Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Webová aplikace pro správu databáze chemických vzorků

Bc. Jan Kočvara

© 2023 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Jan Kočvara

Informatika

Název práce

Webová aplikace pro správu databáze chemických vzorků

Název anglicky

A web application for managing a database of chemical samples

---

Cíle práce

Cílem je refaktorovat, optimalizovat bakalářskou práci na stejné téma a rozšířit ji o nové funkcionality. Původní webová aplikace (MVC) bude rozdělena na dvě části, webovou REST API obsahující business logiku aplikace a klientskou webovou aplikaci. API bude napsána v jazyce C# s technologií .NET Core 6. Pro práci s databází bude použit Entity Framework Core.

Klientská webová aplikace bude napsána na platformě NodeJS za použití knihovny React. Klientská aplikace bude obsahovat dvě úrovně zobrazení. Uživatelské zobrazení, umožňující přidávání nových záznamů, nahrávání souborů s měřením chemických vzorků ve formátu CSV a následné zobrazení a filtrování přidaných záznamů. Uživatelé budou moci být součástí skupin, ve kterých se budou záznamy automaticky sdílet. Druhým pohledem bude jednoduchý pohled administrace, ve kterém bude možné spravovat všechny přidané záznamy a registrované uživatele v systému.

Metodika

Analyzujte požadavky na databázi vzorků.

Proveďte rešerši s cílem nalézt podobná řešení i ve vztahu k původní aplikaci.

Na základě rešerše sestavte novou aplikaci dle následujících metodických pokynů:

Webové API bude navrženo a napsáno v platformě .NET Core 6 za pomocí jazyka C#. Pro navržení webové API bude využita paradigma OOP. V návrhu a struktuře projektu bude také využit koncept DDD (Domain-Driven Design), za účelem implementovat funkcionalitu API co nejabstraktněji. Pro práci s databází bude použita ORM knihovna Entity Framework Core.

Klientská webová aplikace bude napsána v kombinaci jazyků HTML, CSS, JavaScript a TypeScript na platformě NodeJS za použití knihovny React. Pro správu balíčků třetích stran bude použit software NPM. Aplikace bude navržena a realizována pomocí funkcionálního přístupu, tedy bude se skládat ze znovupoužitelných komponent, které budou následně tvořit jednotlivé stránky

---

Doporučený rozsah práce

50-60

Klíčová slova

.NET Core 6, NodeJS, React, JavaScript, TypeScript, C#, API, EF Core, ORM

Doporučené zdroje informací

https://docs.microsoft.com/cs-cz/learn/modules/build-web-api-aspnet-core/?WT.mc_id=dotnet-35129-website
https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio
https://docs.npmjs.com/getting-started
https://dotnet.microsoft.com/en-us/apps/aspnet/apis
https://reactjs.org/docs/getting-started.html
https://www.typescriptlang.org/docs/

Předběžný termín obhajoby

2022/23 LS – PEF

Vedoucí práce

Ing. Josef Pavlíček, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 4. 11. 2022

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 31. 03. 2023

Čestné prohlášení

Prohlašuji, že svou diplomovou práci Webová aplikace pro správu databáze chemických vzorků jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2023 ______________________________

Poděkování

Rád bych touto cestou poděkoval vedoucímu práce Ing. Josefu Pavlíčkovi Ph.D., za veškeré rady, ochotu, čas, asistenci a především trpělivost, kterou měl s konzultacemi práce. Dále bych rád poděkoval kolegům, kteří byli součástí testování výsledné verze aplikace. V neposlední řadě bych rád také rád poděkoval Bc. Martinu Tomsovi za veškerou podporu v průběhu psaní této práce a mé rodině za veškerou podporu kterou mi při realizování práce dali.

Webová aplikace pro správu databáze chemických vzorků

Abstrakt

Hlavním cílem práce je navrhnout a vymodelovat relační databázi určenou pro ukládání libovolných tabulkových souborů ve formátu CSV, včetně výsledků chemické analýzy vzorků. Pro umožnění potencionálního rozšíření řešení je použitý model klient-server, kdy je klientská aplikace spuštěna ve webovém prostředí, pomocí moderních webových prohlížečů. Klientská aplikace komunikuje se serverovou aplikací, která pracuje s databázovým systémem.

V teoretické části práce jsou vysvětleny technologie použité při realizaci návrhu řešení. Mezi tyto technologie jsou zařazeny knihovny a frameworky použité při implementaci řešení, programovací jazyk použitý k vytvoření klientské aplikace a teorie ohledně použitého modelu klient-server.

V praktické části je popsána struktura navržené relační databáze a implementace jak serverové aplikace, tak klientské aplikace. Součástí popisu implementace je i popis použitých návrhových vzorů.

Klíčová slova: .NET Core 6, NodeJS, React, JavaScript, TypeScript, C#, API, EF Core, ORM

A Web Application for Managing a Database of Chemical Samples

Abstract

The main goal of this thesis is to design and model a relational database designed to store any tabular files in the CSV format, including the results of the chemical analysis of samples. To enable potential solution expansion, a client-server model is used, where the client application is launched in a web environment using a modern web browser. The client application communicates with the server application that works with the database system.

The theoretical part of the work explains the technologies used in the implementation of the solution. These technologies include the libraries and frameworks used in the solution implementation, programming languages used to create the client application, and the theory regarding the client-server model used.

The practical part describes the structure of the designed relational database and the implementation of both the server and the client applications. The description of implementation also includes a description of the design patterns used.

Keywords: .NET Core 6, NodeJS, React, JavaScript, TypeScript, C#, API, EF Core, ORM

Obsah

1 Úvod **13**

2 Cíl práce a metodika **14**

2.1 Cíl práce 14

2.2 Metodika 14

3 Teoretická východiska **15**

3.1 Model Client – Server 15

3.1.1 Historie modelu 15

3.1.2 Výhody modelu 16

3.1.3 Nevýhody modelu 17

3.2 .Net Core 17

3.2.1 Historie 18

3.2.2 Xamarin 19

3.2.3 Blazor 19

3.3 Knihovna React 20

3.3.1 Historie Reactu 20

3.3.2 Klíčové vlastnosti 21

3.3.2.1 Komponenty 21

3.3.2.2 Virtuální DOM 21

3.3.2.3 JSX 22

3.3.3 Populární knihovny 22

3.3.3.1 React Router 23

3.3.3.2 React Redux 23

3.4 TypeScript 24

3.4.1 Historie 25

3.4.2 Statické typování 25

3.4.3 Podpora pro OOP 26

4 Vlastní práce **27**

4.1 Požadovaná funkcionalita 27

4.2 Použité technologie a alternativy 27

4.2.1 Serverová aplikace 28

4.2.1.1 Alternativy k .NET 28

4.2.2 Klientská aplikace 28

4.2.2.1 Alternativy k ReactJS 29

4.2.3 Databázový systém 29
4.2.4 Hosting webového serveru 29
4.3 Návrh databázového řešení 30
4.3.1 Tabulka Person 31
4.3.2 Tabulka Login 31
4.3.3 Tabulka Role 32
4.3.4 Tabulka PersonRole 33
4.3.5 Tabulka Contact 33
4.3.6 Tabulka ContactType 34
4.3.7 Tabulka Record 34
4.3.8 Tabulka RecordData 35
4.3.9 Tabulka RecordGroup 36
4.3.10 Tabulka PersonGroup 36
4.3.11 Tabulka PersonGroupRelations 37
4.3.12 Tabulka PersonGroupRecord 38
4.3.13 Tabulka PersonGroupRecordGroup 39
4.4 REST API 39
4.4.1 Návrh architektury serverového řešení 40
4.4.1.1 Repozitáře 40
4.4.1.2 Řadiče 42
4.4.2 Modely databázových entit 45
4.4.3 Data Transfer Object (DTO) 47
4.4.4 Dotazovací modely 47
4.4.5 Entity Framework Core 50
4.4.5.1 Databázové kontexty 51
4.4.5.2 Limitovaný kontext 52
4.4.6 Autentifikace a autorizace 54
4.4.6.1 JSON Web Token 54
4.4.6.2 Zvolení databázového kontextu 57
4.4.7 Nahrávání a zpracování souborů 58
4.5 Klientská aplikace 60
4.5.1 Výhody funkcionálního přístupu 60
4.5.2 Kontext v knihovně React 60
4.5.3 Hooky 61
4.5.3.1 Komunikace s API 62
4.5.3.2 Manažer barevného schéma 63
4.5.3.3 AuthHook 64

