruktúra výrazne obmedzuje možnosť efektívneho využitia flexibility, ktorú nám ponúkajú dokumentové databázy. Okrem toho sa nepredpokladá, že by práca s databázou bola natoľko intenzívna, aby sa musela využiť rýchlosť a efektivita dokumentových databáz.

Napriek tomu, že sú dáta pôvodne komplexné, v kontexte daného doménového modelu je možné vidieť, že nebude priestor na využitie výhod grafovej databázy. Typicky sa aplikujú na analýzu vzorcov správania medzi skupinami entít s podobnými charakteristikami, alebo vzťahmi. Avšak v tomto prípade takéto situácie nevznikajú, keďže vzťahy medzi entitami sú primárne informačného charakteru.

Relačná databáza je schopná efektívne spracovať komplexnú štruktúru dát. Jej schopnosť, udržiavať dáta konzistentné, je kľúčová, najmä keď s informačným systémom pracuje viacero používateľov súčasne. Využívanie relačných databáz pri implementácii informačných systémov je vo svete preferovaným trendom. Všetky spomínané serverové technológie integrujú ORM technológiu, ktorá umožňuje spojenie systému s databázou bez potreby zásahu programátora.

Oracle databázový systém sprostredkúva firma Oracle. Plná verzia databázy je komerčná, na vyskúšanie existuje jej obmedzená verzia zadarmo. Podporuje veľkú škálu rôznych systémov ako Linux, Mac OS, alebo Windows. Pre jazyk Java poskytuje rozšírený spôsob ako vytvárať uložené procedúry. Vyniká svojimi výkonnostnými charakteristikami a schopnosťou efektívneho rozširovania operácií na viacero uzlov.

Microsoft SQL Server je databázový systém od firmy Microsoft. Podobne ako Oracle aj v tomto prípade je plná verzia databázy platená. Je možné si ju vyskúšať v obmedzenom režime zadarmo. Operačný systém pre chod SQL Serveru je možný len na platformách Linux a Windows. Pre .NET framework poskytuje špeciálny spôsob komunikácie cez ADO.NET technológiu. Uložené procedúry môžu byť napísane aj v jazyku C#. Schopnosť SQL Serveru využívať viacero uzlov je značne limitovaná, ale novšie verzie ju už podporujú.

PosgreSQL je open-source databázový systém, ktorý je vyvíjaný a spravovaný skupinou PostgreSQL Global Development Group. Napriek tomu, že je bezplatný, poskytuje väčšinu funkcií ako konkurenčné komerčné produkty. Zo spomínaných technológií podporuje najväčšiu škálu rôznych platforiem vrátane Linux, Mac OS a Windows. Jeho rozhranie umožňuje integráciu s ADO.NET, alebo aj Java technológiami ako JDBC. Implementuje vlastnú verziu skriptovacieho jazyku nazývanú pgSQL. Oproti ostatným riešeniam tu chýba podpora pre prácu s dátami v pamäti no nezaostáva vo svojej schopnosti sa škálovať do viacerých uzlov.

SQLite predstavuje odlišný koncept práce s relačnými databázami. Namiesto použitia samostatnej aplikácie s komunikačným rozhraním, SQLite využíva takzvanú bez serverovú architektúru, realizovanú ako integrovaný databázový systém uložený priamo na disku v podobe súboru. S databázou programátor komunikuje pomocou rozhrania, ktoré dnes podporuje každý známy programovací jazyk. Týmto získava veľkú flexibilitu a úplne bezproblémové nasadenie v každom systéme.

3.2.5 Proprietárny framework Swift

Swift je framework vyvinutý spoločnosťou Mönkemöller IT GmbH, ktorý je založený na platforme .NET. Kladie dôraz na znovupoužiteľnosť a modularitu kódu. Poskytuje unifikovaný základ pre vývoj aplikácií tým, že integruje rovnaký kód v rozličných projektoch. Tým zjednodušuje prechod vývojárov medzi projektami. Využíva rôzne frameworky

a knižnice tretích strán, ktoré zaobaľuje do vlastnej implementácie, ktorá je potom kompatibilná s každým projektom, ktorý ho využíva. Medzi hlavné časti, ktoré by mohli byť použité pri vývoji tejto práce sú object–relational mapping, validácia objektov pre prenos dát, serializácia týchto objektov do objektov obchodnej logiky, Blazor komponenty a kontexty pre dependency injection.

3.2.6 Výber technológií

Ešte donedávna bol vývoj webových technológií doménou špecialistov s rozsiahlymi znalosťami v mnohých disciplínach, od návrhu užívateľských rozhraní s použitím HTML, CSS a JavaScriptu, až po vývoj serverovej časti v objektovo orientovaných programovacích jazykoch. To často vyžadovalo spoluprácu niekoľkých programátorov s tým, že sa každý zameriaval na jednu časť webovej aplikácie.

Dnes je k dispozícii hneď niekoľko frameworkov implementovaných vo viacerých jazykoch, ktoré sa snažia minimalizovať tento problém tým, že buď samotný jazyk generuje užívateľské rozhranie aplikácie, alebo sa priamo podieľa na jej samotnom behu. Tým sa znížia nároky na programátorov, ktorí sú potrební pre vývoj ako taký, a neskôr na údržbu. Zároveň sa znižuje technický dlh celej aplikácie, keďže jej súčasti sú v jednom projekte a je jednoduchšie ich udržiavať jednotné.

Webové technológie neustále napredujú, čo prináša inovatívne prístupy k ich implementácii. JavaScript sa už nevyužíva len na vykonávanie jednoduchých úprav či animácií, ale vďaka nemu vznikajú plnohodnotné dynamické webové aplikácie. Vývoj webových aplikácií pokročil do bodu, kedy v oblasti výkonu konkurujú natívnym aplikáciam a súčasne riešia základné problémy platformovej závislosti.

Za týchto podmienok najlepšie vychádza pre užívateľské rozhranie využiť jeden z JavaScript frameworkov, ako je Angular alebo React. Umožnili by jednoduché nasadenie aplikácie v PWA režime, aj bezproblémovú prácu v offline režime. Zatiaľ čo ostatné framworky sú závislé od neustálej komunikácie so serverom pre generovanie jednotlivých častí stránky, JavaScript aplikácia dokáže pracovať priamo u klienta bez potreby sa neustále spoliehať na server.

Situáciu komplikuje prvá podmienka spomínaná v tejto sekcii. Ak má užívateľské rozhranie aj server využívať rovnaký jazyk, znamenalo by to využívať Node.js technológiu na implementáciu serveru. Napriek tomu, že Node.js nie je úplne nová technológia, jej využitie pre potreby serveru informačného systému bolo donedávna pomerne nezvyčajné v porovnaní s populárnejšími alternatívami ako Java či C#.

Táto situácia spôsobila, že firma Mönkemöller IT GmbH nevyužívala túto technológiu, čo má za dôsledok nedostatok skúseností s týmto typom dizajnu. Firma sa dlhodobo zameriava na vývoj softvéru na platforme .NET a využíva vlastné technológie na ňom postavené. Takým je aj proprietárny framework Swift.

Riešenie tohto problému poskytuje nový štandard nazývaný WebAssembly. Jedná sa o už dnes zaužívanú alternatívu pre spúšťanie programov v prehliadači iným spôsobom ako pomocou JavaScriptu. Pre .NET WebAssembly poskytuje platformu, kde vie plnohodnotne pracovať .NET runtime a umožnil vznik Blazor WebAssembly frameworku.

Blazor WebAssembly spĺňa všetky potrebné kvality. Využíva jednotný jazyk pre frontend aj backend a zároveň v ňom implementovaná aplikácia vďaka WebAssembly pracuje efektívne skoro na úrovni natívnej aplikácie. Nechýba tu ani podpora pre offline režim.

Serverovú časť zastreší ASP.NET Core WebAPI, ktoré pomocou HTTP protokolu môže s užívateľským rozhraním bez problémov komunikovať. Napokon takýto prístup umožní kompatibilitu s existujúcim proprietárnym frameworkom Swift, keďže celá aplikácia bude implementovaná v jazyku C#.

Pre uloženie dát je vzhľadom na zadanie najvhodnejšie využiť relačnú databázu. Napriek tomu, že v databáze sa bude nachádzať väčšie množstvo entít, nejedná sa o veľmi komplexný databázový model s množstvom relácií a komplikovanými štruktúrami. Neočakáva sa ani nadmerné množstvo zápisov, ktoré by vyžadovalo vysokú efektívnosť databázy. Za týchto okolností sú zaujímavejšie open-source riešenia, keďže by systém nedokázal využiť dodatočné funkcionality poskytované komerčnými riešeniami.

SQLite je veľmi unikátne riešenie, ktoré vyniká svojou schopnosťou pracovať ako lokálny súbor. Tým však stráca výkon v porovnaní s osobitným databázovým enginom a napriek menšiemu očakávanému rozsahu databázy by to mohlo spôsobiť ťažko migrovateľné problémy. Za týchto okolností je najlepšou voľbou PostgreSQL databáza, ktorá je open-source takže zadávateľ nemusí platiť za nevyužitý nadštandard komerčných databáz zatiaľ čo poskytuje kompletné riešenie pre relačné databázy s vysokou kompatibilitou, škálovateľnosťou a výborným výkonom.

3.3 Návrhové vzory

Návrhové vzory sú dôležitou súčasťou procesu návrhu softvéru. Zabezpečujú zrozumiteľnosť, rozšíriteľnosť a udržateľnosť systému. Táto sekcia priblíži tie najdôležitejšie, na ktoré sa budú odkazovať rozhodnutia pri výbere architektúry.

3.3.1 GRASP

General Responsibility Assignment Software Patterns (GRASP) je zbierka základných vzorov a programovacích princípov pre priradenie zodpovednosti ku triedam, ktoré hrajú kľúčovú rolu v návrhu efektívnych a udržateľných informačných systémov. Pod zodpovednosťou sa myslí nejaká úloha, ktorú ma daná trieda riešiť. Pri návrhu sú podstatné tri základne princípy a síce informačný expert, nízka previazanosť a vysoká súdržnosť [25].

Informačný expert rozdeľuje zodpovednosť tak, aby poverená časť mala všetky potrebné informácie na vykonávanie aktivít, ktoré ma na starosti.

Nízka previazanosť spočíva v minimalizovaniu počtu nevyhnutných závislostí na jednotlivých častiach projektu. Jej cieľom je, aby každá súčasť vedela plnohodnotne fungovať samostatne. Veľkou výhodou tohto prístupu je znížený dopad pri vykonávaní zmien. Zvyšuje aj možnosť znovupoužitia.

Vysoká súdržnosť sa zameriava na zmysluplnosť kódu ako takého. V zmysle nízkej previazanosti je cieľom, aby zodpovednosti jednotlivých častí boli čo najviac poprepájané a dávali ako celok zmysel. Zvyšuje sa tým súdržnosť a znovupoužitie. Zodpovednosti takto ucelených častí sú jednoduchšie na pochopenie.

3.3.2 Viacvrstvová architektúra

Viacvrstvová architektúra pomáha rozdeliť monolitický softvér do viacerých separátnych vrstiev zodpovedných za osobitné úkony počas behu aplikácie. Týmto priamo naväzuje na princípy GRASP. Výhodou takéhoto prístupu je nezávislosť jednotlivých vrstiev, a teda ich jednoduchá prípadná výmena, testovanie či možnosť využitia viacerých rôznych riešení naraz. Napríklad môže softvér využívať viaceré užívateľské rozhrania na základe toho, akým zaradením je aktuálne využívaný.

Jednovrstvová aplikácia zvyčajne prichádza s jednoduchším počiatočným vývojom, ale zložitou udržateľnosťou. Narúša nízku previazanosť systému a aj vysokú súdržnosť, keďže všetky operácie, dáta aj prípadné užívateľské rozhranie zabezpečuje jedna časť. Je vhodná buď na jednoduché nástroje, alebo ukážkové prototypy.

Dvojvrstvová aplikácia oddeľuje prezentačnú vrstvu od dátovej. Tým získava istú mieru nízkej previazanosti, keďže je možné pri správnom návrhu napríklad vymeniť užívateľské rozhranie bez toho, aby bolo nutné výrazne pozmeniť dátovú vrstvu. Využíva sa primárne pre jednoduché aplikácie, ktorých hlavným účelom je podporovať základné databázové operácie.

Trojvrstvová aplikácia rozdelí oproti tomu dátovú vrstvu ešte na dve ďalšie časti. Získava tak prezentačnú vrstvu na zobrazenie dát, vrstvu obchodnej logiky na spracovanie požiadavkov od užívateľa a dátovú vrstvu, ktorá poskytuje prístup ku perzistentne uloženým dátam. Tento prístup sa stal populárny primárne pre zložitejšie enterprise aplikácie.

Trojvrstvová architektúra sa ďalej delí na dva typy. Prvá nazývaná striktná dovoľuje, aby komunikácia medzi vrstvami prebiehala len zhora na dol a len o jednu úroveň. Čiže prezentačná vrstva komunikuje s vrstvou obchodnej logiky, a tá následne môže komunikovať s dátovou vrstvou. Na druhej strane relaxačná trojvrstvová architektúra dovoľuje komunikácii prechádzať cez ľubovolný počet úrovní. V takomto prípade má prezentačná vrstva prístup priamo ku dátovej vrstve. Jedná sa o najbežnejšie používaný spôsob návrhu informačných systémov.

3.4 Architektúra

3.4.1 Populárne architektonické vzory

V duchu spomínaného GRASPu a viacvrstvových architektúr nasledujú populárne postupy pri navrhovaní softvéru.

Model-View-Controller (MVC) je spôsob organizovania kódu na osobitné súčasti rozdelené na základe ich úloh. Primárnou výhodou tohto prístupu je zjednodušené upravovanie a rozširovanie týchto častí.

View reprezentuje vizuálnu časť aplikácie. Prezentuje dáta užívateľovi a špecifikuje ako môže užívateľ so zvyškom systému interagovať. Napríklad sa môže jednať o užívateľské rozhranie aplikácie.

Model predstavuje dáta. Špecifikuje, aké dáta je možné využívať a akým spôsobom sa k nim dá pristupovať. Je dobrým zvykom sa snažiť dáta modelovať tak, aby čo najviac pripomínali realitu. Zbytočne komplikovaný abstraktný model môže eventuálne viesť ku ťažkému pochopeniu a technickému dlhu.

Controller rozhoduje o chodu aplikácie. Zbiera interakcie od užívateľa, a na základe nich upravuje model a vykonáva validáciu vstupov. Tu sa v aplikácii nachádza všetka obchodná logika a procesy. Upravený model následne vykreslí view a užívateľ môže spozorovať výsledky jeho úkonov.

Užívateľ teda vizuálne vníma view. Vykonáva rôzne úkony, ktoré zachytáva controller a reaguje na ne potrebnými operáciami alebo výpočtami. Controller manipuluje s dátami modelu a takto upravené dáta sú spätne zobrazené cez view naspať k užívateľovi v aktualizovanom užívateľskom rozhraní [26].

Model-View-Presenter (MVP) je nadstavba nad MVC. Jedná sa o výsledok snahy riešiť problém vysokej previazanosti užívateľského rozhrania a logiky aplikácie.

Pri použití MVC controller zachytáva vstup od užívateľa a aktualizuje model, ktorý si následne vyžiada view, aby ho zobrazil. MVP nahrádza controller za presenter a kompletne upraví celý komunikačný tok medzi komponentami.

Užívateľ komunikuje s view, ktoré následne informuje presenter o zmenách. Presenter slúži ako sprostredkovateľ, ktorý modifikuje model podľa potreby. Keď sa v modeli uskutočnia zmeny, informuje o tom presenter prostredníctvom definovaného rozhrania. Presenter potom aktualizuje view, čo umožňuje užívateľovi vidieť aktuálne dáta.

Všetka komunikácia teda prechádza cez prostredníka, ktorým je presenter. Zatiaľ čo využívanie špecifického rozhrania view môže budiť dojem nižšej efektivity, je ďaleko prirodzenejšie ako pôvodný prístup MVC a uľahčuje udržateľnosť kódu ako aj testovanie vďaka nižšej previazanosti s view [27].

Model-View-Viewmodel (MVVM) prichádza s úplne iným spôsobom ako rozdeliť architektúru na jednotlivé vrstvy. View a model stále viac menej zastávajú rovnaké funkcie ako pri MVC a MVP, rozdiel prichádza v absencii controlleru či presenteru, ktoré sú nahradené viewmodelom.

Viewmodel poskytuje rôzne atribúty a príkazy, na ktoré si view vytvára dátové väzby. Zmeny v týchto atribútoch sú sprevádzané udalosťami, ktoré o nich oboznamujú view. Všetka obchodná logika aplikácie je zabezpečená viewmodelom. View sa len rozhoduje, ako zareaguje na zmenu v dátach, ktoré mu viewmodel poskytuje, a ako ich zobrazí.

Viewmodel ďalej pozná rozhranie modelu a vie s nim manipulovať. Nemusí však robiť vždy sprostredkovateľa, je možné vytvárať dátové väzby modelu priamo na atribúty, ktoré sú naviazané na view.

Je potrebné si uvedomiť, že tento druh komunikácie prebieha iba jedným smerom. Model musí prostredníctvom udalostí informovať viewmodel o zmene vo svojich dátach. Zatiaľ čo view pozná rozhranie viewmodelu a viewmodel pozná rozhranie modelu, opačný smer komunikácie neexistuje.

Takýto prístup prináša niekoľko výhod. Keďže viewmodel netuší o existencii view, je jednoduché vytvárať testy, ktoré úlohu view nahradia a tak testovať celú implementáciu od viewmodelu až k modelu. Vzhľadom k tomu, že viewmodel sa správa ako adaptér k modelu, udržiava tak obmedzenia, ktoré musí splniť. To bráni programátorovi narušiť hlavnú funkcionalitu modelu [28].

Controller-Service-Repository prináša iný pohľad a návrh softvéru. Vynecháva kompletne užívateľské rozhranie a miesto neho dedikuje controller na komunikáciu s vonkajším svetom. Service zastáva obchodnú logiku aplikácie a repository uľahčuje prístup do databázy.

Takýto návrhový vzor môže byť vhodný napríklad na realizáciu Web API, kde užívateľské rozhranie nehrá žiadnu rolu. Potrebná je len schopnosť reagovať pomocou dopredu dohodnutého rozhrania s ostatnými časťami systému.

Representional State Transfer (REST) je architektonický štýl, ktorý špecifikuje štandard komunikácie medzi webovými servermi a ich rozhraním. Takto dohodnutý štandard dovoľuje implementovať server a rozhranie úplne osobitne bez nadbytočnej koordinácie. Vyberie sa formát správ, ktorými budú strany komunikovať, a kým je dodržaný, nedôjde ku žiadnemu komunikačnému šumu. Komunikáciu vždy iniciuje strana klienta.

Klient si vyžiada dáta alebo ich serveru poskytne. Celá komunikácia prebieha formou výmeny informácií, nie vykonávaním príkazov, a je vždy špecifikovaná metódou požiadavky HTTP protokolu [29]. Samotnú komunikáciu si nezvykne prechovávať ani jedna zainteresovaná strana, takže prebieha bez udržiavania stavu. To výrazne zjednodušuje škálovanie serveru a zvyšuje celkový výkon [30].

Simple Object Access Protocol (SOAP) je protokol, ktorý udáva štandard akým spolu komunikujú webové technológie. Komunikácia prebieha výmenou XML súborov, ktoré sú oproti RESTu výrazne obsiahlejšie. Podobne ako REST používa HTTP protokol pre špecifikáciu správy [31].

3.4.2 Výber architektúry

Z diskusie nad výberom technológií vyplynulo, že softvér sa bude vyvíjať formou dvoch oddelených aplikácií. Bude využitá technológia ASP.NET Core WebApi na poskytovanie prístupu Blazor WebAssembly aplikácie ku obchodnej logike a dátam.

Serverová časť potrebuje prístup ku databáze a zároveň mať priestor na vykonávanie potrebnej obchodnej logiky. Nebude však mať nijaké užívateľské rozhranie. Komunikácia bude prebiehať čisto na úrovni HTTP protokolu s klientskou aplikáciou podľa potreby. Preto je vhodné využiť Controller-Service-Repository návrhový vzor. Controller bude v tomto prípade realizovať API. Service zastane rolu jednotlivých obchodných operácií, ktoré musí server vykonávať a napokon repository zabezpečí proprietárny framework Swift, ktorý implementuje prístup do databázy aj s objektovým mapovaním entít.

Rozdelenie implementácie na dve časti má svoje výhody aj z pohľadu vývoja. Je možné dopredu dohodnúť fixné rozhranie, ktoré obe strany musia dodržať pre plynulú

komunikáciu. Vďaka tomu bude možné vyvíjať každú stranu nezávisle bez narušenia funkcionality. Takto štandardizovanú komunikáciu je možné docieliť použitím SOAP protokolu, alebo architektúry REST. Keďže hlavnou výhodou SOAP protokolu je zasielanie obsiahlych správ, ktoré systém nebude potrebovať, bude uprednostnená jednoduchšia implementácia REST.

V tejto situácii klientská aplikácia nie je len pasívnym zobrazením view, ktoré je naplnené dátami z modelu. Pracuje ako plnohodnotná samostatná aplikácia. Pri návrhu je teda možné využiť napríklad spomínané trojvrstvové architektúry, kde view slúži ako užívateľské rozhranie. MVC a MVP sú tradične vhodné pre webstránky, ale môžu byť menej efektívne pri častých aktualizáciách užívateľského rozhrania aplikácie, ktorá nie je renderovaná na serveri. Naopak MVVM efektívne spracováva dáta a zabezpečuje, že pri ľubovolnej zmene sa užívateľské rozhranie môže aktualizovať bez potreby kontrolovania modelu. Blazor WebAssembly nasleduje tento prístup, poskytuje podporu pre dátové viazanie a jeho komponenty sú navrhnuté tak, aby zvládli časté úpravy viewmodelu, čím umožňujú tieto zmeny plynule komunikovať s užívateľom. Viewmodel je naviazaný na model, ktorý v tomto prípade reprezentuje rozhranie pre komunikáciu so serverom.

3.5 Rozdelenie do modulov

Práca adresuje rôzne nezávislé funkčné požiadavky vďaka čomu je ju možné rozdeliť na menšie moduly, ktoré budú nezávisle na sebe vyvíjané. Každý modul pokryje celý vývojový proces od priblíženia problematiky až po testovanie.

Emailový klient poskytne užívateľské rozhranie pre prácu s príchodzími emailami. Užívateľ bude môcť na základe prijatého emailu vytvárať novú žiadosť, alebo ho pripojiť k už existujúcej. Rozhranie nahradí potrebu využívania externej aplikácie.

Životný cyklus aplikácie predstavuje hlavnú časť práce. Užívatelia tu budú môcť sledovať a dokumentovať vývoj danej žiadosti. Rozhranie poskytne funkcionalitu pre organizovanie a správu aktivít žiadosti.

Administrácia systému umožní užívateľom konfigurovať vstupné kanály, otvorené pozície a synchronizačné priečinky. Vďaka tomu nebude potrebné počas vytvárania žiadosti paralelne špecifikovať informácie, ktoré môžu byť do systému pridané v dobe, keď sa otvára nová pozícia.

Kapitola 4

Emailový klient

Táto kapitola sa bude zaoberať problematikou implementácie modulu emailového klienta. Úvod kapitoly poskytne prehľad výziev, ktoré je potrebné prekonať, aby modul fungoval správne. Nasledovať bude návrh riešení týchto výziev a detailný popis implementačných krokov. Záver kapitoly sumarizuje testovací proces.

4.1 Výzvy

Hlavnou úlohou tohto modulu je zabezpečiť synchronizáciu s emailovou schránkou Microsoft Cloud, čo umožní užívateľovi vytvárať na základe príchodzích emailov nové žiadosti o zamestnanie, alebo email asociovať s už existujúcou žiadosťou.

Synchronizácia emailov - Je potrebné implementovať synchronizačný mechanizmus, ktorý naplní databázu perzistentne uloženými kópiami emailov zo schránky Microsoft Cloud. Je rovnako dôležité aktualizovať už uložené emaily pre prípad, že sa zamestnanec rozhodne urobiť priamo v schránke nejaké úpravy. Synchronizácia musí zachovať pôvodný formát emailu, aby bolo možné v prípade chyby na strane schránky vymazaný email spätne vytvoriť.

Aktualizovanie dát - Systém sa bude aktualizovať v krátkych časových intervaloch. Aby sa zabránilo nadbytočnej záťaži v prípade, že schránka bude obsahovať stovky emailov musí byť implementovaný spôsob ako dáta len aktualizovať na základe rozdielu posledného a aktuálneho stavu schránky.

Oprávnenia - Dáta ku ktorým systém bude pristupovať nie sú verejné. Je potrebné implementovať spôsob overenia totožnosti, ktorý poskytne serveru prístup ku zdrojom v Microsoft Cloud a Azure.

Vizualizácia emailu - Email synchronizovaný do systému musí byť možné zobraziť v klasickom formáte ako pri používaní obyčajného emailového klienta, akým je napríklad Outlook. Pripnuté prílohy musí byť možné stiahnuť do lokálneho úložiska počítača. Je dôležité dbať na bezpečnosť užívateľa, vykreslený obsah emailu môže potenciálne obsahovať nechcené resp. nebezpečné informácie a tak poškodiť užívateľa.

4.2 Návrh

Táto sekcia najskôr v krátkosti vysvetlí aké možnosti sú k dispozícii pre zdolanie výziev. Nasledovať bude návrh riešenia s čiastočným databázovým modelom, ktorý zobrazuje kľúčové entity, ktoré figurujú v tomto module.

Graph API je REST API, ktoré poskytuje užívateľom prístup ku informáciám uloženým na Azure. Pre zjednodušenie prístupu Microsoft poskytuje knižnicu Graph, ktorá zaobaľuje jednotlivé HTTP požiadavky do objektov a funkcií, ktoré sa dajú priamo z kódu volať a spracovávať [32].

```
1  public void PrintMessages(string mailbox, string folder)
2  {
3    // Initialize Graph API client (differ based on access scenario)
4    GraphServiceClient client = GetClient();
5
6   // List messages of currently logged in user
7    List<Message> data = client.Users[mailbox].
8                              MailFolders[folder].
9                              Messages.GetAsync().Result.Value;
10   // Print retrieved messages
11   foreach (Message message in data)
12     Console.WriteLine($"{message.Subject} - {message.Body}");
13 }
```

Výpis kódu 4.1 Ukážkový kód využitia Graph API pre získanie emailov užívateľa

Delta query je pokročilá funkcia Graph API, ktorá umožňuje efektívne sledovať zmeny v dátach uložených v Azure. Je súčasťou odpovede z Graph API, predstavuje referenčný bod pre budúcu požiadavku. Vďaka tomu sa pri nasledujúcej požiadavke vracajú len také dáta, ktoré sa od poslednej požiadavky zmenili, čo zjednodušuje a zrýchľuje spracovanie dát [33].

```
1  public void PrintMessages(string mailbox, string folder)
2  {
3    // Initialize Graph API client (differ based on access scenario)
4    GraphServiceClient client = GetClient();
5
6    // List messages of currently logged in user along with dealta query
7    DeltaResponse response = client.Users[mailbox].
8                              MailFolders[folder].
9                              Messages.Delta.GetAsync().Result;
10   // Print retrieved messages
11   foreach (Message message in response.Value)
12     Console.WriteLine($"{message.Subject} - {message.Body}");
13
14   // Wait for 10 seconds
15   Thread.Sleep(10000);
16
17   // Request changes in messages through recieved delta query link
18   response = new DeltaRequestBuilder(response.OdataDeltaLink,
19                                      client.RequestAdapter
20                                      ).GetAsync().Result;
21   // Print retrieved messages
22   foreach (Message message in response.Value)
23     Console.WriteLine($"{message.Subject} - {message.Body}");
24 }
```

Výpis kódu 4.2 Ukážkový kód použitia delta query

Delegated access umožňuje aplikácii pristupovať ku dátam v mene užívateľa. Aplikácia aj užívateľ musia byť autorizovaní na vykonávanie požiadaviek. Tento spôsob prístupu umožňuje aplikácii manipulovať len s takými zdrojmi, ku ktorým má užívateľ prístup [34].

App-only access dovoľuje aplikácii manipulovať s dátami samostatne bez autorizovaného užívateľa. Vyžaduje dedikované oprávnenia, odsúhlasené administrátorom organizácie. Jej hlavným využitím je automatizácia, kedy aplikácia vykonáva nejakú službu na pozadí bez interakcie užívateľa [35].

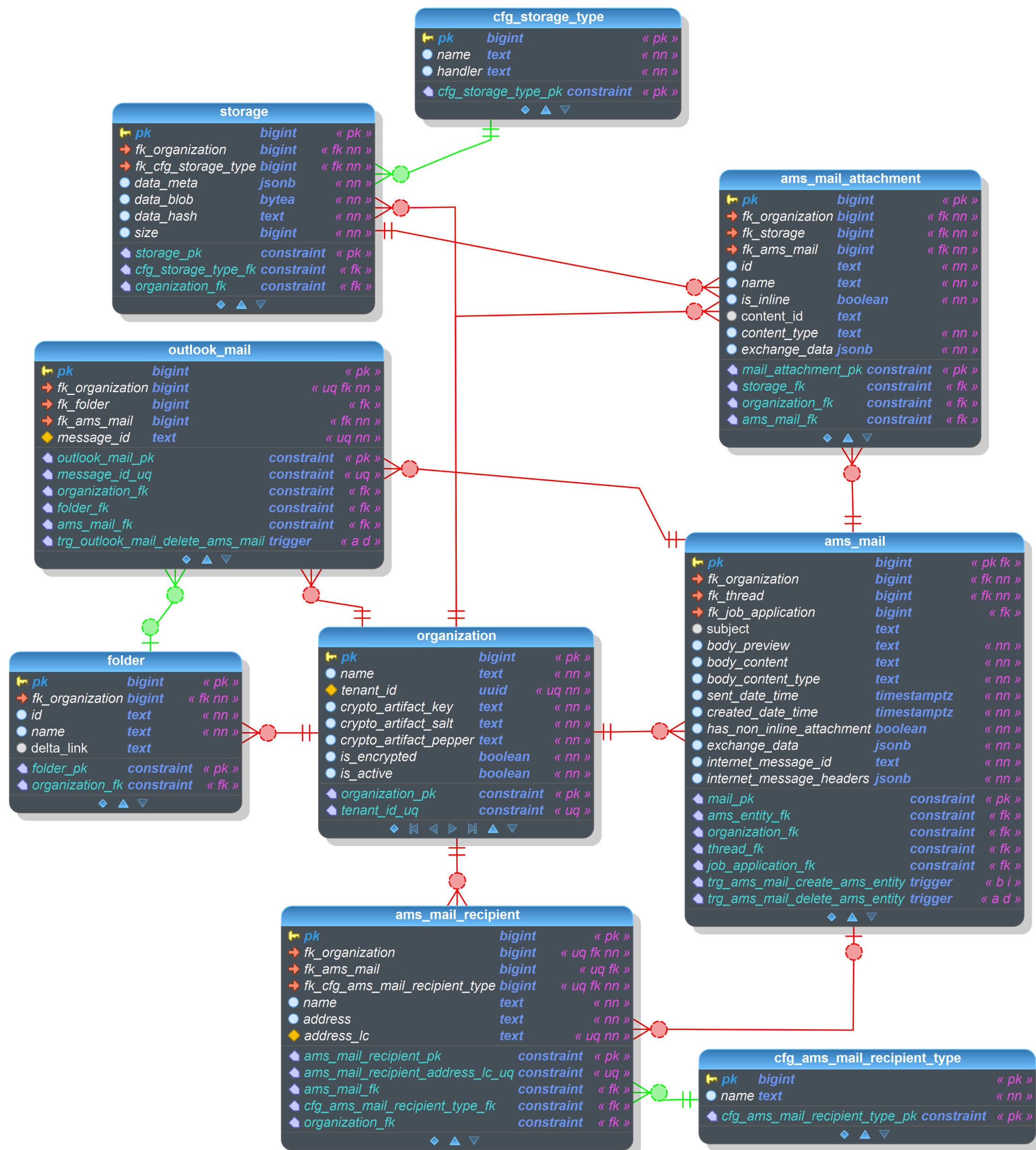
Na prístup ku dátam bude využitá knižnica Graph API. Pre potreby synchronizácie by za normálnych okolností stačil delegovaný prístup, ale zamestnanci budú pristupovať ku zdieľanej schránke, nie ku svojej osobnej. K tomu oprávnenia delegovaného prístupu nestačia, takže je potrebné využiť app-only access, ktorý vyžaduje oprávnenie od administrátora Azure tenanta. Aplikácia bude pristupovať primárne ku priečinkom v Microsoft Cloud a ich obsahu.

Entita reprezentujúca synchronizačný folder bude obsahovať informáciu a delta query z poslednej synchronizácie. Každá synchronizácia začne od posledného záchytného bodu, rozpozná rozdiely v aktuálnom stave schránky a načíta len potrebné zmeny. Po úspešnom dokončení procesu sa vykoná aktualizácia delta query. V prípade výpadku počas synchronizácie ostáva delta query nezmenená, a teda systém nestráca posledný záchytný bod a nedôjde ku strate dát.

Aby bolo možné spätne vytvárať chybne zmazané emaily z Microsoft Cloud schránky, musí entita reprezentujúca email obsahovať všetky informácie, ktoré je možné cez Graph API získať. Okrem typických vecí, ako je subjekt a obsah emailu, správy prichádzajú so štandardnými parametrami. Tieto parametre sa nazývajú internet message headers a Graph API ich komunikuje pomocou zoznamu hodnôt. Keďže sa s týmito hodnotami za bežných okolností nemanipuluje, budú reprezentované ako jeden stĺpec v tabuľke s objektom zakódovaným v JSON formáte. Ak systém bude potrebovať správu vložiť naspäť do schránky, učiní tak za pomoci Graph API, ktorá takúto operáciu podporuje.

Obsah emailu reprezentuje HTML súbor v textovom formáte. Blazor poskytuje podporu pre zobrazenie HTML dokumentov, avšak bez žiadnych bezpečnostných prvkov. Pre bezpečné používanie obsahu bude využitý iframe tag, ktorý je možné spustiť v móde sandbox. Sandbox mód zamedzí nechcenému vykonávaniu JavaScript kódu a iných automatizovaných operácií. Nie všetky na prvý pohľad podozrivé emaily, musia mať zlý úmysel. Môže sa napríklad jednať o obrázok, ktorého zdroj sa načítava z internetu, teda nie je súčasťou príloh. Pre tento prípad bude užívateľ mať možnosť email plnohodnotne načítať a na vlastné riziko takýto obsah zobraziť.