4.5.4 Komponenty ... 65
4.5.4.1 MUI ... 66
4.5.4.2 Tailwind CSS ... 67
4.5.4.3 SCSS ... 67
4.5.5 Směrování ... 67
4.5.5.1 Zabezpečení cesty ... 69
4.5.6 Grafické zpracování ... 70
4.5.6.1 Desktopové zobrazení ... 71
4.5.7 Administrace ... 72
4.5.7.1 Seznam přihlašovacích účtů ... 72
4.5.7.2 Detail uživatelského účtu ... 74
4.5.7.3 Tvorba uživatele ... 75
4.5.7.4 Seznam nahraných souborů ... 75
4.5.7.5 Seznam týmů ... 77
4.5.8 Uživatelské rozhraní ... 78
4.5.8.1 Souborový prostor ... 78
4.5.8.2 Nahrání souboru ... 78
4.5.8.3 Hluboké prohledávání CSV souborů ... 79
4.5.8.4 Sdílené soubory ... 79
4.5.8.5 Seznam týmů ... 80
4.5.8.6 Vytvoření nového týmu ... 80
4.5.8.7 Seznam aktivních pozvánek ... 81
5 Výsledky a diskuse ... 82
5.1 API ... 82
5.1.1 Testování ... 82
5.2 Klientská aplikace ... 83
5.2.1 Testování ... 84
6 Závěr ... 86
7 Seznam použitých zdrojů ... 88
8 Seznam obrázků, tabulek, grafů a zkratek ... 94
8.1 Seznam obrázků ... 94
8.2 Seznam ukázek zdrojového kódu ... 94
8.3 Seznam tabulek ... 95

8.4 Seznam použitých zkratek 96

Přílohy 98

1 Úvod

Jedním z unikátních nosičů tabulkových dat, které je možné přenášet mezi operačními systémy a různými platformami jsou CSV soubory. Tyto soubory však mohou být velmi rozsáhlé a obtížně spravovatelné. Proto se tato diplomová práci zaměřuje na návrh a implementaci klient-server řešení pro správu CSV souborů ve webovém prostředí. Řešení bude umožňovat uživatelům jednoduchý a přehledný způsob nahrávání, odebírání, prohlížení a filtrování nahraných souborů, a to i v případě velkého množství dat. Navrhované řešení bude brát v potaz případnou možnost rozšíření na další platformy, než je webová platforma.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem je refaktorovat, optimalizovat bakalářskou práci na stejné téma a rozšířit ji o nové funkcionality. Původní webová aplikace (MVC) bude rozdělena na dvě části, webovou REST API obsahující business logiku aplikace a klientskou webovou aplikaci. API bude napsána v jazyce C# s technologií .NET Core 6. Pro práci s databází bude použit Entity Framework Core.

Klientská webová aplikace bude napsána na platformě NodeJS za použití knihovny React. Klientská aplikace bude obsahovat dvě úrovně zobrazení. Uživatelské zobrazení, umožňující přidávání nových záznamů, nahrávání souborů s měřením chemických vzorků ve formátu CSV a následné zobrazení a filtrování přidaných záznamů. Uživatelé budou moci být součástí skupin, ve kterých se budou záznamy automaticky sdílet. Druhým pohledem bude jednoduchý pohled administrace, ve kterém bude možné spravovat všechny přidané záznamy a registrované uživatele v systému.

2.2 Metodika

Analyzujte požadavky na databázi vzorků. Proveďte rešerši s cílem nalézt podobná řešení i ve vztahu k původní aplikaci. Na základě rešerše sestavte novou aplikaci dle následujících metodických pokynů:

Webové API bude navrženo a napsáno v platformě .NET Core 6 za pomocí jazyka C#. Pro navržení webové API bude využito paradigma OOP. V návrhu a struktuře projektu bude také využit koncept DDD (Domain-Driven Design), za účelem implementovat funkcionalitu API co nejabstraktněji. Pro práci s databází bude použita ORM knihovna Entity Framework Core.

Klientská webová aplikace bude napsána v kombinaci jazyků HTML, CSS, JavaScript a TypeScript na platformě NodeJS za použití knihovny React. Pro správu balíčků třetích stran bude použit software NPM. Aplikace bude navržena a realizována pomocí funkcionálního přístupu, tedy bude se skládat ze znovupoužitelných komponent, které budou následně tvořit jednotlivé stránky.

3 Teoretická východiska

3.1 Model Client – Server

Tento model rozděluje webovou aplikaci na dvě hlavní části – klienta a server. Klient je typicky webový prohlížeč (jako např. Google Chrome nebo Mozilla Firefox), který se připojuje k serveru prostřednictvím sítě. Klient požádá server o informace, jako jsou HTML stránky, obrázky, videa nebo data z databáze, které chce zobrazit na svém zařízení.[1, 2]

Server je počítač, který má webovou aplikaci a data, která klient požaduje. Server přijme požadavek od klienta, zpracuje ho a odešle odpověď s požadovanými informacemi zpět klientovi.[1, 2]

Webové aplikace mohou používat různé jazyky pro implementaci serverové části, jako je PHP, Python, Ruby, Java nebo JavaScript (Node.js). Tyto jazyky běží na serveru a vytvářejí dynamické webové stránky, které jsou založené na konkrétních požadavcích klienta.[3, 4]

3.1.1 Historie modelu

Architektura Client-server má poměrně dlouhou historii a vznikla v 60. letech 20. století jako reakce na potřebu většího sdílení dat a větší efektivity v oblasti počítačových sítí. V té době se počítače používaly jako samostatné a izolované jednotky, které byly používány pro specifické úkoly a nebyly propojené. To však vedlo k velkému plýtvání zdrojů, protože většinou byly využívány pouze v malé části své kapacity.[5–8]

Client-server architektura byla vytvořena jako řešení tohoto problému tím, že umožnila počítačům sdílet zdroje a umožnila lepší využití jejich kapacity. V té době byly tyto zdroje většinou v podobě velkých mainframe počítačů, které běžely na serverech. [3, 6, 8]

V 70. letech se objevily první sítě, které umožnily propojit tyto mainframe počítače, a to umožnilo vznik nových aplikací, které využívaly výhod architektury Client-server. Tyto aplikace využívaly možnosti sdílení zdrojů a data byla ukládána na serverech a klienti se připojovali k těmto serverům, aby získali přístup k datům.[6, 8]

V 80. letech se rozšířilo používání PC a tato architektura byla přizpůsobena pro použití na těchto osobních počítačích. Architektura Client-server se stala základem vývoje softwaru v té době a tato architektura byla využívána v mnoha oblastech, jako jsou bankovnictví, zdravotnictví a výroba.[6, 8]

V 90. letech se rozšířilo používání internetu a architektura Client-server se stala základem pro vývoj webových aplikací. To umožnilo klientům přistupovat k informacím a službám z různých serverů po celém světě a umožnilo vznik globální sítě.[6, 8]

V současné době architektura Client-server stále hraje důležitou roli v mnoha oblastech a vývojáři stále pracují na jejím zlepšení a rozšíření, aby se mohla přizpůsobit novým výzvám v oblasti informačních technologií.[6, 8]

3.1.2 Výhody modelu

Jednou z nejvýznamnějších výhod architektury Client-server je centralizované řízení a správa dat. Data jsou uložena na centrálním serveru, což umožňuje lepší řízení a správu dat. Správa dat může být prováděna centrálně a úpravy dat mohou být provedeny na jednom místě, což zjednodušuje správu a údržbu.[9–11]

Další výhodou architektury Client-server je efektivita využití zdrojů. Využití zdrojů může být efektivnější, protože výpočetní výkon a úložiště jsou sdíleny mezi klienty, což umožňuje lepší využití kapacity. Klienti mohou využívat zdroje, aniž by si museli sami pořizovat drahé hardware a software.[9–11]

Architektura Client-server také poskytuje větší bezpečnost. Data a aplikace mohou být ukládány na centrálním serveru, což umožňuje lepší zabezpečení dat. Může být snazší ovládat přístup k datům a aplikacím a lépe je chránit před ztrátou nebo poškozením.

Další výhodou architektury Client-server je lepší škálovatelnost. Architektura umožňuje snadné škálování, protože server může být aktualizován, aby zvládal větší zatížení. To umožňuje firmám snadno rozšiřovat své počítačové systémy, jakmile se zvýší požadavky na výkon.[9–11]

Vysoká dostupnost je další výhodou architektury Client-server. Centrální server může být navržen tak, aby poskytoval vysokou dostupnost služeb, což znamená, že služby jsou k dispozici pro klienty po většinu času, bez ohledu na případné problémy s hardwarem nebo sítí.[9–11]

Posledním faktorem, který způsobuje, že je architektura Client-server výhodná, je snadná aktualizace. Aktualizace aplikace nebo dat mohou být prováděny na centrálním serveru a poté jsou k dispozici pro všechny klienty. To umožňuje snadnou aktualizaci aplikací.[9–11]

3.1.3 Nevýhody modelu

Client-server architektura má ve webovém prostředí několik nevýhod. Jedním z problémů je zpoždění, které vzniká, když klient čeká na odpověď od serveru. Pokud je server přetížen, může to vést k prodloužení doby odezvy a ke zhoršení uživatelského zážitku. Dále, klienti jsou závislí na serveru, aby poskytovali data a zpracovávali jejich požadavky. Pokud dojde k výpadku serveru nebo k problému s připojením, klienti nemohou plně využít webové aplikace.[9, 10, 12]

Dalším problémem je řízení dat, protože server je zodpovědný za správu a řízení dat a klienti nemají přímý přístup k těmto datům. To může vést k problémům s ochranou dat a zabezpečením. Kromě toho, klient-server architektura vyžaduje více výpočetních prostředků a síťového připojení pro zpracování požadavků na serveru a poskytování dat klientům, což může vést k vysokým nákladům na provoz a k omezení výkonu. [9, 10, 12]