4.2.1 Databázový model



Obr. 4.1 Databázový model emailového klienta

4.3 Implementácia

4.3.1 Synchronizácia

Synchronizácia je proces, ktorý vykonáva server. V rozhraní klienta sa nachádza tlačítko, ktoré užívateľ môže využiť pre aktualizáciu práve zobrazených dát. Tento proces pozostáva z nasledovných krokov: načítanie dát cez Graph API, spracovanie dát, vloženie

dát do databázy a aktualizácia delta query.

Komunikáciu s užívateľským rozhraním zabezpečí controller. Jeho úlohou je vystaviť API na správu žiadostí potrebných pre rôzne úkony v rámci komunikácie s Microsoft Cloud a Azure. Pre popis rozhrania API sa využíva technológia Swagger, ktorá poskytne internému nástroju podklady pre vygenerovanie rozhrania na strane klienta. Tým sa ľahko zabezpečí kompatibilita medzi oboma stranami a počas vývoja ich nie je potrebné dodatočne synchronizovať.

Každá z entít v databázovom modeli má svoju vlastnú reprezentáciu v kóde. Pomocou proprietárneho frameworku Swift sa tieto entity napĺňajú a figurujú ako repository pre komunikáciu s databázou. Pomocou zdieľaného projektu sa propagujú do oboch častí implementácie zmenšené verzie týchto entít vo forme objektov určených na prenos dát.

Emaily sa synchronizujú po jednotlivých priečinkoch. Graph API obvykle stránkuje výsledky po desiatich, s možnosťou nastaviť počet výsledkov na požiadanie až do maximálnej hodnoty tisíc. Keďže sa nedá spoľahnúť na to, že v schránke dôjde ku maximálne tisíc zmien, posiela Graph API tzv. next link, ktorý ukazuje na ďalšiu stránku zmien. Sekvencia next linkov umožní systému synchronizovať kompletnú históriu zmien. Po synchronizovaní poslednej stránky Graph API odošle delta query ako referenčný bod pre ďalšiu synchronizáciu.

```
1  private static async Task<Tuple<List<DeltaResponse>, FolderEntity>>
2    SyncFolder(GraphServiceClient graphClient, FolderEntity folder, string mailbox)
3  {
4    List<DeltaResponse> deltaResponses = new List<DeltaResponse>();
5
6    DeltaResponse response;
7
8    // Check if folder was synchronized before
9    if (!string.IsNullOrEmpty(folder.DeltaLink))
10     response = await new DeltaRequestBuilder(folder.DeltaLink,
11                                              graphClient.RequestAdapter
12                                              ).GetAsync();
13
14   else
15     response = await graphClient.Users[mailbox].
16                                  MailFolders[folder.Id].
17                                  Messages.Delta.GetAsync();
18
19   deltaResponses.Add(response);
20
21   // Keep fetching messages until there are no more to fetch
22   while (!string.IsNullOrEmpty(response.OdataNextLink))
23   {
24     response = await new DeltaRequestBuilder(response.OdataNextLink,
25                                              graphClient.RequestAdapter
26                                              ).GetAsync();
27
28     deltaResponses.Add(response);
29   }
30
31   // Save latest delta link
32   folder.DeltaLink = response.OdataDeltaLink;
33
34   return Tuple.Create(deltaResponses, folder);
35 }
```

Výpis kódu 4.3 Kód pre synchronizáciu priečinku

Pri synchronizácii dát z Microsoft Cloud je zásadné, aby volania na Graph API nasledovali za sebou v čo najkratšej dobe. Tým sa minimalizuje pravdepodobnosť, že počas

procesu synchronizácie dôjde ku nečakanej zmene a následnej nekonzistencii v dátach, ak by zmena bola len čiastočne synchronizovaná. Keďže sú operácie pre jednotlivé priečinky na sebe nezávislé je možné ich vykonať paralelne, a po načítaní všetkých zmien z Graph API sekvenčne uložiť všetky zmeny do databázy. Pred vytvorením asynchrónnej operácie je možné skontrolovať, či sa priečinok stále nachádza v Microsoft Cloud schránke.

```
1  private static List<Task<Tuple<List<DeltaResponse>, FolderEntity>>>
2    GetSyncTasks(GraphServiceClient graphClient, string mailbox)
3  {
4    List<Task<Tuple<List<DeltaResponse>, FolderEntity>>> syncTasks =
5        new List<Task<Tuple<List<DeltaResponse>, FolderEntity>>>();
6
7    // Fetch folder list from Microsoft Cloud
8    List<MailFolder> firstRowFolders = graphClient.Users[mailbox].
9                                                MailFolders.GetAsync(x =>
10                                                   { x.QueryParameters.Top = 999; }
11                                               ).Result.Value;
12
13
14   // Fetch the rest of the folder tree
15   foreach (MailFolder folder in firstRowFolders)
16     FetchFoldersChildren(graphClient, mailbox, folder);
17
18   List<MailFolder> folders = new List<MailFolder>();
19   foreach(MailFolder folder in firstRowFolders)
20     AddFolderToList(folder, folders);
21
22   // For each folder from the database
23   foreach (FolderEntity folder in new FolderEntity.ListAll())
24   {
25     // Check if folder still exists in Microsoft Cloud and sync its messages if yes
26     if (folders.Exists(x => x.Id == folder.Id))
27       syncTasks.Add(SyncFolder(graphClient, folder, mailbox));
28     else
29       LogAccessProvider.Error.Log(
30         $"The system is trying to sync folder which does " +
31         $"not exist in the exchange mailbox. Mail id: {folder.Id}");
32   }
33
34   return syncTasks;
35 }
```

Výpis kódu 4.4 Kód pre paralelnú synchronizáciu zmien

Ukladanie dát do systému prebieha v niekoľkých korkoch. Po tom, čo sa načítajú všetky zmeny z Microsoft Cloud, sa rozdelia na dve skupiny. Na tie, čo opisujú nejakú zmenu v dátach, prípadne príchod nových dát, a tie, čo naopak dáta odstraňujú. Správy, ktoré Graph API poskytuje vždy opisujú priebeh zmien v rámci jedného priečinku. Preto bude presunutie emailu z jedného miesta na druhé v systéme reprezentované ako zmazanie starého emailu a prijatie nového. Pre ušetrenie zbytočných operácií nad databázou sa najskôr vykonajú všetky zmeny, ktoré dáta upravujú, vrátane tých, čo sa javia ako príchodzie. Potom sa zo systému odstránia emaily, ktoré už ďalej nefigurujú v Microsoft Cloud schránke.

Všetky zmeny prebiehajú v rámci jednej databázovej transakcie pre prípad, že by viacerí užívatelia využili možnosť synchronizácie v rovnakom čase, alebo by došlo ku chybe. Tým sa zabezpečí, že v databáze ostáva pôvodná delta query a najbližšia synchronizácia nestratí zmeny vykonané pri neúspechu.

```
public static SyncResponse Synchronize(SyncRequest request)
{
  // Load shared mail address from configuration
  string mailbox = SwiftConfig.Current.GetConfigPropertyValue<string>(
    "SharedMailAddress@MMIT.AMS.Server.Logic"
  );

  // Execute in database transaction
  DbAccessProvider.ExecuteInTransaction("Sync mails", () =>
  {
    GraphServiceClient graphClient = GetGraphClient();

    // Fetch tasks for synchronization
    List<Task<Tuple<List<DeltaResponse>, FolderEntity>>> syncTasks =
      GetSyncTasks(graphClient, mailbox);

    // Fetch all changes from Microsoft Cloud
    Task.WaitAll(syncTasks.ToArray());

    List<string> deleted = new List<string>();
    foreach (var responseFolderPair in syncTasks.Select(x => x.Result))
    {
      foreach (Message message in responseFolderPair.Item1.SelectMany(x => x.Value))
      {
        // Keep deleted messages for later
        if (message.AdditionalData.ContainsKey("@removed"))
        {
          deleted.Add(message.Id);
        }
        else
        {
          // Save changes to the database
          AmsMailEntity mail = LoadingOfAmsMail(message);
          mail.attachments = LoadAttachments(message);
          mail.Save();
        }
      }
    }

    // Remove all remaining messages
    foreach (string id in deleted)
        OutlookMailEntity.GetByMessageId(id)?.Delete();
  });

  return new SyncResponse();
}
```

Výpis kódu 4.5 Zjednodušená verzia funkcie pre synchronizáciu

Problém pri synchronizácii nastáva, keď sa snaží systém spárovať zmenu v Microsoft Cloud s dátami v databáze. Každý email v Microsoft Cloud má svoj unikátny identifikátor, ktorý je však platný len do bodu, kým sa email nepresunie do iného priečinku. Vtedy Graph API odkomunikuje túto skutočnosť ako zmazanie starého emailu a príchod nového.

Aby nedochádzalo ku zbytočnému mazaniu dát v databáze, je nutné, aby sa emaily rozpoznávali na základne fixne prideleného internet message id, ktoré vygeneruje emailový klient odosielateľa. Internet message id však nerozlíši viaceré kópie rovnakého emailu. Aby nedochádzalo ku nekonzistencii v databáze, je nutné rátať aj s takouto menej používanou funkcionalitou emailových klientov. Riešením je dekompozícia emailu na jeho dáta, identifikované cez fixné a nemenné internet message id, a jeho reprezentáciu v Microsoft Cloud s identifikátorom, ktorý mu bol pridelený.

```
OutlookMailEntity outlookMail = OutlookMailEntity.GetByMessageId(message.Id) ??
  OutlookMailEntity.Open(OutlookMailEntity.CreateIdentifierForEntity(null));

AmsMailEntity mail;
if (message.InternetMessageId != null)
  mail = AmsMailEntity.GetByInternetMessageId(message.InternetMessageId) ??
         AmsMailEntity.Open(AmsMailEntity.CreateIdentifierForEntity(null));
else
  mail = AmsMailEntity.Open(AmsMailEntity.CreateIdentifierForEntity(outlookMail.MailId));

mail.DevourMessage(message);
```

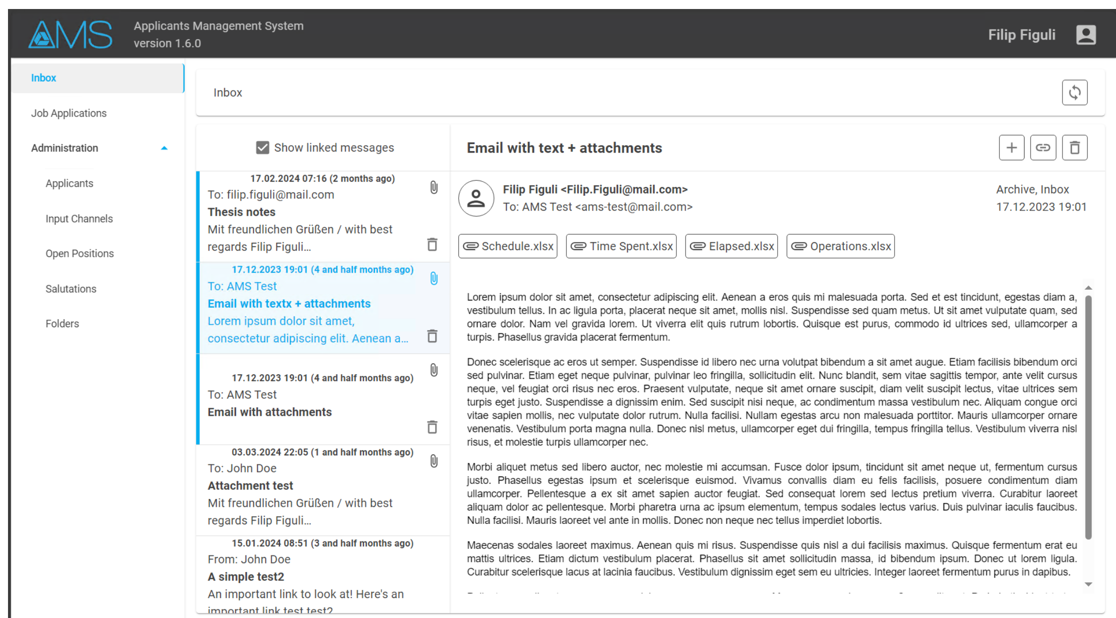
Výpis kódu 4.6 Ukážka kódu na rozpoznávanie pôvodu správy

Dátová reprezentácia emailu predstavuje úplnú kópiu všetkých informácií obsiahnutých v správach z Graph API. Pre uchovávanie internet message headers sa využíva objekt odložený vo formáte JSON, keďže emaily z rôznych zdrojov môžu obsahovať rôzne hlavičky a teda nie je možné im nastaviť fixný formát.

V prípade, že email prílohu obsahuje, si ju synchronizačný mechanizmus dodatočne vypýta zo serveru. Príloha si nesie základné informácie o jej názve, type a veľkosti sprevádzané binárnym súborom, ktorý sa uloží do databázy. Je dôležité spomenúť, že prílohy môžu byť aj vo formáte in-line, kedy sú priamo zakódované do obsahu emailu. Takéto prílohy sú špecificky označené a pri vytváraní obsahu emailu sa berú v úvahu.

4.3.2 Užívateľské rozhranie

Úlohou užívateľského rozhrania je umožniť užívateľovi náhľad a manipuláciu s emailami, ktoré sú synchronizované v systéme. Pre jednoduchosť používania je užívateľské rozhranie inšpirované klasickými emailovými klientmi, ktoré majú zoznam emailov naľavo s náhľadom vybraného emailu napravo.



Obr. 4.2 Užívateľské rozhranie emailového klienta

Stránka po načítaní odošle požiadavku na server, aby dostala zoznam synchronizovaných emailov. Ľavá strana obrazovky zobrazí políčka s náhľadmi emailov na ktoré môže užívateľ kliknúť pre zobrazenie detailu.

Úlohou užívateľského rozhrania je aj ochrana pred nechceným spustením škodlivého kódu. Obsah emailu sa skladá z niekoľkých častí, ktoré je potrebné všetky umiestniť na správne miesto. Túto funkcionalitu zabezpečuje $ProvideMailContent$, služba na serveri. Najskôr je potrebné nájsť všetky HTML tagy, ktoré ukazujú na externé zdroje, a pre bezpečnosť ich odstrániť z emailu. Užívateľ môže požiadavku opakovať pre dodatočné načítanie týchto zdrojov. Microsoft Cloud in-line obrázky odosiela ako regulárnu prílohu s tým rozdielom, že ich identifikátor je vložený na pôvodne miesto v texte emailu. $ProvideMailContent$ tieto miesta nájde a nahradí ich fyzickým binárnym kódom prílohy, aby sa mohla zobraziť užívateľovi.

Pre zabezpečenie obsahu proti nechcenému vykonaniu vloženého JavaScript kódu bol využitý tag iframe. Sandbox mód zabezpečí, že sa žiadny vložený kód nebude môcť vykonať.

```
1  public static MailContentResponse ProvideMailContent(MailContentRequest request)
2  {
3    // Get content of requested email
4    string content = AmsMailEntity.Open(
5        AmsMailEntity.CreateIdentifierForEntity(request.MailId)).BodyContent;
6
7    bool containsExternalSources = false;
8    if (!request.IncludeExternalSources)
9    {
10     // Clear external srouces if it is not specified not to
11     content = content.ExtractHtmlWithoutExternalSources(IsAnExternalSource,
12                                                          out containsExternalSources);
13   }
14
15   // Replace identifiers of all in-line attachmets by its real content
16   DataSourceParameters parameters = new DataSourceParameters();
17   parameters[0] = request.MailId.ToString();
18   foreach (AmsMailAttachmentEntity attachment
19     in new AmsMailAttachmentEntity.SelectionAllByMailId(parameters))
20   {
21     // Only applies to in-line attachment
22     if (attachment.IsInline)
23     {
24       // Create byte representation of an attachment content
25       byte[] storage = StorageEntity.Open(
26           StorageEntity.CreateIdentifierForEntity(attachment.StorageId)).Blob;
27
28       // Replace the identifier of an attachment with its real content
29       content = content.Replace("cid:" + attachment.ContentId,
30                                 "data:image/png;base64," + Convert.ToBase64String(storage));
31     }
32   }
33
34   // Return newly created content with inserted attachments
35   return new MailContentResponse
36   {
37     MailContent = content,
38     ContainsExternalSources = containsExternalSources
39   };
40 }
```

Výpis kódu 4.7 Ukážka kódu pre vytváranie HTML obsahu emailu

4.4 Testovanie

Cieľom pilotnej verzie softvéru je zistiť, či to, čo si zákazník objednal, je realizovateľné a spĺňa všetky jeho potreby. Zákazník si tak môže produkt fyzicky vyskúšať a rozhodnúť sa, či sú potrebné výrazné zmeny. Preto je softvér neustále upravovaný a nemá fixnú špecifikáciu, ktorá by rozlíšila pevne dané časti implementácie od tých, čo sa budú potenciálne v blízkom čase meniť. Vzhľadom k týmto okolnostiam všetko testovanie prebiehalo ručne tak, aby sa overilo splnenie funkčných požiadavok pre daný modul. Nemá zmysel pracne vytvárať automatizované testy ku funkcionalite, ktorá môže behom týždňa zaniknúť. Plnohodnotné testovanie je naplánované na termín, keď sa finálne uzavrie špecifikácia celého projektu a zákazník bude spokojný s realizáciou pilotnej verzie.

Napriek tomu je väčšina funkcionalít už historicky otestovaná v iných projektoch, keďže proprietárny framework Swift zabezpečuje väčšinu základných operácií mimo obchodnej logiky aplikácie. Tým pádom sa vývoj mohol sústrediť len na dosiahnutie funkčnej pilotnej verzie.

Počas vývoja bola každá časť implementácie dokumentovaná cez softvér pre sledovanie problémov Jira. Každá funkčná požiadavka sa ďalej vetvila do rôznych častí implementácie, ktoré museli byť úspešne uzavreté skôr ako sa vývoj posunul do ďalšej etapy.

Synchronizácia emailov - Pri testovaní synchronizačného mechanizmu boli využité rôzne úkony, ktoré podporuje Microsoft Cloud schránka. Išlo o odosielanie nových emailov, vytváranie kópií, presúvanie v rámci schránky, vymazávanie už synchronizovaných emailov a zmeny v ich stave, ako napríklad, či bol email prečítaný alebo označený vlajočkou.

Vzhľadom na zložitosť a nedostatky v dokumentácii Graph API, pôvodná implementácia nezohľadňovala rozmanitosť emailových identifikátorov, ktoré môžu vznikať pri presúvaní alebo kopírovaní dát. Dáta boli často duplikované alebo, v prípade zmazania jednej kópie emailu, systém odstránil všetky. Tieto chyby sa vyriešili oddelením identifikátora emailu z Microsoft Cloud od dátovej časti. Vzniká tak osobitná entita, ktorá reprezentuje rôzne kópie emailu v schránke.

Neúplnosť dokumentácie spôsobila aj opomenutie toho, že email, ktorému sa zmenil stav z neprečítaný na prečítaný a naopak, prichádza cez Graph API ako úplne prázdna správa s neinicializovanými hodnotami až na identifikátor a stav. Pre porovnanie, ľubovolná iná zmena, či už označenie vlajočkou, alebo presunutie do iného priečinku, je vždy odkomunikovaná ako kompletný prehľad aktuálnych dát, ktoré email obsahuje. Táto situácia spôsobila problém pri porovnávaní aktuálneho stavu dát so stavom perzistentne uloženým a dochádzalo ku strate informácií. Riešením bolo zachytávať tento špecifický prípad použitia osobitnou časťou implementácie, ktorá si sním poradila.

Funkcionalita zoznamu emailov - Úlohou zoznamu je zobraziť užívateľovi aktuálne sa nachádzajúce emaily v systéme. Rozdeľuje emaily na priradené a ešte nepriradené s tým, že nepriradené priorizuje. Užívateľ na každý email môže kliknúť a zobrazí sa mu jeho detail.

Táto časť neobsahuje komplikovanú logiku, ktorá by potrebovala výrazné testovanie. Zmeny boli implementované ako kompromis, kde políčko priradené každému emailu muselo obsahovať dostatok informácií. Užívateľské rozhranie zoznamu prešlo viacerými iteráciami, počas ktorých sa hľadalo ideálne rozloženie.

Počas testovania vznikol nápad, že by mohol užívateľ pomocou klávesnice prechádzať tento zoznam. Táto funkcionalita bola v nasledujúcej verzii implementovaná za pomoci prepojenia C# a JavaScript kódu a funkcionality Blazor frameworku.

Vizualizácia emailov - Hlavným cieľom vizualizácie emailu, okrem poskytnutia potrebných informácií ako email odosielateľa a subjekt, je zobrazenie jeho obsahu. Obsah sa prenáša vo forme HTML kódu, ktorý môže potenciálne obsahovať škodlivý obsah. Systém sa musí vedieť vysporiadať aj s takou správou bez rizika, že ohrozí užívateľa. Bolo vytvorených niekoľko desiatok emailov, ktoré mali imitovať rôzne komplikované správy, ale aj útočníkov a systém ich musel zobraziť.

Pri testovaní sa ukázalo, že iframe v sandbox režime je veľmi spoľahlivý nástroj pre tento účel. Nepodarilo sa narušiť jeho štruktúru tak, aby došlo ku spusteniu skriptov v obsahu emailu. Filtrovanie externých zdrojov tiež fungovalo bez problémov. Pre prípady, keď užívateľ určí, že obsah nie je nebezpečný, bolo pridané tlačítko, ktoré umožňuje načítanie aj externých zdrojov.

Testovacie scenáre

Synchronizácia emailov - Cieľom je otestovať funkčnosť synchronizácie emailov z Microsoft Cloud schránky.

Postup: Užívateľ sa prihlási do systému a v sekcii emailového klienta inicializuje synchronizáciu dát. Následne v externom emailovom klientovi urobí v schránke niekoľko z nasledovných zmien: pridanie nového emailu, kópia emailu, presunutie emailu do nového priečinku, zmazanie emailu. Užívateľ znovu synchronizuje dáta a zhodnotí, či sa systému podarilo dostať schránku do ekvivalentného stavu v akom sa nachádza externý emailový klient.

Výsledky: V prvotnej implementácii sa našli problémy spojené s kopírovaním emailu, čo spôsobovalo duplikáciu dát. Podobné problémy sprevádzalo kopírovanie emailov naprieč schránkou.

Zobrazenie detailu emailu - Cieľom je otestovať funkčnosť zobrazenia obsahu emailu a jeho zabezpečenie.

Postup: Užívateľ vytvorí email s jedným z nasledujúcich obsahov: JavaScript kód, obrázok s externým zdrojom, in-line obrázok, text. Takto vytvorený email odošle do testovacej schránky. Prihlási sa do systému a inicializuje synchronizáciu emailov. Pokúsi sa zobraziť obsah synchronizovaného emailu. V prípade, že sa jedná o obrázok s externým zdrojom, užívateľ využije možnosť dodatočného načítania plného obsahu.

Výsledky: Iframe v režime sandbox zabránil vykonávaniu ľubovolného JavaScript kódu. Funkcia pre odstraňovanie externých zdrojov funguje správne a dovoľuje dodatočné načítanie obsahu. Ľubovolný neškodný formát HTML kódu aj s in-line obrázkami sa úspešne zobrazí vo forme v akej bol pri odoslaní.

Vytváranie novej žiadosti - Cieľom je otestovať funkčnosť vytvárania novej žiadosti cez rozhranie emailového klienta.

Postup: Užívateľ vytvorí email simulujúc záujemcu o zamestnanie a odošle ho do zdieľanej schránky. V rozhraní emailového klienta inicializuje synchronizáciu. Otvorí si detail prijatého emailu a vyberie možnosť vytvoriť novú žiadosť. Na základe obsiahnutých informácií v emaily vyplní formulár a žiadosť odošle na spracovanie.

Výsledky: Testovanie ukázalo, že pre vytvorenie žiadosti musí užívateľ využiť administráciu, aby najskôr vytvoril žiadateľa. Ako riešenie tohto problému boli prerobené formuláre pre jednotlivé entity v databáze, tak aby podporovali znovupoužitie. Formulár sa následne pridal ku formuláru žiadosti kde si môže užívateľ zvoliť, že chce paralelne s vytváraním žiadosti vytvoriť aj žiadateľa.

Linkovanie emailu ku žiadosti - Cieľom je otestovať funkčnosť prepájania emailov ku existujúcim žiadostiam.

Postup: Užívateľ vytvorí email a odošle ho do zdieľanej schránky. V rozhraní emailového klienta inicializuje synchronizáciu. Otvorí si detail prijatého emailu a vyberie možnosť pripojiť ku existujúcej žiadosti. Z pola možností vyberie žiadosť ku ktorej chce email pripojiť. Odošle požiadavku na spracovanie.

Výsledky: Pole zobrazujúce možnosti pri vyberaní žiadosti nezobrazovalo dostatočne obsiahle informácie. Formát názvov v zozname bol zmenený na meno žiadateľa - názov otvorenej pozície - dátum vytvorenia žiadosti

Vymazanie emailu - Cieľom je otestovať funkčnosť odstránenia emailu zo systému.

Postup: Užívateľ sa prihlási do systému a v sekcii emailového klienta si zobrazí detail emailu, ktorý chce vymazať. Stlačí tlačítko pre vymazanie emailu.

Výsledky: Emaily boli vymazané zo systému, ale pretrvávali v Microsoft Cloud schránke. Bola doplnená implementácia, ktorá pri zmazaní emailu odošle cez Graph API požiadavku, aby sa rovnaká operácia vykonala aj v Microsoft Cloud schránke.

Kapitola 5

Životný cyklus žiadosti

Kapitola sa začína vysvetlením požiadaviek pre modul životného cyklusu žiadosti. Nasleduje návrh riešenia s detailným popisom dôležitých častí implementácie. Kapitolu zakončí popis testovacieho procesu a zmien, ktoré boli na základe testovania vykonané.

5.1 Výzvy

Proces životného cyklu žiadosti predstavuje zásadnú časť práce. Umožňuje zamestnancom presúvať žiadosť medzi rôznymi fázami, zaznamenávať jej vývoj a koordinovať súvisiace aktivity v rámci tímu.

Užívateľské rozhranie - Hlavnou výzvou životného cyklu žiadosti je navrhnúť vhodné užívateľské rozhranie, ktoré je intuitívne a jednoduché na používanie. Rozhranie by malo zahrňovať podrobnosti o žiadosti, vizualizáciu životného cyklu, prehľadné zobrazenie aktivít a súhrnnú správu s odporúčaniami pre ďalší postup.

Validácia vstupov - Aby sa zachovala jednotná štruktúra v dokumentovaní postupu, je potrebné implementovať validačné mechanizmy, ktoré nedovolia ďalší postup v žiadosti, kým sa nenaplnia všetky potrebné aktivity z danej fázy a nespíše sa súhrnná správa s odporúčaním pre ďalší postup. Validácie musia byť zreteľne zobrazené užívateľovi, aby bolo pochopiteľné, čo je potrebné urobiť pre ďalší pokrok v žiadosti.

Organizovanie aktivít - Systém musí implementovať flexibilný spôsob vytvárania aktivít. Napriek tomu, že je výberový proces rokmi zaužívaný a efektívny, v zmysle voľného spojenia by nemali byť jednotlivé aktivity pevne zakódované do systému. Každá aktivita musí mať priradenú zodpovednú osobu za jej vykonanie a poskytovať priestor pre dokumentáciu priebehu a hodnotenie žiadateľa.

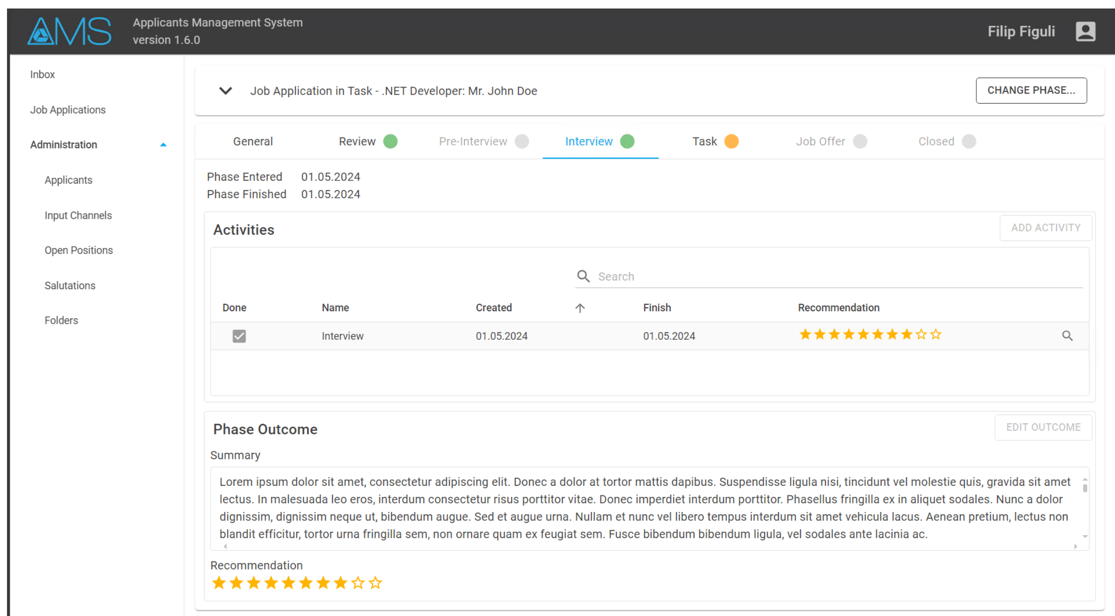
Spisovanie výstupov z fáz - Na konci každej fázy musí užívateľ zaznamenať jej priebeh a poskytnúť návrhy pre ďalšie kroky. Vďaka tomu sa môže ďalší postup lepšie prispôsobiť predchádzajúcim skúsenostiam so žiadateľom a adekvátne na ne reagovať.

Voľné prechádzanie medzi fázami - Výberové konanie nie je fixný proces. Na starosti ho má zamestnanec z ľudských zdrojov, ktorý sa môže rozhodnúť vykonať aj

netypické prechody medzi fázami. Fázy teda môžu byť preskočené, alebo aj vrátené naspäť.

5.2 Návrh

Webové aplikácie sa typicky odlišujú tým, že na rozdiel od statickej webstránky, ktorá sa vinie po vertikálnej ose pre zobrazenie ďalšieho obsahu, využívajú dynamický obsah. Takýto prístup používa aj užívateľské rozhranie produkované Blazor frameworkom. Vďaka tomu je možné využiť rôzne vizuálne efekty, ktoré napríklad skryjú alebo naopak zobrazia istú časť užívateľského rozhrania.



Obr. 5.1 Užívateľské rozhranie žiadosti o prácu

Užívateľské rozhranie životného cyklu žiadosti bude začínať panelom s nástrojmi, ktorý umožní vykonávať so žiadosťou základné operácie ako je zobrazenie detailu, úprava informácií či posun v rámci životného cyklu do ďalšej fázy. Zvyšok užívateľského rozhrania sa môže sústrediť na aktuálnu fázu. Každá fáza má svoje vlastné dáta, takže nie je potrebné zobrazovať medzi nimi prienik. Pre zjednodušenie navigácie medzi už vykonanými fázami a aktuálnou fázou bude využitý systém kariet, kde každá fáza bude reprezentovaná vlastnou kartou s indikátorom splnenia. Každá takáto karta musí obsahovať zoznam aktivít pre danú fázu a súhrnnú správu s odporúčaním.

Je nevyhnutné, aby komponenty použité v formulároch, ktoré vypĺňujú užívatelia, poskytovali nejakú spätnú väzbu v prípade chybných vstupov. Vzhľadom na to, že Blazor framework štandardne takúto funkcionalitu neobsahuje a tvorba vlastných komponentov by bola časovo náročná, bolo rozhodnuté, že sa využije jedna z dostupných komponentových knižníc. Na výber je viacero možností ako napríklad FluentUI, MutBlazor či MudBlazor, ale nie každá poskytuje v rámci svojich komponent validačné chyby. Preto bol vybraný MudBlazor, ktorý až na ojedinelé výnimky podporuje pre ľubovolný druh

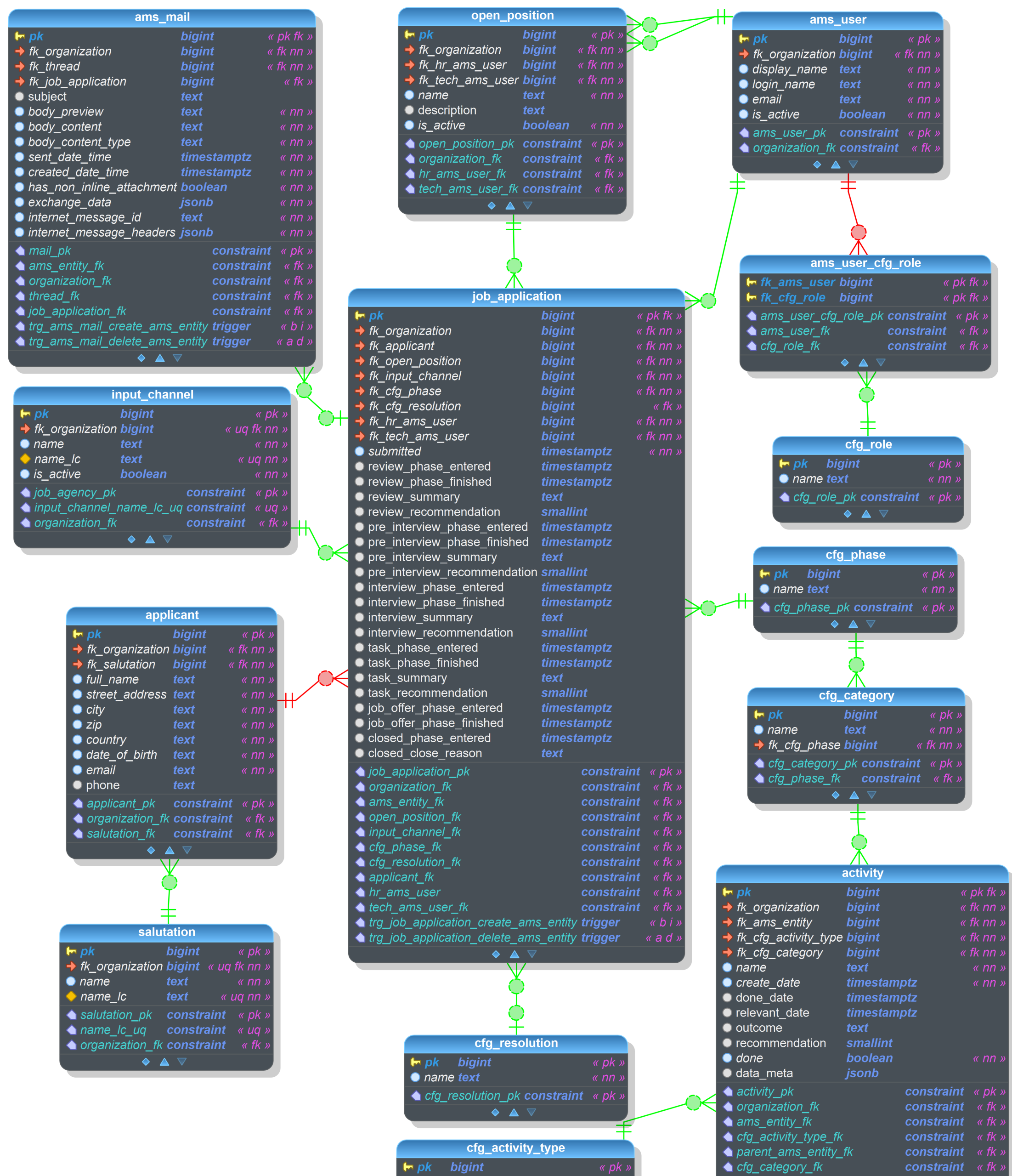
vstupu vizualizáciu validácie. Samotná validácia môže prebiehať na oboch stranách komunikácie, vzhľadom na integrovanú podporu však v proprietárnom frameworku Swift bude využitá validácia na strane serveru. Veľkou výhodou je, že takáto validácia umožní ochrániť server a integritu dát aj napríklad pri útoku mimo užívateľského rozhrania cez odosielanie náhodných požiadaviek na server. Jednotlivé objekty na prenos dát môžu za pomoci anotácií mať označené atribúty, ktoré sa majú pred vykonaním operácie skontrolovať. V prípade chybného vstupu sa správa vráti na klienta a komponenty MudBlazoru ju zobrazia užívateľovi.