Další značnou nevýhodou klient-server architektury je prodloužení vývoje, protože vyžaduje programování jak serverové aplikace (API), tak i klientské aplikace. Vývojáři musí navrhnout a vytvořit komunikační protokoly, které umožní klientům komunikovat se serverem a zpracovávat data. To zvyšuje náročnost vývoje a může vést k většímu objemu práce a nákladům na vývoj a údržbu.[12, 13]

Při zvyšování počtu klientů a objemu dat se může objevit potřeba škálovat serverovou infrastrukturu, což může být nákladné a složité. [9, 10, 12]

3.2 .Net Core

.NET Core je open-source, multiplatformní framework pro vývoj moderních aplikací. Byl vytvořen společností Microsoft jako vylepšení původního .NET Frameworku a je dostupný pro vývojáře zdarma.[14, 15]

Jednou z hlavních výhod .NET Core je jeho multiplatformnost, díky čemuž je možné framework použít na vývoj aplikací pro různé operační systémy, jako jsou Windows, Linux

a macOS. To umožňuje vývojářům psát aplikace v programovacích jazycích C#, F# nebo Visual Basic a spouštět je na různých platformách bez nutnosti přepisovat kód.[14, 15]

.NET Core také nabízí širokou škálu knihoven a nástrojů, které usnadňují vývoj moderních aplikací, včetně ASP.NET Core pro vývoj webových aplikací a Entity Framework Core pro práci s databázemi. Díky těmto nástrojům lze aplikace v .NET Core vyvíjet rychleji a efektivněji.[14, 16, 17]

Další důležitou výhodou frameworku .NET Core je jeho modularita. Vývojáři mohou použít jen to, co potřebují. Díky tomu jsou výsledné binární soubory menši a dochází tak k efektivnějšímu využití zdrojů. [14, 16, 17]

3.2.1 Historie

Historie .NET Core začíná v roce 2014, kdy společnost Microsoft oznámila, že pracuje na novém open-source, multiplatformním frameworku pro vývoj aplikací. Tento framework měl být vylepšením původního .NET Frameworku a umožnit vývojářům psát aplikace, které by byly spustitelné na různých operačních systémech, včetně systémů Windows, Linux a macOS.[18, 19]

První verze .NET Core, která se jmenovala .NET Core 1.0, byla publikována v roce 2016. Byla to zcela nová platforma, která se skládala z různých modulů, do kterých patří například runtime, knihovny a různé nástroje určené pro vývoj.[19, 20]

Během následujících let byl .NET Core postupně vylepšován a rozšiřován o nové funkce a vylepšení, jako například podporu pro .NET Standard, což umožnilo jednodušší sdílení kódu mezi různými implementacemi .NET, včetně .NET Frameworku a Xamarinu.[19, 21, 22]

V roce 2019 byl .NET Core sloučen s Mono, což je open-source implementace .NET Frameworku, a vznikla tak nová platforma s názvem .NET 5. Tato verze nabízí mnoho nových vylepšení a funkcí, jako například podporu pro nové jazykové prvky v C# verze 9, vylepšenou podporu pro WPF (zkratka pro *Windows Platform Foundation*), Windows Forms aplikace a mnoho dalších.[19, 21, 23]

Dnes je nejnovější .NET Core ve verzi 7. Framework je stále aktivně vyvíjen a používán v mnoha projektech po celém světě. [16, 17, 23]

3.2.2 Xamarin

Xamarin je platforma pro vývoj mobilních aplikací, která umožňuje vývojářům psát aplikace pro iOS, Android a Windows Phone v jazyce C#. Tato platforma umožňuje vývojářům sdílet více než 75 % kódu mezi platformami, což znamená menší náklady na vývoj a kratší dobu uvedení na trh.[24–26]

Platforma obsahuje několik klíčových nástrojů, včetně *Xamarin.iOS* a *Xamarin.Android*. Zmíněné nástroje jsou knihovny pro vývoj nativních aplikací pro iOS a Android v jazyce C#. Knihovna *Xamarin.Forms* pak umožňuje psát kód pro vytvoření uživatelského rozhraní aplikace, které je možné použít na všech zmíněných operačních systémech.[24–26]

Aktuálně však popularita platformy Xamarin upadá, a to z důvodu uvedení nové platformy pojmenované .NET MAUI. Tato knihovna byla navržena jakožto nástupce platformy Xamarin. [27, 28]

3.2.3 Blazor

Blazor je moderní webová technologie vyvíjená společností Microsoft, která umožňuje vývoj webových aplikací pomocí C# a .NET frameworku. Jedním z hlavních prvků frameworku je možnost psát kód v C# a používat ho přímo ve webových aplikacích pomocí komponent. Blazor může být použit pro vytváření jednostránkových aplikací (z anglického *Single-Page Applications*) i pro tradiční webové stránky.[29–31]

Platforma používá webový standard WebAssembly k běhu C# kódu přímo v prohlížeči uživatele. To umožňuje vývojářům psát kód v jazyce C# a používat ho na straně klienta bez nutnosti psát JavaScriptový kód. Blazor také podporuje používání .NET Standard knihoven, což umožňuje vývojářům sdílet kód mezi různými platformami.[30, 32]

Platforma navíc poskytuje různé možnosti pro práci s uživatelským rozhraním, včetně možnosti použití hotových komponent, nebo tvorby vlastních komponent. Hotové komponenty je možné získat například pomocí knihoven třetích stran. Mezi takové knihovny patří Antd, Radzen Blazor, či Telerik UI. [29, 33–35]

3.3 Knihovna React

React je open-source knihovna pro tvorbu uživatelských rozhraní, zkráceně UI (z anglického *User Interface*) v jazycích JavaScriptu, či Typescript, která byla vyvinuta společností Facebook. Používá se především pro tvorbu webových stránek a aplikací, ale může být také použita pro vytváření mobilních aplikací. Pro vývoj mobilních aplikací se používá knihovna React Native. [36, 37]

React se soustředí na tvorbu jednostránkových aplikací, zkráceně SPA (z anglického *Single Page Application*). Tyto aplikace jsou tím, že jsou spuštěné pouze na jedné stránce. Tedy v aplikacích SPA se nepoužívá klasická navigace mezi částmi webové stránky, ale žádaný obsah se vykresluje v rámci jedné neustále načtené stránky. Hlavní výhodou Single Page Application je plynulost při interakci, právě protože je stránka načtena pouze jednou. Veškeré další změny způsobené interakcí se následně provádí pomocí Javascriptu bez nutnosti aktualizace či přesměrování stránky.[36, 37]

Hlavní nevýhodou aplikací SPA, kterou tedy trpí i React jsou problémy s optimalizací SEO (Search Engine Optimization). Jelikož se veškerý obsah obměňuje pomocí Javascriptu v rámci jedné stránky, může být pro roboty obtížné takové stránky procházet.[36, 38]

3.3.1 Historie Reactu

Knihovna React veřejně vznikla v roce 2013 ve společnosti Meta (dříve známá jako Facebook). Důvodem pro vznik knihovny byla neustále se zvedající komplexnost grafického uživatelského rozhraní webového portálu Facebook, a spolu s tím i stále náročnější správa zdrojového kódu.[39]

První vlastnost Reactu byla však vytvořena už roku 2010, kdy společnost Meta představila Xhp, které představilo možnost tvoření a skládání komponent. V roce 2011 vznikl první prototyp knihovny React, známý jako FaxJS. [39]

V roce 2012 byla společnost Instagram adoptována společností Meta. Společnost Instagram projevila zájem o použití interního FaxJS společnosti Meta ve své aplikaci. To ovšem nebylo možné, protože knihovna jako taková byla navržena a napsána specificky pro portál Facebook. Na základě této události byla následně knihovna upravena, aby byla schopna pracovat nezávisle a samostatně a v roce 2013 byl poprvé světu představena knihovna React licencována jako open-source.[39]

3.3.2 Klíčové vlastnosti

Jako každá knihovna, či nástroj v oblasti IT, i React má své klíčové vlastnosti. React umožňuje vyvíjet jak pomocí funkcionálního paradigmatu, tak objektového. Knihovna jako taková však využívá návrhový vzor známý jako „komponentní architektura". Tento vzor umožňuje tvořit grafické komponenty v podobě Javascriptových funkcí. [40]

3.3.2.1 Komponenty

Každá komponenta obsahuje minimálně jednu funkci, nebo třídu, která je následně exportována. Díky tomu je možné každou jednu komponentu znovupoužít prakticky kdekoliv v aplikaci. [41, 42]

Komponenta však nemusí být nutně jen tvořena pomocí funkcí, je zde možné i aplikovat OOP principy a tvořit komponenty z tříd. Avšak s aktuálně rostoucím trendem funkcionálního paradigma a stoupající náročnosti údržby kódu a komponent se čím dál více aplikují principy funkcionálního paradigma i do OOP, konkrétně principy jako „čisté funkce". Právě díky aplikováním těmto principům, je možné tvořit prakticky „atomické" komponenty, které řeší jen jeden specifický problém.[42, 43]

Hlavní výhodou komponent je jejich znovupoužitelnost. Proto je obvykle kladen obzvláště důraz na navržení komponent, aby byly co nejobecnější. V takové situaci je potom možné komponenty snadno použít kdekoliv je potřeba a případně jim předat parametry rozšiřující jejich funkcionalitu. Díky aplikování těchto zásad se značně zjednodušuje správa zdrojového kódu a zmenšuje se množství redundantního zdrojového kódu.[43]

3.3.2.2 Virtuální DOM

VDOM (z anglického *Virtual Document Object Model*) je abstraktní reprezentace DOM v paměti počítače, kterou React používá pro účely optimalizace a rychlosti při aktualizaci uživatelského rozhraní. VDOM je vysoce efektivní a umožňuje minimalizovat počet aktualizací, které jsou potřebné k dosažení požadovaného výsledku.[44]

Při vytváření dochází k procesu zvaném „*reconciliation*". Při tomto procesu, vytváří React virtuální DOM pro každou renderovanou komponentu. Když se změní stav nebo předávané parametry komponenty, také známé jako „*props*", React porovná VDOM komponenty před a po změně, aby zjistil, jaké změny jsou potřeba v reálném DOMu.