Na podporenie opätovného použitia a ľahkej rozšíriteľnosti aktivity musia byť aktivity implementované pomocou špeciálneho stĺpca v databázovej tabuľke, ktorý bude obsahovať dodatočné údaje o objekte v JSON formáte. Tento prístup deleguje zodpovednosť za dodržiavanie štandardov pri zápise aktivít na implementáciu obchodnej logiky. Server pri každom načítaní aktivity overí jej typ a príslušný JSON súbor deserializuje do novej inštancie objektu. V zmysle rozšíriteľnosti aktivity musia byť implementované spôsobom, ktorý umožní jednoduché pridávanie nových aktivít a k nim sprievodných objektov. Každá aktivita sa skladá z názvu, typu a formulára pre jej vytvorenie, alebo úpravu.

Pre zabezpečenie dodržiavania pravidiel v rámci jednotlivých fáz životného cyklu žiadosti, budú zavedené viaceré druhy validácií. Každá fáza bude mať preddefinované špecifické pravidlá, ktoré sa musia uplatniť pred jej uzavretím. Ako prostriedok na zjednodušenie hodnotiaceho procesu bude zavedený systém hodnotenia založený na pocitovom vnímaní poverených osôb. Ich dojem z uchádzača zhodnotia pomocou slovného opisu, priebehu danej aktivity či fázy a číselným hodnotením pomocou hviezdičiek. Priemerné hodnotenie z týchto výstupov bude prezentované pri uzavretí žiadosti, čím poskytne súhrnný obraz o subjektívnych názoroch zamestnancov zapojených do procesu. Tento prístup umožní ostatným účastníkom výberového konania získať lepší prehľad o výkone a postupe kandidáta v predchádzajúcich fázach.

Keďže výberový proces má zohľadniť schopnosť zamestnanca subjektívne rozhodovať, musí implementovať spôsob ako sa pohybovať v rámci životného cyklu žiadosti mimo bežného postupu. Za bežných okolností by to nepredstavovalo problém, ale každý prechod medzi fázami musí dodržať špecifikované pravidlá danej fázy a zároveň inicializovať nasledujúcu. Tým vzniká veľké množstvo kombinácií, ktorými sa môže žiadosť pohybovať v životnom cykle. Každá takáto kombinácia musí dodržať konkrétnu validáciu a celá implementácia by mala poskytovať možnosť rozšírenia pre prípad, že sa pridá nová fáza do procesu výberového konania. Podobne ako v prípade aktivít sa musí ku týmto prechodom pristupovať tak, že sa definuje jedno rozhranie, ktoré bude prechody zabezpečovať, a doň sa pridajú len nové variácie v prípade potreby. Každý prechod bude definovať špecifickú validáciu, typ tranzície a kód, ktorý má server vykonať pre úspešné ukončenie zdrojovej fázy a inicializáciu cieľovej fázy.

5.2.1 Databázový model



Obr. 5.2 Databázový model žiadosti o zamestnanie

5.3 Implementácia

Najskôr bolo potrebné implementovať spôsob, akým môže užívateľ nahliadnuť na detail práve otvorenej žiadosti. Tieto informácie by nemali byť neustále súčasťou užívateľského rozhrania, keďže nie sú potrebné pre manipuláciu so žiadosťou. Preto bola vytvorená

vlastná komponenta *ExpansionPanel*, ktorá sa skladá z niekoľkých prepojených komponent MudBlazor knižnice. Funkcionalita schovávania je zabezpečená zapuzdrením v MudBlazor komponente *MudExpansionPanels*, ktorá sa pri interakcii užívateľa roztiahne a tak umožní nahliadnuť do jej obsahu. Obsahom komponenty sú údaje, ktoré užívateľ vypĺňa pri vytváraní žiadosti a tlačítko pre úpravy.

Vďaka komponente *MudTabs* z MudBlazor knižnice boli vytvorené jednotlivé karty pre každú fázu počínajúc kartou so všeobecnými informáciami, ako je súhrn všetkých aktivít naprieč celou žiadosťou. Karty sú ideálnym riešením v zmysle budúcej rozšíriteľnosti, keďže pridanie ďalšej potenciálnej fázy by znamenalo len rozšíriť už existujúci zoznam fáz o jednu kartu.

Horná časť karty zachytáva zoznam aktivít vo forme tabuľky, ktorú tvorí *MudDataGrid* naplňovaný dátami zo serveru. Každá takáto tabuľka využíva priložené tlačítko na vytvorenie novej aktivity, prípadne tlačítka nachádzajúceho sa priamo v nej ako posledný stĺpec pre úpravu a zmazanie daného riadku. Úpravu ako aj pridávanie novej aktivity zaobstaráva *MudDialog* s formulárom, ktorý validuje server. Spodná časť karty poskytuje formulár pre vyplnenie výstupu zo žiadosti. Formulár sprevádza aj *MudRating* komponenta obohatená o vlastnú implementáciu validácie.

Trieda *ActivityManager* bola navrhnutá tak, aby obsahovala všetky nevyhnutné funkcie pre aktivity a aby určovala, ktorá aktivita je zodpovedná za ich vykonanie.

```
1  public static partial class ActivityManager
2  {
3    public static List<IActivityBaseDescriptor> ActivityDescriptors
4      => new List<IActivityBaseDescriptor>
5      {
6        {new ActivityReviewDescriptor()},
7        {new ActivityMeetingDescriptor()},
8        {new ActivityApplicantTaskDescriptor()},
9        {new ActivityNegotiationDescriptor()}
10     };
11
12   public static UiAction ViewActivityDetailsInDialog()
13     => data =>
14     {
15       // Fetch activity type from the UI request
16       long? activityTypeId = data.ProvideMandatoryFragment<IActivityTypeProvider>().ActivityType;
17
18       if (activityTypeId == null)
19           throw new Exception("Dialog expects type of an activity as parameter.");
20
21       // Invoke function within the ActivityDescriptor
22       return ActivityDescriptors.Where(x => x.ActivityTypeId == activityTypeId)
23                                 .FirstOrDefault().ViewDetails().Invoke(data);
24     };
25
26   // More functions similar in nature to ViewActivityDetailsInDialog
27 }
```

Výpis kódu 5.1 Ukážka kódu triedy spravujúcej operácie nad aktivitami

Každá aktivita má svoj vlastný deskriptor, ktorý dedí *ActivityBaseDescriptor* a implementuje prepojenie spravovací formulára, zobrazovacieho formulára a objektov na prenos dát. Hlavnou výhodou takéhoto riešenia je veľmi jednoduché rozširovanie. Stačí vytvoriť novú inštanciu deskriptora, ktorý dedí z *ActivityBaseDescriptor*, a vložiť ju do *ActivityManager*.

```
public abstract class ActivityBaseDescriptor<TEditDialog, TViewDialog, TOpen, TUpdate>
  : IActivityBaseDescriptor
  where TEditDialog : AddEditSEFEntityDialog<TEditDialog, TOpen, TUpdate>
  where TViewDialog : ViewActivityDetailsDialog<TViewDialog, TOpen>
  where TOpen : SEFEntityPublishedTransport
  where TUpdate : SEFEntityPublishedTransport
{

  public abstract string Name { get; }
  public abstract long ActivityTypeId { get; }

  public UiAction OpenDialog()
    => AddEditSEFEntityDialog<TEditDialog, TOpen, TUpdate>.ActionEditExistingInDialog();

  public UiAction ViewDetails()
    => ViewActivityDetailsDialog<TViewDialog, TOpen>.ViewDetailsInDialog();
}
```

Výpis kódu 5.2 Ukážka kódu základnej triedy pre deskriptor aktivít

Užívateľské rozhranie využíva zoznam aktivít pre vytváranie interaktívnych menu tlačítok. Keďže Blazor framework plnohodnotne disponuje podporou jazyku C# a jeho funkcionalít, je napríklad možné vygenerovať zoznam položiek pri výbere typu aktivity priamo využitím zoznamu deskriptorov z triedy $ActivityManager$. To navyše umožňuje naviazať jednotlivé položky so správnym volaním adekvátnej funkcie a zároveň pre pridanie nového deskriptoru do zoznamu. Stačí ho dať k dispozícii triede $ActivityManager$.

```
<MudMenu Label="Add Activity" Variant="Variant.Outlined">
  @foreach (var activityDescriptor in ActivityManager.ActivityDescriptors)
  {
    <MudMenuItem OnClick="ActivityManager.AddNewActivityInDialog(activityDescriptor))">
      @activityDescriptor.Name
    </MudMenuItem>
  }
</MudMenu>
```

Výpis kódu 5.3 Ukážka kódu využitia jazyku C# pri generovaní obsahu

Podobný prístup bol použitý pri implementácii prechodu medzi rôznymi fázami. Tentokrát funkcionalitu zabezpečuje trieda $TransitionManager$, ktorá rozpoznáva jednotlivé deskriptory prechodov medzi fázami.

```
public abstract class TransitionBaseDescriptor<TDialog> : ITransitionBaseDescriptor
  where TDialog: PhaseTransitionDialog
{
  public abstract JobApplicationTransitionBase Transition { get; }

  public async Task<DialogResult> OpenDialogAsync(DataFragments dataFragments)
    => await OverlaysManager.Current.OpenDialogAsync<TDialog>(dataFragments, new DialogOptions());
}
```

Výpis kódu 5.4 Ukážka kódu deskriptor triedy pre prechod medzi fázami

Či sa má pre daný deskriptor otvoriť dialóg, rozhodne fixne nastavená dvojica zdrojovej a cieľovej fázy. Aby bolo možné každý prechod osobitne prispôsobiť, vznikol pre každú kombináciu prechodu jeden objekt zdedený z $JobApplicationTransitionBase$. Tieto ob-

jekty umožnia rôzne kombinovať špecifické druhy validácie a zároveň špecifikovať kód, ktorým sa majú zvyšné objekty inicializovať pri danom prechode.

```
public abstract class JobApplicationTransitionBase
{
  protected readonly long sourcePhase;

  protected readonly long targetPhase;

  protected abstract long ValidationKind { get; }

  public JobApplicationTransitionBase(long sourcePhase, long targetPhase)
  {
    this.sourcePhase = sourcePhase;
    this.targetPhase = targetPhase;
  }

  public bool ShouldOpenDialog(long sourcePhase, long targetPhase)
    => sourcePhase == this.sourcePhase && targetPhase == this.targetPhase;

#if SERVER
  protected abstract void PopulateData(JobApplicationEntity entity,
                                       JobApplicationTransitionRequest request);

  public void ApplyTransitionRequest(JobApplicationEntity entity,
                                     JobApplicationTransitionRequest request)
  {
    if(entity.CurrentPhase.ID == this.sourcePhase &&
       request.TargetPhase.Value == this.targetPhase)
    {
      if (entity.CurrentPhase.ID < request.TargetPhase)
      {
        if (!entity.CurrentPhaseActivitiesDone)
        {
          throw new CallFailedException(
            "It is not possible to leave this phase, " +
            "because there is at least one unfinished activity. " +
            "Please finish, or remove the offending activity from the phase.");
        }

        JobApplicationService.PerformValidation(request, this.ValidationKind);
      }

      PopulateData(entity, request);
    }
  }
#endif
}
```

Výpis kódu 5.5 Ukážka kódu základnej triedy pre prechod medzi fázami

Žiadosť sa môže z každej fázy presunúť do ľubovolnej inej fázy. Implementácia pomocou $TransitionManager$ triedy vždy rozpozná, aké dialógové okno sa má zobraziť, zabezpečí špecifickú validáciu pre daný prechod a napokon zavolá kód, ktorý správne ukončí aktuálnu fázu a inicializuje nasledovnú fázu. Je dôležité podotknúť, že samotná špecifikácia prechodov je implementovaná v zdieľanom projekte, čo zabezpečí integritu dát na oboch stranách. Takáto implementácia je možná vďaka direktívam kompilátora, ktoré je možné vidieť v implementácii $JobApplicationTransitionBase$.

5.4 Testovanie

Rovnako ako v prípade emailového klienta aj funkcionalita životného cyklu žiadosti je v pilotnej verzii a teda bude prechádzať neustálymi zmenami. Testovanie prebehlo špecifikáciou požiadaviek na funkcionalitu modulu, ktoré boli opakovane skúšané kým sa nepodarilo všetky naplniť.

Vzhľadom na to, že tento modul neobsahoval žiadnu komplikovanú logiku, testovanie sa zameriavalo na použiteľnosť užívateľského rozhrania, správnu validáciu vstupov od užívateľa a schopnosť žiadosti sa dynamicky prispôsobiť priebehu neštandardným prechádzaním fáz a vytváraním rôznych aktivít.

Užívateľské rozhranie - Pôvodný návrh usporiadania komponentov na obrazovke nebol praktický. Vrchnú časť obrazovky zaberal panel nástrojov, ktorý zobrazoval len názov aplikácie a tlačítko pre presun medzi fázami. Pre úpravu samotnej žiadosti sa musel užívateľ presunúť do administrácie a tam ju ručne upraviť. Tento problém vyriešil *ExpansionPanel*, ktorý po kliknutí poskytuje rozhranie pre úpravu žiadosti. Pre zlepšenie orientácie medzi fázami, boli pridané krúžky, ktoré indikujú stav danej fázy. Ukázalo sa, že implementácia *MudDataGrid* má tendenciu rásť do nekonečnej výšky, čo narúša vzhľad aplikácie tým, že spôsobuje vertikálne rolovanie stránky. Tabuľky boli zaobalené do vlastných komponentov, ktoré ju obmedzili na maximálnu výšku rodiča. Pri nadmernom množstve zápisov sa zobrazí možnosť rolovať tabuľku ako takú, nie celú stránku.

Validácia vstupov - Prebehla kontrola všetkých užívateľom vypĺňaných políčok v celej žiadosti. Našlo sa niekoľko zle inicializovaných validácií, ktoré umožnili nedefinované správanie aplikácie. Nie všetky validácie bolo možné zobraziť v rámci formulárov cez MudBlazor komponenty. S využitím *MudMessageBox* bola pridaná dodatočná informácia, napríklad, ak pre danú fázu neboli hotové všetky aktivity, aby užívateľ vedel, čo je potrebné urobiť pre dokončenie aktuálnej fázy. Tieto validačné chyby boli zachytávané v obchodnej logike aplikácie na serveri a následne cez špeciálne objekty na prenos dát komunikované s klientom.

Prechod medzi fázami - Po vyskúšaní rôznych kombinácií prechodu v žiadosti niektoré prechody vopred vyžadovali pridanie komplikovanejších validácií pre dodržanie úplnosti výberového konania. Preskočené fázy boli dodatočne označené odlišnou farbou. Pôvodná implementácia pri návrate do predošlej fázy zmazala všetky dáta vo fázach medzi zdrojovou fázou a cieľovou fázou. Bolo pridané varovanie pre užívateľa, že dôjde ku takejto zmene, keďže takéto správane nie je pre užívateľa intuitívne. Prípad, kedy je žiadosť zamietnutá bez klasického ukončenia výberového procesu, nebude viac vyžadovať úplne ukončenie všetkých aktivít v rámci fázy, kde sa práve nachádza. Tento scenár použitia popisuje situáciu, keď sa počas výberového procesu odhalí, že kandidát má značné nedostatky v požadovaných technických oblastiach, čo vedie zodpovednú osobu k rýchlemu ukončeniu žiadosti.

Vytváranie aktivít - Prvotná implementácia aktivít nepočítala so sledovaním ich naplnenia. Presun do ďalšej fázy je podmienený ukončením všetkých aktivít v aktuálnej fáze. Toto pravidlo bolo potrebné zakomponovať do samotného servisného spracovania na serveri, ktorý v prípade neúspechu oboznámi klienta o tomto stave, a ten

upozorní užívateľa cez *MudMessageBox*. Pre ľahšie manipulovanie s aktivitami boli do tabuľky pridané dodatočné dva stĺpce, ktoré umožňujú jednoduchú úpravu a zmazanie priamo z rozhrania práve zobrazenej fázy.

Testovacie scenáre

Vytvorenie aktivity - Cieľom je otestovať funkčnosť vytvárania novej aktivity v rámci žiadosti.