Tento celý proces je zodpovědný za aktualizaci pouze těch částí DOM, které jsou skutečně změněny.[44, 45]

Protože aplikování změn v DOMu je časově velmi náročná operace, je použití VDOM spolu s „*reconciliation*“ velmi efektivní. VDOM umožňuje nejen Reactu, ale i vývojáři pracovat s vysokou úrovní abstrakce nad reálným DOM. [44, 45]

3.3.2.3 JSX

Javascript XML, zkráceně JSX, je značkovací jazyk umožňující prokládat značkovací jazyk HTML s Javascriptem. V knihovně React se právě JSX používá jako abstrakce pro tvorbu HTML prvků. Protože je DOM v Reactu virtuální, využívá se interně funkce knihovny známá jako „*React.createElement()*“ . Tato funkce vloží reálný HTML prvek do DOMu v momentě potřeby. Protože by bylo velmi nepraktické, aby vývojář musel každý prvek takto vkládat manuálně, používá se v knihovně React právě JSX k definování obsahu DOMu. Knihovna následně tento Javascript XML kód interně překládá do již zmíněné funkce „*createElement()*.“[46–49]

JSX umožňuje i prokládat HTML kód s Javascript kódem. Toho je možné dosáhnout pomocí obalení kódu do množinových závorek. Pokud množinové závorky neobsahují funkci, je s kódem nakládáno jako s proměnnou určenou k vykreslení.[47]

JSX je kompilován do Javascriptu pomocí speciálních nástrojů, jako je například Babel. Pokud obdobné nástroje nejsou při kompilování k dispozici, kód by v prohlížeči neměl být funkční.[49]

3.3.3 Populární knihovny

Od počátku Reactu vznikla okolo knihovny opravdu velká komunita vývojářů, kteří vnesli do světa Reactu velké množství nápadů, rozšíření a knihoven. Jelikož React sám od sebe v základu nenabízí řešení pro časté problémy, jako je například routování mezi stránkami uvnitř aplikace, nebo předávání data a stavů mezi jednotlivými stránkami, či komponentami v rámci aplikace, vzniklo velké množství knihoven, které řeší právě zmíněné problémy a mnoho dalších.[50, 51]

3.3.3.1 React Router

Základní myšlenkou, za knihovnou React Router, je, že každá adresa URL v aplikaci odpovídá nějaké existující komponentě. Knihovna tak umožňuje tvořit webové aplikace, které mají více stránek. V kombinaci s Reactem pak dochází k velmi příjemnému uživatelskému zážitku, ve kterém se mění obsah webové stránky, ale stránka samotná se nemusí znovu načítat. Směrování je pak mnohem rychlejší, interaktivnější a uživatelsky přívětivější.[51]

Klíčovou vlastností knihovny React Router je schopnost spravování historie směrování v rámci aplikace. Díky této správě je možné přecházet mezi stránkami jak směrem zpět, tak následně i směrem vpřed. [51]

3.3.3.2 React Redux

Redux je knihovna, která umožňuje jednodušší správu stavu aplikace a sdílení stavů mezi komponentami. Redux je samostatná knihovna, kterou je možné použít i s jinými knihovnami či frameworky, než je React. Pro kombinaci s Reactem se však používá oficiální varianta knihovny zvaná React Redux.[52]

Knihovna obsahuje pět základních komponent:

- Store
- Reducer
- Action
- Dispatch
- Connect

Komponenta „Store" je objekt, který udržuje stav aplikace. Komponenta zároveň poskytuje metody, které umožňují s udržovaným stavem manipulovat. Store by měl vždy obsahovat veškerá data, ostatní komponenty se musí následně se Storem synchronizovat. [53]

Reducer je funkce, která manipuluje s uloženým stavem aplikace ve storu. Reducer je čistá funkce, tedy funkce, která neaplikuje žádný vedlejší efekt. Vedlejším efektem v tomto kontextu můžeme chápat například jako dotazování vzdáleného serveru, nebo překreslování aplikace. Reducer lze tedy označit za funkci transformující starý stav aplikace na nový stav aplikace. [53]

Action, neboli akce, jsou objekty, které přenášejí informaci, co se má stát v aplikaci. Každá akce musí být unikátního typu. Akce může také obsahovat aditivní data, známé jako „payload“. Pomocí akce lze tedy například předat nová data Reduceru, který následně uloží nová data do Storu, čímž transformuje stav aplikace.[53]

Dispatchem se rozumí funkce, která je používaná k vyvolání akce v aplikaci. Funkce Dispatch je volána v komponentách aplikace v momentě, kdy je třeba změnit stav aplikace. Skrze Dispatch se předává akce s novými daty do Reduceru.[53]

Poslední základní komponentou knihovny Redux je funkce Connect. Tato funkce umožňuje získat data ze storu.[53]

3.4 TypeScript

TypeScript je staticky typovaný jazyk programování, který vznikl jako nadstavba nad jazykem JavaScript. Byl vytvořen v roce 2012 společností Microsoft a od té doby jeho popularita mezi stále roste vývojáři.[54, 55]

Jednou z hlavních výhod TypeScriptu je jeho statické typování, což znamená, že při psaní kódu se musí specifikovat typ proměnných, funkcí a parametrů. Toto umožňuje vývojářům odhalit chyby v kódu během vývoje a zároveň usnadňuje porozumění kódu ostatním vývojářům.[56, 57]

Další výhodou TypeScriptu je jeho silné typování, což znamená, že jazyk dokáže kontrolovat typy nejen při běhu programu, ale také během kompilace. To vede k menšímu množství chyb za běhu programu a k lepšímu pochopení kódu.[56, 57]

TypeScript také nabízí nové funkce, které nejsou k dispozici v JavaScriptu, jako například deklarace typů, rozhraní, enumy, typové aliasy a další. Tyto funkce zlepšují čitelnost a údržbu kódu a zároveň umožňují vývojářům psát robustnější kód.[56, 58]

Důvody pro použití TypeScriptu jsou různé. Pro vývojáře, kteří pracují na velkých projektech, může být výhodné mít statické typování, aby se minimalizovaly chyby v kódu a usnadnilo se jeho čtení a úprava. TypeScript také může být užitečný pro vývojáře, kteří se učí programovat, protože statické typování pomáhá lépe porozumět tomu, jak programovací jazyk funguje. Silnou inspirací pro TypeScript je jazyk C#, který je taktéž vytvořen společností Microsoft.[54, 59, 60]

3.4.1 Historie

Samotná historie TypeScriptu sahá do roku 2010, když se začal tvořit projekt s názvem "Strada" pod vedením Andersa Hejlsberga a jeho týmu ve společnosti Microsoft. Projekt Strada byl původně zamýšlen jako nový jazyk pro vývoj webových aplikací, který by měl řešit některé problémy, s nimiž se vývojáři setkávají při používání JavaScriptu.[61, 62]

V průběhu vývoje se ukázalo, že mnoho lidí si přeje jazyk, který by byl kompatibilní s JavaScriptem, ale zároveň by umožňoval psát kód s větší předvídatelností a robustností. Proto bylo rozhodnuto projekt Strada transformovat na nový jazyk s názvem TypeScript. [61, 62]

TypeScript byl oficiálně vydán v říjnu 2012 a ihned získal pozornost vývojářů po celém světě. Od té doby se stal stále populárnějším a získal si širokou podporu ze strany společností a vývojářské komunity.[61]

3.4.2 Statické typování

V TypeScriptu je statické typování založeno na tom, že každá proměnná, parametr funkce, vlastnost objektu nebo návratová hodnota funkce má deklarovaný datový typ. Tyto datové typy jsou definovány pomocí speciálních typových anotací.[63, 64]

Při kompilaci kódu TypeScriptu se provede kontrola typů, která zajišťuje, že proměnné a hodnoty jsou použity v souladu s jejich definovanými datovými typy. Tím se zabrání mnoha běhovým chybám, které by jinak mohly nastat, například při pokusu o použití hodnoty s nesprávným typem nebo při předání nesprávného počtu argumentů do funkce.[64, 65]

Statické typování také umožňuje IDE a vývojářským nástrojům poskytovat pokročilou funkcionalitu, jako například automatické dokončování kódu, navigaci v kódu a refaktorování, což zvyšuje produktivitu vývojářů.[56, 66]

Ve TypeScriptu jsou k dispozici různé typové anotace, včetně základních typů jako například number, string, boolean a object, ale také pokročilých typů jako například union types, intersection types, generics, a další. To umožňuje programátorům přesně specifikovat, jaké typy dat jsou očekávány a jak mají být zpracovávány.[67, 68]

3.4.3 Podpora pro OOP

Díky statickému typování je tak i mnohem jednodušší při vývoji aplikovat objektové paradigma. Protože v klasickém JavaScriptu nelze využít statického typování, jsou obecně třídy nepřehledné a je třeba využívat konvencí a dohod mezi vývojáři. To vede velmi často k velkému množství komentářů a pravidel pro psaní zdrojového kódu, které mohou při chybném použití tvořit zmatek v kódu. [69, 70]

JavaScript také neobsahuje modifikátory přístupu. Této chybějící funkcionality lze dosáhnout funkcemi, které jsou známé jako „uzávěry“ (přeloženo z anglického „closure“). Podpora OOP je v TypeScriptu rozšířena právě o zmíněné modifikátory přístupu, konkrétně o tři modifikátory: [69–71]

- Public
- Private
- Protected

Všechny metody a vlastnosti označené prvním zmíněným modifikátor *„public“*, jsou přístupné jak uvnitř třídy, tak i z vnějšího kódu. Tento modifikátor je zároveň výchozím modifikátorem, který se implicitně nastaví vždy, když není modifikátor přístupu uveden. [69–71]

Naopak všechny vlastnosti a metody, které jsou označené jako *„private“*, jsou dostupné pouze pro kód uvnitř třídy. Není tedy možné, aby kód mimo třídu k takto označeným vlastnostem a metodám přistoupil. [69–71]

Posledním modifikátorem je *„protected“*. Takto označené metody a vlastnosti jsou podobně jako u modifikátoru *„private“*, přístupné jen z vnitřku třídy. Na rozdíl od *„private“*, jsou však metody a vlastnosti dostupné i všem potomkům. Potomci, jsou v tomto kontextu všechny třídy, které dědí z originální třídy, v tomto případě tedy z třídy, kde je *„protected“* metoda či vlastnost definována. [69–71]

4 Vlastní práce

Diplomová práce byla realizována průběžně za použití několika technologií. Jelikož je v dnešní době větší a větší poptávka po mobilních a desktopových aplikací, které spolu sdílí data, byla jako architektura projektu zvolena architektura „client-server“. Kvůli tomuto rozhodnutí, byl vývoj aplikace značně prodloužen, protože bylo potřeba navrhnout nejdříve program, který bude spuštěn a provozován na vzdáleném serveru. Tento program bude ve zbytku práce označován pod pojmem „API“.

Druhou aplikací, která se vyvíjela v rámci diplomové práce je klientská webová aplikace. Tato aplikace je dostupná na veřejné IP adrese a přistupuje se k ní pomocí webového prohlížeče. Tato aplikace je spuštěna a exekuovaná v zařízení klienta.

4.1 Požadovaná funkcionalita

Hlavní a kritickou funkcionalitou je nahrávání CSV souborů s libovolným obsahem do relační databáze. Toto nahrávání musí být vázané k právě přihlášenému uživateli, aby jiný uživatel nemohl bez povolení od autora k souboru přistoupit.

Nahrané a zpracované soubory je možné následně zobrazit ve formě tabulky. Protože může nastat situace, kdy k jednomu souboru potřebuje přístup více lidí, je možné vytvořit skupiny uživatelů a sdílet soubory v rámci skupiny. Tyto skupiny jsou v klientské aplikaci označovány za týmy.

Další funkcionalitou je filtr umožňující vyhledat nahrané CSV soubory podle jejich obsahu. Na základě funkcionality byl tento filtr pojmenován jako „hluboké prohledávání CSV souborů“. Tato funkce umožňuje vyhledat všechny soubory (ve vlastnictví uživatele), které splňují uživatelem nastavený filtr pro jejich obsah.

4.2 Použité technologie a alternativy

Z důvodu možnosti budoucího rozšíření aplikace o programy, mobilní či desktopové aplikace, případně o jinou klientskou webovou aplikaci, byla použita architektura „client-server“. Díky tomuto rozhodnutí bylo možné použít rozdílné technologie, knihovny, programovací jazyky i vývojářské paradigma pro každý projekt.

Autor se rozhodl použít jiné programovací jazyky, technologie i paradigma pro klientskou aplikaci a API. Výhodou tohoto rozhodnutí je, že bylo možné použít industriální, velmi obsáhlé a výkonné jazyky a frameworky umožňující pro serverovou (API) část. Naopak pro klientskou aplikaci byly zvoleny dynamické a méně výpočetně náročné jazyky.

4.2.1 Serverová aplikace

API je realizována pomocí frameworku .NET verze 7 a vysoko-úrovňového programovacího jazyka C#. Aplikace je navržena jako RESTful API. Taková aplikace komunikuje pomocí protokolu HTTP s jinými aplikacemi.

Aby bylo možné se serverovou aplikací komunikovat, je potřeba definovat „endpointy“. Veškeré endpointy jsou definovány uvnitř api kontrolerů, což jsou třídy zpracovávající HTTP požadavky přijaté od jiných aplikací. Obsah této třídy jsou metody, které často reflektují CRUD operace. Pro přenesení dat se používá textový formát JSON.

Pro přístup k databázovému systému byla použita knihovna Entity Framework Core, zkráceně EF (Entity Framework). EF abstrahuje databázový operace a SQL jazyk do C#.

4.2.1.1 Alternativy k .NET

Jednou z nejčastějších alternativ pro .NET API aplikace jsou API aplikace napsané v jazycích Java, GO, PHP, Javascript, Typescript a mnoho dalších. V jazyce PHP je jednou z častých alternativ framework Laravel, naopak v jazyce Javascript a za použití NodeJS, patří mezi populární alternativy k .NET api aplikacím například aplikace napsané za použití frameworků Fastify, NestJS, Koa, či ExpressJS.

4.2.2 Klientská aplikace

Grafická klientská webová aplikace je realizována primárně pomocí jazyků Javascript a Typescript. Jako hlavní knihovna se používá knihovna React JS. Knihovna React umožňuje jednoduše „reagovat“ na jakékoliv změny v rámci jednotlivých komponent v aplikaci. V reálném použití potom dochází k překreslení komponenty, uvnitř které byl změněn stav, díky čemuž aplikace působí responzivně.

Veškeré komponenty jsou navrženy s důrazem na funkcionální paradigma. Díky tomuto přístupu, každá atomická komponenta řeší pouze jeden problém, typickou

atomickou komponentou je například <*LabeledInput* />. Tato komponenta je následně použita v rámci jiných komponent a pokud bude třeba změnit její chování či grafické zpracování globálně, stačí upravit přímo její vlastní zdrojový kód.

Dále jsou v klientské aplikaci použity další jazyky spojené především s vývojem webových aplikací, jako je značkovací jazyk HTML, či jazyk CSS. Pro správu balíčků a knihoven byl použit správce balíčků NPM.

4.2.2.1 Alternativy k ReactJS

Možných alternativ ke zvolenému řešení je celá řada, zástupci jsou především napsání v jazyku Javascript. Patří mezi ně například knihovny, či farmeworky: Vue.js, Angular, či Ember.

Alternativy je možné najít i jiných jazycích a technologiích, například mezi alternativu lze zařadit i framework Blazor, který je napsán v jazyce C#. Tento framework je následně spouštět na straně klienta pomocí Webassembly.

4.2.3 Databázový systém

Jako databázový systém byl použit server Microsoft SQL, konkrétně Microsoft SQL Express 2022. Databáze je kontejnerizována skrze platformu Docker. Microsoft SQL, zkráceně MSSQL, je relační databázový systém, obdobný jiným relačním databázovým systémům, jako je například MySQL, PostgreSQL, SQLlite a další.

MSSQL, stejně jako technologie .NET, je vytvořena společností Microsoft. Tyto dvě technologie mají společnou oficiální podporu a je tedy velmi jednoduché je spolu propojit. Právě toto byl důvod k použití MSSQL databázového systému v návrhu řešení pro diplomovou práci. Databázový systém je používán v jeho výchozí konfiguraci.

4.2.4 Hosting webového serveru

V rámci vývoje aplikace bylo celé řešení, tedy jak klientská aplikace, tak API, tak databázový systém průběžně nasazen a testován na VPS hostingu od společnosti Forpsi.

Databázový systém je pro zjednodušení nasazován v rámci kontejneru Docker. Tento kontejner obsahuje veškeré balíčky a komponenty, které jsou potřeba k provozu

databázového systému Microsoft SQL. Stačí tedy na VPS server nainstalovat balíček „docker“ a „docker-compose“.

Pro hostování API je třeba nainstalovat prostředí pro kompilování a spuštění aplikací vyvinuté na platformě .NET 7. Je potřeba konkrétně balíček „dotnet runtime“, nebo “dotnet SDK”. API je následně také kontejnerizováno a spouštěno pomocí platformy Docker.

Klientská aplikace požaduje k hostování nainstalovaný software NPM, který umožní nainstalovat závislosti, sestavit a zkompilovat řešení.