Postup: Užívateľ sa prihlási do systému. V sekcii žiadosti si zobrazí detail vybranej žiadosti. Vyberie aktuálnu fázu a otvorí dialógové okno pre vytvorenie novej aktivity. Vyplní formulár, priradí aktivitu zodpovednej osobe a odošle serveru na spracovanie.

Výsledky: Proces vytvorenia aktivity prebiehal bez problémov. Jediná úprava bola vykonaná v tabuľke aktivít, ktorá neobsahovala dostatočne veľa informácií.

Posunutie žiadosti do novej fázy - Cieľom je otestovať funkčnosť presúvania žiadosti do novej fázy v rámci výberového konania.

Postup: Užívateľ sa prihlási do systému. V sekcii žiadosti si zobrazí detail vybranej žiadosti. Zo zoznamu fáz vyberie fázu, kam sa má žiadosť presunúť. V prípade, že nie sú dodržané všetky potrebné validácie, ako splnenie všetkých aktivít či vypísanie výstupu fázy, žiadosť ostáva v pôvodnej fáze a užívateľovi sa zobrazí chyba s popisom problému. Užívateľ vyplní výstup fáze, hodnotenie priebehu fáze a dokončí všetky aktivity. Žiadosť sa úspešne môže presunúť do ďalšej fázy.

Výsledky: Prvotná implementácia nekomunikovala dostatočne to, čo je potrebné dodržať pre dokončenie fázy. Bolo pridané dialógové okno, ktoré ukáže užívateľovi aké akcie je potrebné vykonať pre ukončenie fázy. Pre prípad, kedy sa žiadosť zamietne bola vytvorená výnimka, ktorá nevyžaduje ukončenie všetkých aktivít vo fáze.

Úprava informácií v žiadosti - Cieľom je otestovať funkčnosť úpravy informácií v rozhraní žiadosti.

Postup: Užívateľ sa prihlási do systému. V sekcii žiadosti si zobrazí detail vybranej žiadosti. Pre zobrazenie detailu, užívateľ klikne na panel nástrojov v hornej časti obrazovky. Zvolí úpravu žiadosti stlačením tlačítka v detaile. Zobrazí sa dialógové okno s jednotlivými informáciami o žiadosti. Užívateľ vykoná zmenu a odošle požiadavku serveru na spracovanie.

Výsledky: Úprava žiadosti prebiehala úspešne bez žiadnych vážnych problémov. Jediná zmena voči prvotnej implementácii je pridanie tlačítka pre úpravu informácii žiadateľa pre prípad, že je potrebné počas vybavovania žiadosti informácie upraviť.

Dokumentácia priebehu fázy žiadosti - Cieľom je otestovať funkčnosť spisovania dokumentácie priebehu žiadosti.

Postup: Užívateľ sa prihlási do systému. V sekcii žiadosti si zobrazí detail vybranej žiadosti. Pre úpravu výstupu aktuálnej fázy otvorí dialógové okno. Vo forme textu zapíše potrebné informácie a zaznačí svoje subjektívne odporúčanie.

Výsledky: Funkcionalita fungovala správne okrem chýbajúcej vizualizáciu subjektívneho odporúčania. Komponenta *MudRating* z knižnice MudBlazor neobsahuje zobrazenie validačných chýb. Pre potreby práce bola vytvorená vlastná komponenta, ktorá *MudRating* zabalila a pridala chýbajúce validačné chyby.

Kapitola 6

Administrácia systému

Úlohou tejto kapitoly je priblížiť vývojový proces za návrhom a implementáciou administrácie systému. Najskôr sa čitateľ dozvie, aké výzvy sprevádzali tento proces. Následne odkryje návrh riešenia týchto výziev a napokon detailný popis dôležitých krokov implementácie. Kapitolu zakončí priebeh testovania s úpravami, ktoré boli potrebné pre dodržanie funkčných požiadaviek tohto modulu.

6.1 Výzvy

Administrácia systému je kľúčová z pohľadu konfigurácie jednotlivých mechanizmov. Systém by mal byť schopný prednastaviť potrebné informácie pre jednotlivé operácie, a tým napomôcť zamestnancom v procese spracovania a vybavovania žiadostí.

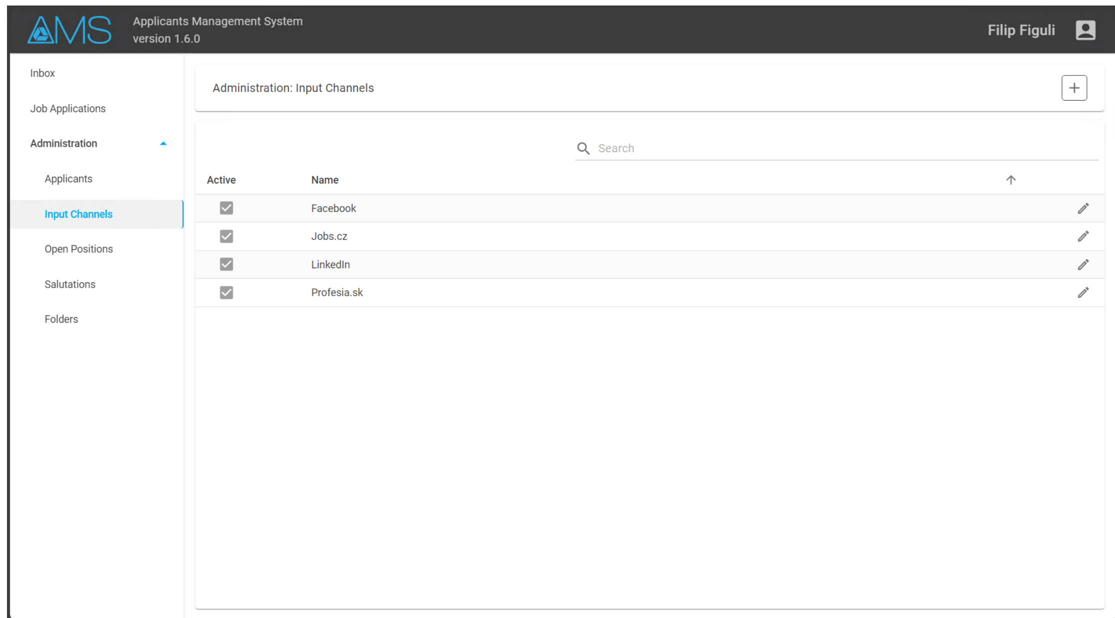
Základné nastavenie by malo reflektovať jednotlivé ponuky práce, ktoré firma zverejňuje. Pod tieto nastavenia spadajú vstupné kanály a jednotlivé ponuky. Zamestnanec, ktorý sa prihlási do systému, by nemal byť nútený paralelne so spracovaním žiadosti vytvárať aj tieto sprievodné entity. Administrácia zároveň poskytne stredisko pre špecifické upravovanie dát v databáze. Napríklad, ak by došlo ku preklepu v mene žiadateľa, bude možné tento údaj upraviť.

Okrem sprievodných informácií ku žiadosti musí administrácia poskytnúť funkcionalitu pre nastavovanie synchronizačných priečinkov. Priečinky sú organizované v stromovej hierarchii a ich názvy sa môžu za života aplikácie meniť. Užívateľ by mal byť oboznámený o tom, že v názve priečinku prebehla zmena a môcť túto zmenu aplikovať. V prípade, že dôjde ku zmazaniu priečinku, ktorý v danom momente obsahuje už synchronizované emaily, systém musí takéto emaily, archivovať, alebo odstrániť.

6.2 Návrh

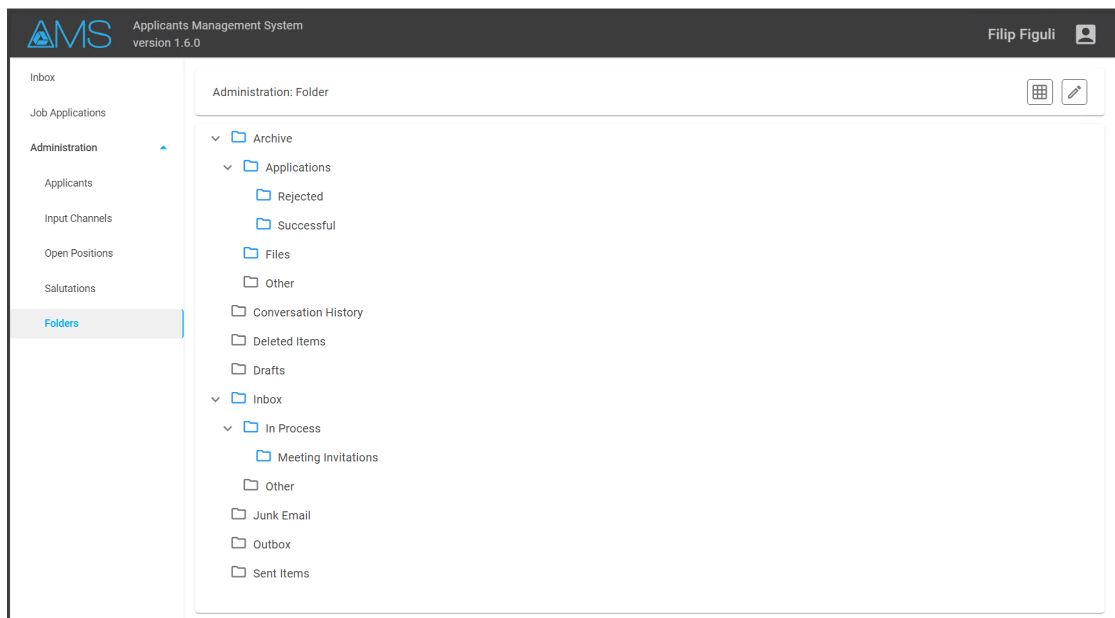
Užívateľské rozhranie jednoduchších častí administrácie, ako sú vstupné kanály či otvorené ponuky, využijú rovnaké komponenty, ako životný cyklus aplikácie. Hornú časť obrazovky zaplní panel nástrojov, ktorý užívateľovi zobrazí, na akú časť administrácie sa pozerá a zároveň poskytne interaktívne tlačítko pre pridanie nového zápisu do databázy. Pre všeobecný náhľad dát, ktoré sa nachádzajú v databáze, bude využitá tabuľka, do

ktorej sa mimo samotných dát z databázy vloží stĺpec pre úpravu daného riadka. Vytvorenie nového zápisu či úpravu existujúceho zabezpečí dialógové okno.



Obr. 6.1 Užívateľské rozhranie administrácie

Rozhranie administrácie priečinkov, podobne ako zvyšok administrácie, využije panel nástrojov pre interakciu s užívateľom. Skladá sa z tlačítka pre prepínanie zobrazenia v tabuľke, alebo v strome a tlačítka na úpravu.

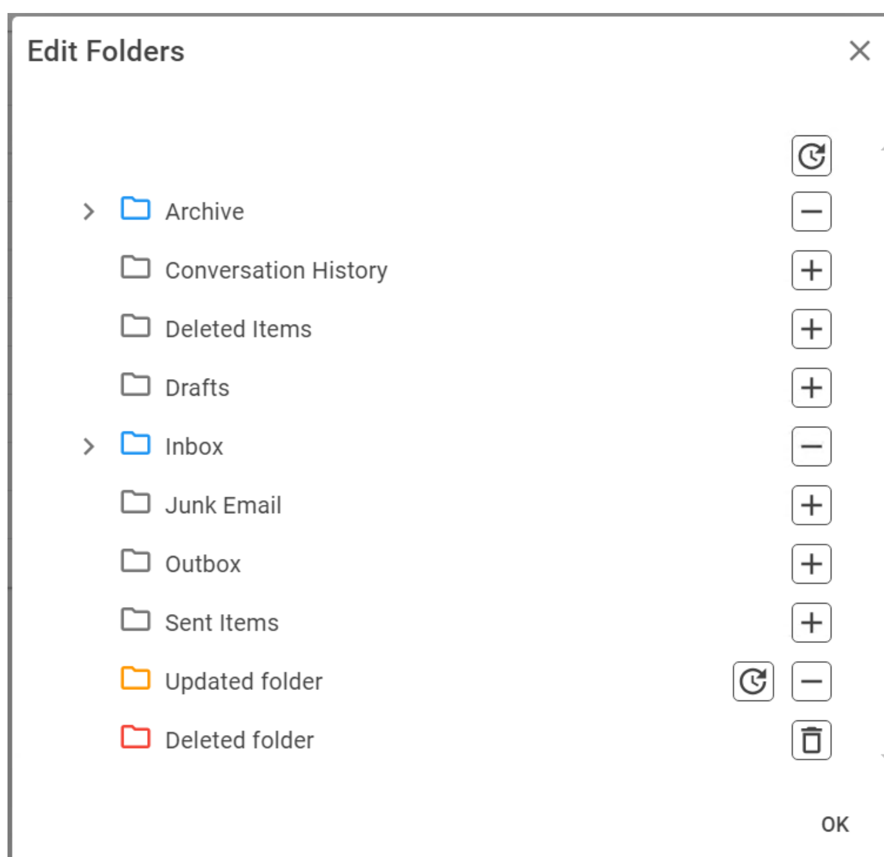


Obr. 6.2 Užívateľské rozhranie administrácie priečinkov

Pre lepšiu orientáciu v rozložení priečinkov je potrebné implementovať náhľad do stromovej štruktúry, avšak s možnosťou nazerať na dáta aj v prehľadnej tabuľke. Panel nástrojov teda okrem samotného tlačítka na úpravu tejto konfigurácie bude obsahovať tlačítko pre zmenu zobrazenia priečinkov z tabuľky na stromovú štruktúru a naopak.

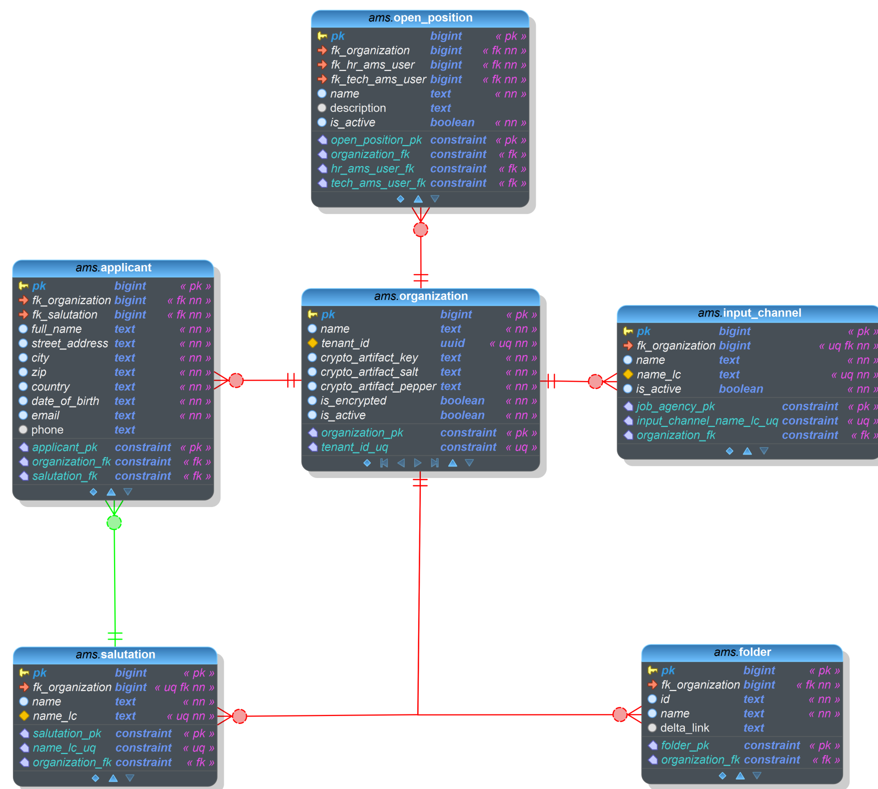
Vzhľadom na veľmi časté zmeny v stromovej štruktúre priečinkov v Microsoft Cloud schránke nemá zmysel si udržiavať stav štruktúry v databáze. Miesto toho bude systém udržiavať len názvy priečinkov. Vždy aktuálnu stromovú štruktúru poskytne požiadavka na Graph API. Keďže sa identifikátory priečinkov nemenia nič bez ohľadu na to, ako s nimi užívateľ manipuluje nie je potrebný komplikovaný systém synchronizácie. Postačí načítať do pamäte štruktúru a následne ju spárovať podľa identifikátorov s dátami v databáze.

Ak by mali byť dáta v databáze neustále aktuálne, vyžadovalo by to osobitnú službu, ktorá by v intervale niekoľkých minút opakovane aktualizovala stav databázy. Keďže aktuálnosť názvu priečinka nie je rozhodujúca informácia pri žiadnej činnosti systému, nebude potrebné implementáciu zbytočne komplikovať. Priečinky budú farebne odlíšené podľa ich stavu, či sa už jedná o aktuálny, upravený, alebo vymazaný priečinok. Užívateľ bude následne môcť rozhodnúť o tom ako sa systém zachová voči každému priečinku osobitne.



Obr. 6.3 Dialóg pre úpravu konfigurácie priečinkov

6.2.1 Databázový model



Obr. 6.4 Databázový model administrácie

6.3 Implementácia

Keďže väčšina administrácie zdieľa rovnaké rozloženie užívateľské rozhrania, bola vytvorená komponenta *AmsManagePagePart*, ktorá využíva *MudToolBar* a *MudDataGrid*. Každá konfigurovateľná entita má vlastnú stránku v administrácii, ktorá túto komponentu inicializuje. Ako parameter sa do nej posiela funkcia, ktorá otvára dialógové okno špecifické pre danú entitu. MudBlazor poskytuje v *MudDataGrid* podporu pre využívanie vlastných formátov stĺpcov, čo umožňuje vytvoriť špeciálnu verziu stĺpcu s tlačítkom a konfigurovateľnou funkciou, ktorá sa po kliknutí vykoná. Každá stránka administrácie je teda zodpovedná za dodanie špecifických funkcií pre danú entitu na ovládanie užívateľského rozhrania. Implementácia zo strany servera spočíva v jednoduchých CRUD operáciach a validáciou nad každým dialógom, aby sa predišlo neplatným dátam v systéme.

```
<MudStack Class="pa-4" Style="height: 100%">
  <MudPaper Elevation="1">
    <MudToolBar>
      <MudText>@Caption</MudText>
        <MudSpacer />
        @if (ActionAdd != null)
        {
          <MudTooltip Text="New entry">
            <MudIconButton Variant="Variant.Outlined"
                           Icon="@Icons.Material.Outlined.Add"
                           OnClick="OnClickHandler" />
          </MudTooltip>
        }
    </MudToolBar>
  </MudPaper>
  <div style="flex-grow: 1; min-height: 0">
    <ScDataSourceDataGrid @ref="this.dataGrid"
                          DataSourceDescriptor="@DataSourceDescriptor"
                          Columns="GridColumns" />
  </div>
</MudStack>
```

Výpis kódu 6.1 Ukážka kódu komponenty AmsManagePagePart

Implementácia administrácie priečinkov a ich následná synchronizácia vyžadovala vlastné riešenie. Základný problém pri synchronizácii priečinkov cez Graph API je neschopnosť API poslať kompletnú stromovú štruktúru. Pri odoslaní žiadosti na API je možné požiadať aj o potomkov práve zvoleného priečinku, ale odpoveď vždy zachádza len do prvej hĺbky stromu. V prípade, že je potrebné získať celý strom, je nevyhnutné posielať rekurzívne požiadavky, kým sa nepodarí celý strom spracovať.

```
private static void FetchFoldersChildren(GraphServiceClient graphClient,
                                         string mailbox,
                                         MailFolder folder)
{
   List<MailFolder> children = graphClient.Users[mailbox].
                                           MailFolders[folder.Id].
                                           ChildFolders.GetAsync(x =>
                                           { x.QueryParameters.Top = 999; }).Result.Value;

   folder.ChildFolders = new List<MailFolder>();
   foreach (MailFolder child in children)
   {
     FetchFoldersChildren(graphClient, mailbox, child);
     folder.ChildFolders.Add(child);
   }
}
```

Výpis kódu 6.2 Ukážka kódu rekurzívnej funkcie pre načítavanie stromu priečinkov

Funkcia *FetchFoldersChildren* sa spolieha, že daný priečinok nebude mať viac ako 999 potomkov. Táto hodnota vychádza zo samotnej limitácie Graph API, ktorá pre väčší počet výsledkov v jednej požiadavke využíva už spomínaný next link. Keďže účelom aplikácie je zefektívniť a urýchliť výberové konanie, je predpoklad, že by firma mala tendenciu využiť menej chaotickú emailovú schránku, ktorá by svojou štruktúrou prispievala skôr ku efektivite ako ju komplikovala tisícami priečinkov. Načítané dáta sa ukladajú postupne do stromovej štruktúry a na konci rekurzie ostáva koreňový objekt.

Načítané dáta sa ukladajú do triedy *FolderData*, ktorá sprostredkúva prenos informácií na klienta a späť pre synchronizáciu s databázou. Súčasťou týchto dát sú aj

priečinky, ktoré sa nepodarilo viac nájsť v schránke Microsoft Cloud a je potrebné odkomunikovať s užívateľom túto skutočnosť.

```
public static FolderSyncResponse SynchronizeFolders(FolderSyncRequest request)
{
  GraphServiceClient graphClient = GetGraphClient();
  string mailbox = SwiftConfig.Current.GetConfigPropertyValue<string>(
    "SharedMailAddress@MMIT.AMS.Server.Logic");

  // List all folders that are stored within the database
  List<FolderEntity> databaseData = new FolderEntity.ListAll().Items.ToList();

  // List first row of folders inside Microsoft Cloud mailbox
  List<MailFolder> folders = graphClient.Users[mailbox].
                                       MailFolders.GetAsync(x =>
  { x.QueryParameters.Top = 999; }).Result.Value;

  // Fetch the rest of the folder tree within mailbox
  foreach (MailFolder folder in folders)
    FetchFoldersChildren(graphClient, mailbox, folder);

  FolderData root = new FolderData();
  root.Children = new HashSet<FolderData>();

  // Transform Graph API objects into folder DTOs
  foreach (MailFolder folder in folders)
    root.Children.Add(GetFolderData(graphClient, mailbox, folder, databaseData));

  // Create list that can be iterated through
  List<MailFolder> folderList = new List<MailFolder>();
  foreach (MailFolder folder in folders)
    AddFolderToList(folder, folderList);

  // For each folder that does not exists in Microsoft Cloud mailbox
  // create placeholder to inform the user
  foreach (FolderEntity folder in new FolderEntity.ListAll())
  {
    if (!folderList.Exists(x => x.Id == folder.Id))
    {
      FolderData deletedFolder = new FolderData();

      deletedFolder.SIdentifier = folder.GetSIdentifier();
      deletedFolder.Id = folder.Id;
      deletedFolder.OriginalName = folder.Name;
      deletedFolder.OutlookName = folder.Name;
      deletedFolder.IsDeletedFromOutlook = true;
      deletedFolder.Children = new HashSet<FolderData>();
      deletedFolder.IsActive = true;

      root.Children.Add(deletedFolder);
    }
  }

  return new FolderSyncResponse() { RootFolderData = root };
}
```

Výpis kódu 6.3 Ukážka kódu funkcie zodpovednej za sprostredkovanie dát priečinkov z Graph API na klienta

Klient vďaka komponente *MudTreeView* dáta zobrazí v stromovej štruktúre. Farba jednotlivých ikon priečinku zobrazuje, v akom vzťahu sa daný priečinok nachádza voči dátam v databáze. Sivá ikonka symbolizuje priečinok, ktorý existuje v schránke, ale nie je súčasťou synchronizačného procesu systému. Modrá ikonka vyznačuje priečinky, ktoré sú synchronizované do systému a ich názov je stále aktuálny a zhodný s tým čo užívateľ môže vidieť v schránke. Žlté priečinky označujú tie, ktoré boli v schránke premenované

a užívateľ má možnosť tento nový názov do systému synchronizovať. Napokon červené priečinky reprezentujú tie, ktoré sa v systéme nachádzajú, už viac ale nie sú súčasťou Microsoft Cloud schránky. Pre každý z týchto stavov existujú akcie, ktoré užívateľ môže vykonať kliknutím na tlačítko, ktoré sa nachádza na konci riadku pre daný priečinok v stromovej štruktúre.

```
public static FolderUpdateResponse UpdateFolder(FolderUpdateRequest request)
{
  // If is folder active persist it in the database
  if (request.IsActive)
  {
    // Find folder within the database or create new
    FolderEntity folder = FolderEntity.GetById(request.FolderId) ??
                          FolderEntity.Open(FolderEntity.CreateIdentifierForEntity(null));

    // If folder is new it needs to be initialized
    if (folder.IsNew)
    {
        folder.Name = request.Name;
        folder.Id = request.FolderId;
        folder.DeltaLink = null;
        folder.Save();
    }
  }
  // If is not active check if is in the database and delete it if so
  else
  {
    FolderEntity folder = FolderEntity.GetById(request.FolderId);
    // Check if folder was stored within the database
    if (folder != null)
    {
      // Fetch mails connected to the folder
      DataSourceParameters parameters = new DataSourceParameters();
      parameters.LoadFromStringList(new List<string>()
                                   { $"{folder.EntityIdentifier.EntityId.Value}" });
      var mails = new OutlookMailEntity.ListAllByFolderId(parameters).Items.ToList();
      // If the reuqest is forced delete the folder and all of its mails
      if (request.Force)
      {
        foreach (OutlookMailEntity mail in mails)
            mail.Delete();

        folder.Delete();
      }
      else
      {
        // If folder contains any mails that are connected to any Job Applicaiton
        if (mails.Exists(x => !x.IsAssigned))
          return new FolderUpdateResponse() { FolderContainsSyncedMails = true };
        else
        {
          // Delete all mails and the folder
          foreach (OutlookMailEntity mail in mails)
              mail.Delete();

          folder.Delete();
        }
      }
    }
  }

  return new FolderUpdateResponse();
}
```

Výpis kódu 6.4 Ukážka kódu funkcie zodpovednej za uloženie zmien v priečinkoch do databázy

Každá akcia pridania alebo odstránenia priečinku z konfigurácie je spracovaná funkciou *UpdateFolder* v rámci synchronizačnej služby. Funkcia určuje, či je priečinok nový a inicializuje ho alebo odstráni už nepotrebné dáta. V prípade odstránenia priečinku funkcia skontroluje, či sa v ňom ešte nachádzajú nejaké emaily. Ak áno, upozorní užívateľa, ktorý môže operáciu opakovať, čím potvrdí, že si je tejto straty dát vedomý. Emaily, ktoré sú už pripojené k nejakej žiadosti, sa pre konzistenciu dát presunú do abstraktného neexistujúceho priečinku.

```
1   public static FoldersNamesUpdateResponse UpdateFoldersNames(FoldersNamesUpdateRequest request)
2   {
3     foreach (IdValuePairCore<string, string> folderIdNamePair in request.UpdatedFoldersNames)
4     {
5       FolderEntity folder = FolderEntity.GetById(folderIdNamePair.ID);
6       folder.Name = folderIdNamePair.Value;
7       folder.Save();
8     }
9
10    return new FoldersNamesUpdateResponse();
11  }
```

Výpis kódu 6.5 Ukážka kódu funkcie zodpovednej za aktualizáciu mien priečinkov v systéme

Napokon ostáva funkcia *UpdateFoldersNames*, ktorá prijíma zoznam priečinkov a postupne aktualizuje ich názvy. Implementácia klienta umožňuje jedným tlačítkom synchronizovať názvy naprieč celým stromom priečinkov v prípade, že by užívateľ nechcel osobitne každý z nich aktualizovať.

6.4 Testovanie

Najskôr prebehlo testovanie užívateľského rozhrania a jeho používania. Aby sa zjednotilo rozhranie celej aplikácie s ostatnými časťami, boli vymenené všetky tlačítka za ikony. Tie však na prvý pohľad nie vždy dostatočne zachytávali podstatu funkcie tlačítka, preto boli následne pridané popisky pomocou *MudTooltip* naprieč celou aplikáciou. Podobným spôsobom sa pridali aj vysvetlivky ku ikonám pri úprave konfigurácie synchronizovaných priečinkov. Pre zlepšenie orientácie pri zobrazení priečinkov v tabuľke bola ku každému priečinku pridaná aj cesta formou nového stĺpca v tabuľke.

Pôvodná implementácia synchronizácie priečinkov neukladala názvy. Vďaka tomu nebolo potrebné vykonávať väčšinu spomínaných aktivít, keďže sa len načítala stromová štruktúra cez Graph API a bola spárovaná s unikátnymi identifikátormi v databáze. Testovanie však ukázalo komplikácie. Pri orientácii užívateľa v rámci schránky a pri zložitejších priečinkových štruktúrach sa emaily zle hľadali, preto bola implementácia kompletne zmenená, aby podporovala synchronizáciu názvov tak, ako bola popísaná v sekcii Implementácia tejto kapitoly.

Samotný synchronizačný proces prebiehal bez problémov. Keďže Microsoft Cloud používa nemenné identifikátory pre priečinky, nenastáva väčšina problémov spojená s meniacim sa identifikátorom, ako to bolo v prípade synchronizácie emailov. Rovnako Graph API neponúka zmeny vo forme správ o tom, že bol priečinok vymazaný alebo presunutý. Systém si načíta aktuálny stav v schránke a k nemu presne spáruje dáta z databázy. Tento proces je priamočiary a teda nie je veľký priestor na výskyt chýb.

Testovacie scenáre

Pridanie nového zápisu do konfigurácie systému - Cieľom je otestovať funkčnosť pridávania nového zápisu do konfigurácie systému

Postup: Užívateľ sa prihlási do systému. V sekcii administrácia vyberie vstupné kanály. Kliknutím na tlačítko v panele nástrojov otvorí dialóg pre pridanie vstupného kanálu. Vyplní informácie a odošle serveru na spracovanie.

Výsledky: Pridávanie vstupného kanálu prebiehalo úspešne avšak bolo možné vytvoriť duplikátne záznamy. Tie v kontexte, v akom sa práca pohybuje nemajú význam a spôsobovali by len nejednotnosť dát. Bola preto pridaná validácia, ktorá zabezpečí, že sa v databáze v danom čase môže nachádzať len jeden vstupný kanál s rovnakým názvom.

Úprava zápisu v konfigurácii systému - Cieľom je otestovať funkčnosť úpravy existujúceho zápisu v konfigurácii systému

Postup: Užívateľ sa prihlási do systému. V sekcii administrácia vyberie vstupné kanály. V tabuľke vyberie záznam, ktorý chce upraviť a klikne na tlačítko v poslednom stĺpci tabuľky pre úpravu. Vykoná potrebné úpravy v dialógovom okne a odošle serveru na spracovanie.

Výsledky: Úprava vstupného kanálu prebiehala úspešne bez nejakých dodatočných úprav.

Pridanie nového priečinku do synchronizácie - Cieľom je otestovať funkčnosť pridania priečinku do konfigurácie.

Postup: Užívateľ sa prihlási do systému. V sekcii administrácia vyberie administráciu priečinkov. V paneli nástrojov vyberie úpravu konfigurácie priečinkov. Zo zobrazeného stromu vyberie priečinok, ktorý chce pridať do synchronizácie a stlačí tlačítko pridania.

Výsledky: Synchronizácia priečinkov pôvodne fungovala tak, že si užívateľ mohol vybrať viacero priečinkov naraz a synchronizovať ich uložením dialógu. Táto implementácia však nebrala v úvahu ukladanie názvov priečinkov a synchronizáciu týchto názvov. Po pridaní synchronizácie názvov bolo nevyhnutné spraviť jednotlivé úkony ako dedikované akcie. Tým sa predišlo kolíziám v prípade prístupu viacerých osôb v rovnaký čas.

Odobratie priečinku zo synchronizácie - Cieľom je otestovať funkčnosť odobratia priečinku z konfigurácie.

Postup: Užívateľ sa prihlási do systému. V sekcii administrácia vyberie administráciu priečinkov. V paneli nástrojov vyberie úpravu konfigurácie priečinkov. Zo zobrazeného stromu vyberie priečinok, ktorý chce odobrať zo synchronizácie a stlačí tlačítko pridania.

Výsledky: Samotné odobratie priečinku prebiehalo úspešne. Pre užívateľa bol problém rozlíšiť, ktoré priečinky sú synchronizované a ktoré nie, preto bola implementovaná paleta farieb, ktorá ich typovo vizuálne odlišuje.

Aktualizovanie názvu priečinku - Cieľom je otestovať funkčnosť aktualizácie názvu priečinku v konfigurácii.

Postup: Užívateľ sa prihlási do systému. V sekcii administrácia vyberie administráciu priečinkov. V paneli nástrojov vyberie úpravu konfigurácie priečinkov. Zo zobrazeného stromu vyberie priečinok, ktorý nemá aktualizovaný názov a stlačením tlačítka ho aktualizuje.

Výsledky: Aktualizácia názvov fungovala správne. Keďže je nezvyčajné, aby užívateľ aktualizoval len jeden priečinok a zvyšné nechal neaktuálne, bolo pridané tlačítko, ktoré umožní synchronizovať všetky priečinky naraz.

Kapitola 7

Nasadenie

Táto kapitola popíše proces nasadenia aplikácie do Microsoft Azure. Najskôr je potrebné nakonfigurovať Azure Resource, potom registrovať aplikáciu a napokon publikovať kód aplikácie na Azure. Tento postup bude v následných sekciách popísaný.

Pri vytváraní zdroju bola potrebná asistencia zodpovednej osoby za správu tenanta firmy Mönkemöller IT GmbH, keďže ku manipulácií s platenými zdrojmi spoločnosti sú potrebné špeciálne oprávnenia.

7.1 Konfigurácia Azure

Služby Microsoft Azure, známe ako resources, sú usporiadané do skupín nazývaných resource groups. Tieto skupiny umožňujú správu oprávnení, lokalizáciu metadát a monitorovanie nákladov. Každá infraštruktúra musí byť zaradená aspoň do jednej resource groupy.

Prvý krok je vytvorenie resource groupy. Je odporúčané používať systematické pomenovanie pre lepšiu organizáciu všetkých prvkov v Azure. Lokalizácia nie je pre testovaciu verziu projektu dôležitá.

Infraštruktúra ďalej vyžaduje virtuálnu sieť, ktorá umožňuje komunikáciu medzi resourcami. Táto sieť napodobňuje jej fyzický ekvivalent, ktorý by bol využitý v prípade on-premise riešenia. Každý prvok siete by mal mať svoju vlastnú podsieť, alebo byť súčasťou už existujúcej.

Ďalší krok je vytvorenie PostgreSQL databázy. Pre testovacie účely postačí najmenšia možná veľkosť. Pre zabezpečenie sa využije vstavaná autentifikácia PostgreSQL, ktorá umožňuje prístup pomocou hesla. Netreba zabudnúť nastaviť DNS pre rozpoznávanie adries podľa názvov.