4.3 Návrh databázového řešení

Struktura návrhu relační databáze obsahuje celkem třináct tabulek. Struktura databáze byla navrhována tak, aby byla normalizovaná do třetí normální formy. Ve většině databázových entit je možné vidět tři opakující se sloupce. Jedná se o sloupce

- Created_at
- Modified_at
- Deleted_at

Tyto tři sloupce mají účel udržování timestamp informací. První zmíněný sloupec má zároveň nastavenou výchozí hodnotu ekvivalentní k vždy aktuálnímu časovému razítku. Jak název vypovídá, jedná se o sloupec určený k evidování informace, kdy byl záznam v databázi vytvořen.

Druhým sloupcem, je sloupec *Modified_at.* Účelem tohoto sloupce je držení informace, kdy byl záznam v databázi změněn. Tato hodnota by v ideálním případě měla být upravována pomocí databázového triggeru. Sloupec však obsahuje ještě jednu informaci a to konkrétně, jestli vůbec byl záznam někdy modifikován. Pokud nabývá sloupec hodnoty *NULL*, víme, že záznam ještě upraven nikdy nebyl.

Posledním zmíněným sloupcem je sloupec *Deleted_at*. Tento sloupec, podobně jako sloupec *Modified_at*, udržuje v praxi dvě informace. Zaprvé slouží jako atribut k bezpečnému mazání záznamů, tedy k mazání, kdy záznam není fyzicky smazán, ale je jen označený za odebraný. Druhou informací, kterou sloupec nese, je informace, kdy byl záznam smazán.

Tyto sloupce se vyskytují ve všech hlavních tabulkách a většině vazebních tabulek. Díky jejich existenci tabulkách vázající jiné entity je možné evidovat například informace, kdy taková vazba vznikla, případně jestli byla vazba odebrána či jinak upravena.

4.3.1 Tabulka Person

Tuto entitu je možné označit za základní stavební kámen návrhu databázového řešení. Tabulka *Person* obsahuje informace o uživatelském účtu. Právě ID této entity se následně používá ve zbytku celého návrhu.

Tato databázová entita neobsahuje přihlašovací informace o uživateli, ale jen informace o jeho účtu. Díky tomu je možné mít vícero účtu svázané s jednou kombinací přihlašovacích údajů. To vede k možnému rozšíření, kdy je umožněno uživateli mít vícero účtů, kdy každý účet používá pro jiný účel.

Tabulka 1 - Struktura databázové entity Person

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	LoginID	bigint(8)
	Firstname	nvarchar(-1)
	Lastname	nvarchar(-1)
	DisplayName	nvarchar(-1)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.2 Tabulka Login

Login, je entita určená pro držení dat ohledně přihlašovacích údajů. Tabulka má existující relační vazbu typu 1:N s tabulkou *Person*, kdy jeden záznam v tabulce *Login*, je možné svázat s vícero záznamy z tabulky *Person.*

Tabulka obsahuje navíc sloupec *Verified_at*, který obsahuje informaci, jestli a kdy byl uživatelský účet ověřen. V aktuální verzi aplikace však není možné účet ověřit, a to z důvodu chybějící implementace posílání emailů s odkazem k ověření.

Tabulka 2 - Struktura databázové entity Login

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
	Username	nvarchar(450)
	Password	nvarchar(-1)
	Verified_at	datetime2(8)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.3 Tabulka Role

Další navrženou entitou je *Role*. Tato tabulka obsahuje informace o roli oprávnění, které systém podporuje. Aktuálně jsou veškeré role a jejich oprávnění definovány staticky přímo ve zdrojovém kódu, avšak tato tabulka je přípravou na potencionální rozšíření v budoucnu. Takové rozšíření by vyžadovalo přidat další tabulku, která by definovala specifické operace. Ty by byly následně svázány s tabulkou rolí, pravděpodobně v vazbou typu 1:N.

Tabulka 3 - Struktura databázové entity Role

Typ	Název sloupce	Datový typ
PK	ID	int(4)
	Slug	nvarchar(-1)
	Name	nvarchar(-1)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

Zde je možné provést optimalizaci a odebrat nadbytečný sloupec *ID*. Každý *slug* je unikátní, a ačkoliv se jedná o řetězec znaků, je možné ho použít jakožto primární klíč.

4.3.4 Tabulka PersonRole

PersonRole je vazební tabulka typu M:N mezi tabulkami *Person* a *Role*. Tato entita neobsahuje trio časových razítek. Účel této entity je pouze pro spojení uživatelského účtu s rolí oprávnění. Popisovaná entita je navíc zcela generována knihovnou Entity Framework Core.

Tabulka 4 - Struktura vazební tabulky PersonRole

Typ	Název sloupce	Datový typ
FK	PersonsID	bigint(8)
FK	RolesID	int(4)

Zdroj: vlastní zpracování (2023)

4.3.5 Tabulka Contact

Databázová tabulka *Contact* obsahuje informace o kontaktních údajích uživatele. Entita obsahuje dva cizí klíče. Jedním je *ContactTypeID*, který odkazuje na typ kontaktního údaje. Druhým cizím klíčem je *PersonID*, vázající záznam uživatele z tabulky *Person*. Obě vazby jsou typu 1:N.

Tabulka 5 - Struktura databázové entity Contact

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	ContactTypeID	bigint(8)
FK	PersonID	bigint(8)
	Value	nvarchar(450)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.6 Tabulka ContactType

Vznik databázové entity *ContactType* je čistě z důvodu normalizace databáze. Aby nedocházelo k opakování jedné a té samé hodnoty v tabulce *Contact*. Zároveň však touto normalizací bylo dosaženo možnosti dynamicky přidávat nové typy kontaktů a velmi jednoduše skrze seznam kontaktních typů filtrovat a vyhledávat.

Tabulka 6 - Struktura databázové entity ContactType

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
	Name	nvarchar(450)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

Nicméně, v této entitě je existence sloupce ID pravděpodobně zbytečná a použít sloupce *Name* jakožto unikátní primární klíč by bylo pro účely diplomové práce dostačující. Tato změna je zde uvedena jakožto návrh případné optimalizace této databázové tabulky.

4.3.7 Tabulka Record

Databázová tabulka *Record* byla navržena za účelem udržení dat nahraných CSV souborů. Obsahuje dva cizí klíče. Prvním cizím klíčem je primární klíč autora záznamu, *PersonID*, který tvoří vazbu typu 1:N, mezi *Record* a tabulkou *Person*. Druhým cizím klíčem je *RecordGroupID*, který je *NULLABLE.* Tento klíč indikuje, že je záznam součástí skupiny souborů. Takovou skupinu lze také označit za virtuální adresář.

Tabulka 7 - Struktura databázové entity Record

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	PersonID	bigint(8)
FK	RecordGroupID	bigint(8)
	Name	nvarchar(-1)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.8 Tabulka RecordData

Pro zajištění požadované funkcionality, kdy je třeba prohledávat obsah CSV souborů, byla navržena databázová tabulka *RecordData* pro ukládání obsahu nahraných tabulkových souborů do relační databáze. Díky tomu je velmi jednoduché prohledávat a filtrovat nahrané soubory dle jejich obsahu. Tabulka je navíc strukturována tak, aby byla schopna pojmout data tabulkových souborů s libovolným obsahem a strukturou.

Databázová entita *RecordData* obsahuje jeden cizí klíč, který tvoří 1:N vazbu s tabulkou *Record*.

Tabulka 8 - Struktura databázové entity RecordData

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	RecordID	bigint(8)
	Row	bigint(8)
	Column	bigint(8)
	Value	nvarchar(-1)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.9 Tabulka RecordGroup

Další databázovou entitou je tabulka *RecordGroup*. Tato tabulka byla navržena pro umožnění seskupování nahraných CSV souborů. Tyto skupiny lze chápat jako virtuální adresáře. Obdobně, jako reálné adresáře například v operačním systému Microsoft Windows, i tyto virtuální adresáře mají stromovou strukturu. Díky rekurzi je tak možné tvořit a zanořovat skupiny uvnitř jiné skupiny.

Entita obsahuje dva cizí klíče. První cizím klíčem je *PersonID*, který tvoří 1:N vazbu s tabulkou *Person* a odkazuje na autora skupiny. Druhým klíčem je *RecordGroupID*. Tento klíč může nabývat hodnoty *NULL* a odkazuje přímo na svého potencionálního rodiče ve stromové struktuře. Pokud tento klíč vyplněný není, jedná se o virtuální složku v kořenovém adresáři.

Tabulka 9 - Struktura databázové entity RecordGroup

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	PersonID	bigint(8)
FK	RecordGroupID	bigint(8)
	Name	nvarchar(-1)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.10 Tabulka PersonGroup

Pro zjednodušení možnosti sdílení souborů a dat mezi uživateli, byla navržena tabulka *PersonGroup*. Každý záznam v této tabulce je instancí jedné uživatelské skupiny. Tyto skupiny jsou často v diplomové práci označovány za týmy. Základní myšlenka za uživatelskými skupinami je možnost sdílet soubory a virtuální složky (skupiny souborů) pro celé týmy, čímž je možné sdílet soubory vícero lidem naráz.

Tabulka 10 - Struktura databázové entity PersonGroup

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	PersonID	bigint(8)
	Name	nvarchar(450)
	DisplayName	nvarchar(-1)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

Databázová entita obsahuje sloupec *PersonID*, který odkazuje na autora skupiny. Díky této informaci je možné povolit specifické operace, jako je například změna názvu skupiny, nebo smazání skupiny, jen tvůrci skupiny.