Keďže PostgreSQL neumožňuje konfiguráciu obmedzujúcu externú komunikáciu s databázou, z bezpečnostných dôvodov ju umiestňujeme do vnútornej virtuálnej siete. Vďaka tomu k nej môžu pristupovať len resource pridané do siete a teda schválené administrátorom. Pre prístup z vonku je potrebné pridať do siete malý virtuálny stroj s minimálnymi požiadavkami, ktorý umožní komunikáciu cez protokol SSH.

Pred vytvorením samotnej webovej aplikácie je potrebné alokovať potrebné zdroje. Takáto alokácia sa nazýva app service plan a predstavuje výber balíčku pozostávajúci z

CPU, RAM a miesta na disku. Následne je možné v rámci app service plánu vytvoriť web app, ktorá bude prevádzkovať aplikáciu. Pri vytvorení je potrebné špecifikovať platformu .NET8, operačný systém Linux a typ zverejnenia code, keďže bude na ňu nahrávaný priamo kód projektu.

Aby bola aplikácia dostupná na internete je potrebné nastaviť doménu aplikácie. Pre konfiguráciu aplikácie sa využíva Azure portal, kde je možné špecifikovať pre danú inštanciu aplikácie špeciálne premenné. Tieto premenné potom sprostredkúva proprietárny framework Swift zvyšku implementácie.

7.2 Registrácia aplikácie

Aby aplikácia mohla komunikovať so zvyšnými službami, ktoré poskytuje Microsoft ako napríklad Graph API, musí sa vedieť preukázať. Na preukázanie slúži registrácia aplikácie v Azure portáli. Všeobecne je dobrým zvykom vytvárať pre každú časť aplikácie osobitnú registráciu takže server aj klient by mal mať svoju vlastnú.

Pre vytvorenie registrácie je potrebné ísť na Azure portál a vybrať z pomedzi služieb App registrations. Pri vytváraní registrácie sa špecifikuje jej názov, ktorý primárne slúži na interné rozpoznávanie jednotlivých registrácií a typ podporovaných účtov. Tu je možné zvoliť, že ku aplikácii budú pristupovať užívatelia z rôznych tenantov, alebo len užívatelia priamo z firmy, kde sa registrácia vytvára.

Pri registrácii klientskej aplikácie je dôležité nastaviť v sekcii Authentication konfiguráciu platformy. V tomto prípade sa jedná o Single-page application. Zarovno s tým sa nastavuje redirect URI, ktorá špecifikuje kam môže Azure presmerovať užívateľa po úspešnom overení. Primárnym účelom tohto nastavenia je zabezpečiť, že sa pri prihlásení užívateľ vráti priamo na stránky fyzicky nakonfigurované vlastníkom registrácie takže sa predpokladá, že to bude bezpečné. Ďalej treba nastaviť API permissions, kde je potrebné špecifikovať API serverovej časti aplikácie. Týmto spôsobom sa zabezpečí, že komunikácia, ktorá s týmto API bude prebiehať je odobrená samotným správcom registrácie. Azure poskytuje aj dôkladnejšie nastavovanie rôznych parametrov, aj takých čo nepodporuje samotné užívateľské rozhranie Azure, v sekcii manifest. Tu je možné nájsť súhrnné informácie vo formáte JSON o tom ako je daná registrácia nakonfigurovaná.

Nasleduje registrácia serveru. Vzhľadom k tomu, že celá aplikácia využíva pre komunikáciu so službami od firmy Microsoft prevažne app-only access je potrebné vytvoriť takzvaný secret. Jedná sa o heslo, ktoré správca registrácie nastaví a od toho bodu sa s ním môže server preukázať, že sa naozaj jedná o aplikáciu, ktorá registráciu využíva oprávnene. Je to náhodne vygenerovaná sekvencia znakov, ktorú je možné prečítať len raz pri samotnej inicializácii. V momente ako správca opustí túto čast Azure portálu už nikdy nebude možné ju vyčítať. Ďalej je nutné nastaviť špecifické API permissions, kde je možné špecifikovať ku akým službám má server dovolené pristupovať. V prípade tejto práce sa jedná o Graph API služby a to pre čítanie zdieľanej schránky a základných údajov o užívateľoch. V sekcii Expose an API sa špecifikuje API aj je rozsah, ktorú bude server poskytovať. Toto je ten bod, ktorý sa v predošlej časti textu prepájal s registráciou klienta. V podstate hovorí o tom, že je povolená komunikácia medzi týmito dvoma registráciami.

7.3 Nasadenie aplikácie

Populárne vývojové prostredie pre .NET projekty od firmy Microsoft Visual Studio má integrovanú podporu pre nasadenie aplikácie priamo na Azure. Stačí kliknúť na projekt, ktorý má byť nasadený, vybrať Azure a autentifikovať užívateľa. Projekt sa následne vybuduje a nahrá do služby v Microsoft Azure.

Tento prístup však nevyhovuje potrebám tejto práce. Keďže sa neskladá z jedného projektu, ale viacerých, ktoré je potrebné spolu vybudovať a odoslať na Azure. Visual Studio nemá podporu pre nasadenie celého solution. Zároveň tento proces ignoruje existenciu externej databázy ako PostgreSQL, ktorá paralelne s nasadením aplikácie potrebuje odoslať a vykonať všetky SQL skripty.

S Azure sa dá pracovať cez command-line interface priamo v konzole, alebo pomocou skriptov. Projekty sa postupne každý vybudujú a všetko sa odloží do priečinku. To ako projekt vybudovať sa špecifikuje v publish profile každého projektu. Takto vystavaný solution sa následne môže nasadiť do Web API na Azure pomocou command-line interface. Paralelne s tým sa vyskladá aj migračný skript pre databázu a pomocou už SSH tunela do virtuálnej siete sa nasadí v databáze.

Výhoda tohto prístupu je, že sa môže využiť aj napríklad v internej sieti nejakej firmy. Za bežných okolností by tento proces vyžadoval odoslať kód zodpovednej osobe, vysvetliť jej ako nainštalovať Visual Studio a následne asistovať pri procese nasadenia. Takto miesto toho je možné odoslať celý archív projektu s prednastaveným skriptom, ktorý sa len spustí bez nejakých komplikácií, ktorým by nemusel daný zamestnanec firmy rozumieť.

Kapitola 8

Možné vylepšenia

Úlohou tejto kapitoly je popísať vylepšenia, ktoré v budúcnosti môžu prispieť ku efektivite výberového procesu, automatizovať niektoré operácie a umožniť lepšiu konfiguráciu systému.

Automatizácia synchronizácie s Microsoft Cloud - Synchronizácia sa spúšťa kliknutím tlačítka v užívateľskom rozhraní emailového klienta. Vzhľadom k tomu, že pre prístup ku dátam v Graph API sa využíva app-only access môže táto operácia byť vykonávaná bez ohľadu na to či užívateľ je prihlásený, alebo nie. Vďaka tomu sa môže nahradiť tlačítko za službu, ktorá na pozadí bude emaily aktualizovať v určenom časovom intervale.

Komentáre - Keďže na žiadosti spolupracuje viacero zamestnancov je dôležitá komunikácia medzi nimi. Hlavnou úlohou tejto práce je podporovať automatizáciu procesu a redukovať komunikačný šum medzi členmi tímu. Aby sa predišlo zbytočnej komunikácii mimo aplikácie je možné implementovať nadstavbu nad entitami žiadosti, ktoré by umožnili pridávať textové komentáre na rôzne jej súčasti. Týmto spôsobom by mohla prebehnúť diskusia, ktorá by bola centralizovaná so súvisiacou žiadosťou a eventuálne aj archivovaná.

Administrácia užívateľov - Na procese vybavovania žiadosti sa podieľa vždy viacero zamestnancov. V aktuálnej implementácii nie je k dispozícii administrácia pre pridávanie, alebo odoberanie jednotlivých užívateľov zo systému prípadne priradenie práv a zodpovedností. Graph API vystavuje rozhranie pre správu užívateľov. Je možné implementovať rozhranie, ktoré pomocou zobrazí celý zoznam užívateľov, umožní pridanie, alebo odobratie užívateľa zo systému a napokon pridelenie zodpovednosti.

Upozornenia - Počas vybavovania žiadosti sa vykonávajú rôzne aktivity. Zodpovednosť za vykonanie novovzniknutej aktivity nemusí vždy spočívať na zamestnancovi, ktorý ju vytvoril. Aby sa zabezpečil hladký priebeh spracovania žiadosti, systém by mohol automaticky upozorniť zodpovedného užívateľa o potrebe jeho intervencie. Týmto spôsobom by sa predišlo zbytočnej komunikácii medzi zamestnancami a efektívne by sa eliminoval komunikačný šum.

Archivácia - Každá ukončená žiadosť by mala byť archivovaná spolu s emailovou komunikáciou. To umožní vytvárať súhrnné správy, ktoré pomôžu firme sa lepšie orientovať v procese prípadne sledovať, kde by bolo možné proces vylepšiť. Ak by sa rovnaký kandidát v budúcnosti uchádzal o novú pozíciu je možné, na základe archivovaného výberového konania z minulosti, to aktuálne upraviť a preskočiť tak nepotrebné kroky.

Záver

Cieľom práce bolo vyvinúť softvér, ktorý adresuje problémy spojené s aktuálnou implementáciou výberového konania vo firme Mönkemöller IT GmbH.

Na začiatku boli tieto problémy zjednodušené a preformulované do požiadaviek. Všeobecné požiadavky usmerňovali proces obecného návrhu pre výber technológií a architektúry. Zvyšok implementácie bol rozdelený do troch modulov, ktoré jednotlivo riešili funkčné požiadavky kladené na systém.

Prvý krok ku zvýšeniu efektivity výberového konania bolo zjednotenie potrebných nástrojov pre vykonanie žiadosti. Prevažne sa jednalo o emailový klient, ktorý bol integrovaný do systému aby poskytoval užívateľom dodatočné rozhranie pre vytváranie žiadostí a manipuláciu s nimi. Emailový klient synchronizuje nakonfigurované priečinky do systému a kompletne nahradzuje potrebu použitia inej externej aplikácie na správu emailov.

Systém ďalej poskytuje rozhranie pre úpravu vytvorenej žiadosti a udržiavanie jej aktuálneho stavu. Každá úprava je sprevádzaná množstvom validácií, ktoré zabezpečujú, že všetky informácie v systéme sú úplné a organizované. Jednotlivé fázy žiadosti sú vždy pred ukončením dokumentované pre komunikáciu stavu zvyšným členom tímu a prípadnú archiváciu v budúcnosti. Úlohou žiadosti nie je len zbierať a centralizovať informácie, ale aj organizovať potrebné aktivity. Zamestnanci môžu počas každej fázy stanoviť aké aktivity je potrebné vykonať, naplánovať ich a prideliť zodpovedným osobám. Tým zaniká potreba využívania externých aplikácií pre plánovanie udalostí či komunikácie so zvyškom tímu o stave žiadosti.

V zmysle automatizácie a plynulého behu aplikácie bolo implementované užívateľské rozhranie pre administráciu systému. Umožňuje zamestnancom dopredu nakonfigurovať vstupné kanály, otvorené pozície a synchronizačné priečinky. Vďaka tomu sa aj v prípade súbežného využívania systému zabráni zbytočným duplikátom a dosiahne vyššia efektivita.

Vývoj práce prebehol úspešne a pôvodný zámer bol plnohodnotne dosiahnutý. Proces vybavovania žiadosti už viac nevyžaduje použitie externých aplikácií, všetky informácie sú uložené v štandardizovanom formáte a doba potrebná na vybavenie jednej žiadosti sa výrazne skrátila.

Bibliografia

1. *What is a progressive web app?* [Online]. [cit. 2024-04-18]. Dostupné z : https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/What_is_a_progressive_web_app.
2. *5 Types of Programming Languages* [online]. [cit. 2024-04-19]. Dostupné z : https://www.coursera.org/articles/types-programming-language.
3. *What is Procedural Language?* [Online]. [cit. 2024-04-19]. Dostupné z : https://www.geeksforgeeks.org/what-is-procedural-language/.
4. *Functional programming* [online]. [cit. 2024-04-19]. Dostupné z : https://en.wikipedia.org/wiki/Functional_programming.
5. *What Is Object-Oriented Programming?* [Online]. [cit. 2024-04-19]. Dostupné z : https://www.codecademy.com/resources/blog/object-oriented-programming.
6. *Java (programming language)* [online]. [cit. 2024-04-20]. Dostupné z : https://en.wikipedia.org/wiki/Java_%28programming_language%29.
7. *Python (programming language)* [online]. [cit. 2024-04-20]. Dostupné z : https://en.wikipedia.org/wiki/Python_(programming_language).
8. *A tour of the C# language* [online]. [cit. 2024-04-19]. Dostupné z : https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp.
9. *JavaScript* [online]. [cit. 2024-04-19]. Dostupné z : https://en.wikipedia.org/wiki/JavaScript.
10. *Vaadin* [online]. [cit. 2024-04-20]. Dostupné z : https://en.wikipedia.org/wiki/Vaadin.
11. *Reflex: A Library to Build Performant and Customizable Web Apps in Pure Python* [online]. [cit. 2024-04-20]. Dostupné z : https://medium.com/@HeCanThink/reflex-a-library-to-build-performant-and-customizable-web-apps-in-pure-python-2bfde0344af2.
12. *ASP.NET Core Blazor* [online]. [cit. 2024-04-20]. Dostupné z : https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0.
13. *What is Angular?* [Online]. [cit. 2024-04-20]. Dostupné z : https://angular.io/guide/what-is-angular.

14. *React Introduction* [online]. [cit. 2024-04-20]. Dostupné z : https://www.geeksforgeeks.org/reactjs-introduction.
15. *What is Java Spring Boot?* [Online]. [cit. 2024-04-21]. Dostupné z : https://www.ibm.com/topics/java-spring-boot.
16. *Introduction to Spring Framework* [online]. [cit. 2024-04-21]. Dostupné z : https://www.geeksforgeeks.org/introduction-to-spring-framework.
17. *Django introduction* [online]. [cit. 2024-04-21]. Dostupné z : https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction.
18. *Overview of ASP.NET Core* [online]. [cit. 2024-04-21]. Dostupné z : https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0.
19. *How Node.js Powers the Many User Interfaces of Netflix* [online]. [cit. 2024-04-21]. Dostupné z : https://thenewstack.io/netflix-uses-node-js-power-user-interface.
20. *What is Node.js? A beginner's introduction to JavaScript runtime* [online]. [cit. 2024-04-21]. Dostupné z : https://www.educative.io/blog/what-is-nodejs.
21. *System Properties Comparison Microsoft SQL Server vs. Oracle vs. PostgreSQL vs. SQLite* [online]. [cit. 2024-04-22]. Dostupné z : https://db-engines.com/en/system/Microsoft+SQL+Server%3BOracle%3BPostgreSQL%3BSQLite.
22. *What is a Document Database?* [Online]. [cit. 2024-04-22]. Dostupné z : https://www.mongodb.com/document-databases.
23. *What is a relational database?* [Online]. [cit. 2024-04-22]. Dostupné z : https://www.ibm.com/topics/relational-databases.
24. *What is Graph Database – Introduction* [online]. [cit. 2024-04-22]. Dostupné z : https://www.geeksforgeeks.org/what-is-graph-database.
25. *GRASP Design Principles in OOAD* [online]. [cit. 2024-04-24]. Dostupné z : https://www.geeksforgeeks.org/grasp-design-principles-in-ooad/.
26. *Model–view–controller* [online]. [cit. 2024-04-24]. Dostupné z : https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller.
27. *Model–view–presenter* [online]. [cit. 2024-04-24]. Dostupné z : https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter.
28. *Model-View-ViewModel* [online]. [cit. 2024-04-24]. Dostupné z : https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm.
29. *HTTP* [online]. [cit. 2024-04-24]. Dostupné z : https://en.wikipedia.org/wiki/HTTP.
30. *REST* [online]. [cit. 2024-04-24]. Dostupné z : https://en.wikipedia.org/wiki/REST.
31. *SOAP* [online]. [cit. 2024-04-24]. Dostupné z : https://en.wikipedia.org/wiki/SOAP.
32. *Use the Microsoft Graph API* [online]. [cit. 2024-04-26]. Dostupné z : https://learn.microsoft.com/en-us/graph/use-the-api.

33. *Use delta query to track changes in Microsoft Graph data* [online]. [cit. 2024-04-26]. Dostupné z : `https://learn.microsoft.com/en-us/graph/delta-query-overview`.

34. *Delegated access (access on behalf of a user)* [online]. [cit. 2024-04-29]. Dostupné z : `https://learn.microsoft.com/en-us/graph/auth/auth-concepts#delegated-access-access-on-behalf-of-a-user`.

35. *App-only access (access without a user)* [online]. [cit. 2024-04-29]. Dostupné z : `https://learn.microsoft.com/en-us/graph/auth/auth-concepts#app-only-access-access-without-a-user`.

Obsah príloh

```
│ readme.txt ........................................ stručný popis obsahu média
├── src
│   ├── MMIT.AMS.Client ........................ zdrojové kódy implementácie klienta
│   ├── MMIT.AMS.Client.Logic ........... zdrojové kódy implementácie logiky klienta
│   ├── MMIT.AMS.Server ......................... zdrojové kódy implementácie serveru
│   ├── MMIT.AMS.Server.Logic ........... zdrojové kódy implementácie logiky serveru
│   ├── MMIT.AMS.Shared ............ zdrojové kódy implementácie zdieľaného projektu
│   ├── SQL Scripts ........................................... databázové skripty
│   └── Database Model.dbm ..................................... databázový model
├── thesis ................................. zdrojová forma práce vo formáte LaTeX
└── text .......................................................... text práce
    └── thesis.pdf ..................................... text práce vo formáte PDF
```