4.3.11 Tabulka PersonGroupRelations

Databázová tabulka *PersonGroupRelations* má účel vazební tabulky. Tato vazební tabulka byla navržena pro připojení uživatelů k uživatelským skupinám. Každý záznam značí pozvánku uživatele do skupiny v určitém stavu. Tento stav se ukládá do sloupce *State*. V aktuální implementaci API se počítá se třemi stavy, a to konkrétně stavy:

- Čekání na reakci na pozvánku
- Přijetí pozvánky
- Odmítnutí pozvánky

Sloupec *State* nabývá číselných a je tedy možné tyto stavy rozšířit dle budoucích potřeb. Tabulka tvoří M:N vazbu mezi entitami *Person* a *PersonGroup*. Strukturu databázové entity je možné vidět v tabulce níže.

Tabulka 11 - Struktura databázové entity PersonGroupRelations

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	PersonID	bigint(8)
FK	PersonGroupID	bigint(8)
	State	int(4)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.12 Tabulka PersonGroupRecord

Další navrženou vazební tabulkou je tabulka *PersonGroupRecord*. Tato entita byla navržena za účelem možnosti sdílení nahraných CSV souborů se skupinou uživatelů. Tabulka tak tvoří M:N vazbu mezi databázovými entitami *PersonGroup* a *Record*.

Pro jednoduší práci s vazbou zde byl vytvořen primární klíč ID, ačkoliv není potřeba. Lepším způsobem pro vytvoření primárního klíče by zde bylo použití složený primární klíč (z anglického *Composite Primary Key*). Ten by se skládal z unikátní kombinace sloupců *PersonGroupID* a *RecordID*. Tato kombinace by měla být vždy unikátní, protože každý soubor by měl být sdílen v rámci jedné skupiny nanejvýše jednou.

Tabulka 12 - Struktura databázové entity PersonGroupRecord

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	PersonGroupID	bigint(8)
FK	RecordID	bigint(8)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.3.13 Tabulka PersonGroupRecordGroup

Poslední entitou v databázi je vazební tabulka *PersonGroupRecordGroup*. Tato tabulka vznikla za účelem možnosti sdílení souborových skupin s uživatelskými skupinami. Hlavním cílem zde tedy byl use-case, kdy by bylo možné sdílet naplněné virtuální složky napříč skupinami a sdílet tak větší množství nahraných CSV souborů naráz.

Obdobně jako u tabulky *PersonGroupRecord*, i tato tabulka obsahuje navíc primární klíč ve formě sloupce ID, který není v teorii potřeba. I zde by bylo možné použit složený primární klíč, tentokrát ze sloupců *PersonGroupID* a *RecordGroupID*.

Tabulka 13 - Struktura databázové entity PersonGroupRecordGroup

Typ	Název sloupce	Datový typ
PK	ID	bigint(8)
FK	PersonGroupID	bigint(8)
FK	RecordGroupID	bigint(8)
	Created_at	datetime2(8)
	Modified_at	datetime2(8)
	Deleted_at	datetime2(8)

Zdroj: vlastní zpracování (2023)

4.4 REST API

Pro realizaci řešení API (z anglického *Application Programming Interface*) byl použit framework *.NET Core 6*. V průběhu tvorby řešení však byla vydána nová verze frameworku, která slibovala velké zrychlení při používání jazyka LINQ (z anglického *Language-Integrated Query)*, které se používá například v knihovně Entity Framework Core. Jazyk LINQ umožňuje psát například SQL dotazy pomocí jazyka C# a tím snížit počet používaných programovací, či dotazovacích jazyků v aplikaci.

Ačkoliv framework *.NET Core 6* i *.NET Core 7* umožňuje tvořit takzvané minimální rozhraní API, nicméně pro účely diplomové práce bylo nakonec aplikováno tradiční API s řadiči.

4.4.1 Návrh architektury serverového řešení

Při návrhu architektury aplikace byl kladen velký důraz na oddělení logiky do jednotlivých vrstev aplikace. API je rozdělena do několika vrstev, konkrétně do:

- Kontrollery
- Repozitář
- Model

V návrhu API se vyskytuje takzvaný Repository pattern. Tento návrhový vzor se používá k oddělení business logiky, tedy specifické implementace, od datové vrstvy aplikace. Pro oddělení business logiky a datové vrstvy se použily kontrollery a repozitáře. Zároveň zde platí pravidlo, že každá jedna databázová entita má přesně jeden model, který popisuje strukturu dané entity. Ke každé entitě je pak následně přiřazen jeden kontroler, který implementuje a umožňuje manipulaci s entitou. Aplikování dané modifikace následně umožňuje repozitář vázaný s konkrétní entitou.

4.4.1.1 Repozitáře

Třídy obsahující implementaci logiky pro práci s databází nazýváme repozitáře. Každý repozitář je určen pro správu jedné databázové entity a implementuje metody definované v jemu přiřazeném rozhraní. Pro každý repozitář tedy existuje jedno rozhraní. Každé rozhraní definuje všechny veřejně dostupné metody a jelikož jsou prakticky identické a mění se jen a pouze datový typ parametrů a dat, které metoda vrací, byl vytvořen generický základní interface *„IBaseRepository"*.

Toto základní generické rozhraní pro repozitáře vyžaduje při použití předání datového typu, který se používá již pro parametry metody a jakožto výstupní datový typ metod. Zmíněné základní rozhraní definuje metody pro načtení detailu záznamu, načtení všech záznamů, přidání, odebrání a upravení záznamů. Všechny zmíněné metody jsou definovány jak v synchronní, tak asynchronní variantě.

Zdrojový kód 1 - Ukázka generického rozhraní IBaseRepository

```
 1 public interface IBaseRepository<T>
 2 {
 3     T? Get(T entity);
 4     Task<T?> GetAsync(T entity);
 5     IEnumerable<T?> GetAll();
 6     Task<IEnumerable<T?>> GetAllAsync();
 7     T Add(T entity);
 8     IEnumerable<T> Add(IEnumerable<T> entity);
 9     Task<T> AddAsync(T entity);
10     Task<IEnumerable<T>> AddAsync(IEnumerable<T> entity);
11     T? Update(T entity);
12     IEnumerable<T?> Update(IEnumerable<T> entity);
13     Task<T?> UpdateAsync(T entity);
14     Task<IEnumerable<T?>> UpdateAsync(IEnumerable<T> entity);
15     bool Delete(T entity);
16     Task<bool> DeleteAsync(T entity);
17 }
```

Zdroj: vlastní zpracování (2023)

Všechna rozhraní určené pro repozitáře dědí ze základního rozhraní, přičemž efektivně dědí veškeré popsané metody a předávají informaci o datovém typu, který má být použit. Některé rozhraní definují aditivní metody navržené speciálně pro speciální use-case. Typickým příkladem je *„ILoginRepository“*, který definuje aditivní metodu *„LoginUser“*.

Zdrojový kód 2 - Ukázka dědění generického IBaseRepository rozhraní a definování aditivní metody

```
1 public interface ILoginRepository : IBaseRepository<Login>
2 {
3     Login? LoginUser(Login entity);
4 }
```

Zdroj: vlastní zpracování (2023)

Právě díky rozhodnutí, že každý repozitář bude mít vlastní rozhraní, je možné jednoduše navrhnout dvě rozdílné implementace repozitáře na základě jednoho rozhraní. Díky tomuto rozhodnutí není v budoucnu překážkou přejít na jiný databázový systém, nebo nahradit aktuální ORM knihovnu za jiný nástroj či způsob, jak komunikovat s databázový serverem. V takové situaci je dostačující vytvořit novou třídu, která bude dědit ze stejného rozhraní jako třída, kterou se snažíme nahradit. V ten moment nám nebrání nic v nahrazení původní repository třídy za novou bez jakýchkoliv problémů, případně mezi nimi přepínat dle libosti.

Abychom zajistili dostupnost repozitářů v celé API, jsou všechny repository třídy registrované jako „*Scoped*“ služby pod jménem jejich rozhraní. Tato registrace se provádí při každém přijetí HTTP požadavku a k jejímu odstranění dochází při odeslání odpovědi.

Zdrojový kód 3 - Ukázka registrování repozitářů jako služeb

```
1    // Add repositories to the DI container
2    builder.Services.AddScoped<ILoginRepository, LoginRepository>();
3    builder.Services.AddScoped<IPersonRepository, PersonRepository>();
4    builder.Services.AddScoped<IRecordRepository, RecordRepository>();
5    builder.Services.AddScoped<IRoleRepository, RoleRepository>();
```

Zdroj: vlastní zpracování (2023)

V momentě, kdy je repozitář registrován jako služba, je automaticky přidán do kontejneru závislosti (z anglického *Dependency Container*). Díky zaregistrování takovéto služby, je poté možné díky procesu zvaného *vkládání závislostí* (z anglického *Dependency Injection*), vkládat a používat služby v rámci jiných tříd, jako jsou například řadiče.

4.4.1.2 Řadiče

Všechny řadiče (z anglického *Controller*) v API jsou registrované jako endpointy. Endpointem se rozumí koncový bod síťového spojení, nebo jakéhokoliv komunikačního kanálu. Řadič tedy obsahuje metody dostupné na URL adrese specifikované v jeho názvu. V případě kontroleru pojmenovaného „*AccountController*“ by byl tedy dostupný tento endpoint na url *„adresa-k-api/account“*. Seznam základních kontrolerů je obsažen v tabulce níže:

Tabulka 14 - Seznam základních kontrolerů

Název kontroleru (bez slova Controller)	**Popis kontroleru**
Base	Základní řadič (kontroler) definující metody pro získání ID a Emailu přihlášeného uživatele. Tento kontroler je děděn všemi ostatními kontrolery.
Login	Účelem kontroleru *Login* je manipulace se záznamy v databázové entitě *Login*. Používá se především k vytváření nových uživatelských účtů a jejich procházení. Tento řadič neumožňuje přihlašování uživatelů.
Person	Kontroler *Person* umožňuje prohledávat, vytvářet a mazat záznamy v databázové entitě Person.
PersonGroup	Pro správu záznamů uživatelských skupin, v klientské aplikaci označených za týmy, byl navržen kontroler *PersonGroup*.
PersonGroupRelations	Zmíněný kontroler se používá k zobrazení, přijímání a odmítání pozvánek do uživatelských skupin. Jedná se o řadič, který manipuluje se záznamy ve stejnojmenné vazební tabulce.
Record	Pro správu záznamů o nahraných CSV souborech se používá kontroler *Record*. Tento řadič umožňuje zobrazit seznam nahraných souborů, zobrazit detail záznamu včetně obsahu CSV souboru, nahrávat nové soubory a odstraňovat již existující.
RecordData	Řadič umožňující procházet a filtrovat seznam obsahu pro různé nahrané CSV soubory. Kontroler také obsahuje implementaci pro editování obsahu CSV souborů.
Role	Kontroler *Role* definuje operace dostupné pro externí služby pro zobrazení seznamu, detailu a vytváření nové úrovně oprávnění.

Zdroj: vlastní zpracování (2023)

Každý kontroler je navržen tak, aby měl přístup k libovolnému repozitáři v podobě služby. Zároveň má každý řadič k sobě připojenou logovací službu (servisu). Tato služba

umožňuje vypisovat do terminálu, ve kterém je aplikace spuštěna, informace o aktuálně zpracovávané funkci pro účely debugování, nebo hlášení chyb a upozornění.

Zdrojový kód 4 - Ukázka GET metody implementované v PersonController řadiči

```
1    [HttpGet("{id}")]
2    public async Task<ActionResult<PersonDTO>> Get([FromRoute] uint id)
3    {
4    Person? data = await this._repository.GetAsync(new Person{ ID = id });
5        if (data is null) return NotFound();
6        return Ok(_mapper.Map<PersonDTO>(data));
7    }
```

Zdroj: vlastní zpracování (2023)

V rámci vývoje byly vytvořeny navíc tři kontrolery, které nemají k sobě přiřazený unikátní databázový model. Jedná se o řadiče:

- AuthController
- RecordSearchController
- RecordCanvasController

Jak již název napovídá, kontroller *„AuthController"* implementuje funkcionalitu pro přihlašování uživatelů do aplikace. Tato implementace je následně detailněji vysvětlena v kapitole 4.4.6 Autentifikace a autorizace.

Dalším zmíněným řadičem je *„RecordSearchController"*. Tento kontroler implementuje prohledávání nahraných tabulkových souborů na úrovni relační databáze.

Posledním zmíněným kontrolerem je *„RecordCanvasController"*, který obsahuje metody určené pro nalezení všech souborů vázané s uživatelem, jenž tento dotaz inicializoval. Zároveň se soubory se hledají i vytvořené virtuální složky, které jsou s uživatelem svázány. Předáním boolean parametru *„Shared"* je možné specifikovat, jestli si uživatel přeje získat pouze soubory a složky jím vytvořené, nebo soubory a složky, které mu jsou nasdíleny v rámci týmů.

Všechny řadiče, až na *„AuthController"* a POST metodu v *„LoginController"* třídě vyžadují autentizaci a autorizaci. Není tedy možné k těmto řadičům přistoupit bez platného oprávnění. Pokud klientská aplikace zasílá neplatné, nebo žádné autentizační údaje, API vrátí HTTP odpověď s chybovým kódem „*401 Unauthorized*", což znamená, že

klientská aplikace musí představit platné autentizační údaje pro získání přístupového tokenu a následně k tomuto zdroji přistoupit s platným tokenem.

4.4.2 Modely databázových entit

Všechny modely byly vytvořeny podle struktury databázových entit. Každý model však rozšiřuje základní model, pojmenovaný *"BaseModel"*, který obsahuje časová razítka, která jsou ve všech hlavních entitách. Jediné entity, které tato razítka nemusí nutně obsahovat jsou vazební tabulky.

Již zmíněná třída *„BaseModel"* obsahuje konkrétně tři časová razítka:

- Created_at
- Modified_at
- Deleted_at

Atribut „*Created_at"* bude vždy obsahovat hodnotu. Tento atribut nese informaci o časovém razítku, kdy byl záznam v databázi vytvořen. Při vytváření databázových entit přímo na úrovni databáze má tento sloupec nastaven defaultní hodnotu *CURRENT_TIMESTAMP*. Při vytvoření nového záznamu bude tedy vždy časové razítko vytvoření záznamu na úrovni databáze.

Dalším atributem je *„Modified_at"*, který umožňuje nastavit hodnotu, kdy byl záznam upraven. Díky této informaci je možné nejen říci, kdy přesně byl záznam v databázi modifikován, ale že vůbec byl modifikován.

Posledním děděným atributem je *„Deleted_at"*. Tento sloupec slouží k implementaci funkcionalitě známé jako *„Soft delete"*, v češtině známé jako *„měkké smazání"*. Díky tomuto sloupci je možné označit záznam za smazaný, a zároveň mít informaci, kdy přesně byl záznam smazán. Takový záznam by měl být při výchozím nastavení odfiltrován a nevybrán z databáze. Tato funkcionalita eventuelně umožňuje obnovit smazané záznamy, případně umožnit administrátorovi „skrýt" některé záznamy tím, že je označí za smazané. Nadefinované modely v rámci API je možné vidět v tabulce níže.

Tabulka 15 - Seznam modelů databázových entit

Název modelu	**Popis modelu**
BaseModel	Model obsahující základní časové razítka, která jsou obsažena ve všech modelech.
Contact	Entita obsahující kontaktní údaj pro, který je svázán s Person.
ContactType	Typ kontaktu, předpokládá se naplnění hodnotami jako je „Email“, „Telefon“ a další.
Login	Login je entita obsahující přihlašovací údaje.
Person	Model obsahující informace o osobě, tento model je svázán s entitou Login. Každý jeden Login může mít více Person.
PersonGroup	PersonGroup model obsahuje informace o vytvořeném týmu, jinak řečeno skupině person.
PersonGroupRecord	Vazební tabulka propojující entity PersonGroup a Record. Existence záznamu v této tabulce znamená existující sdílení mezi entitami Record a PersonGroup.
PersonGroupRecordGroup	Vazební tabulka propojující entity PersonGroup a RecordGroup. Existence záznamu v této tabulce znamená existující sdílení mezi entitami PersonGroup a RecordGroup.
PersonGroupRelations	Vazební tabulka spojující entity PersonGroup a Person. Existující záznam reflektuje stav pozvánky a případné spojení osoby a týmu.
Record	Entita obsahující základní informace o nahraném záznamu.
RecordData	RecordData je tabulka držící obsah nahraného záznamu, má přímou vazbu na entitu Record.
Role	Model definující role používané v rámci oprávnění v kontextu API.
Tag	Tag je model, který definuje skupinové označení, které je možné přiřadit k entitám Record.

Zdroj: vlastní zpracování (2023)

4.4.3 Data Transfer Object (DTO)

DTO jsou objekty, které jsou určeny pro komunikaci s externími aplikacemi. Celá myšlenka za DTO je relativně jednoduchá. Ne vždy chceme zobrazit aplikacím třetích stran veškeré atributy co máme v databázi, DTO jsou jednoduchým řešením, které tento problém řeší. Data, která jsou načítána z databáze ve formě databázových modelů jsou na endpointech před odesláním přemapována do jednoduchých objektů, které definují entity tak, jak by měly být modely viditelné pro externí aplikace.

Další výhodou DTO je jejich přidaná míra abstrakce. Kdyby došlo k nutnosti zásadní úpravy databázové struktury, není nutné přeprogramovat či jinak upravit externí aplikace, které s API pracují. V takovém případě můžeme v teorii pokračovat se stávajícími DTO objekty, případně tyto objekty rozšířit o nové atributy a stále podporovat ty původní. Na endpointech, tedy v kontrolerech, následně tyto DTO přemapujeme do databázových modelů.

Dalším využitím DTO je zjednodušení relačních vazeb. Pokud máme v modelech definovanou oboustrannou vazbu, kdy pomocí ORM jako je například Entity Framework Core jsme schopni dotazovat k relačně spojenými entitami z obou stran relace, může při načtení takových entit nastat situace, kdy dojde k zacyklení při pokusu o serializaci modelů do textového JSON formátu. V případě oboustranné vazby může dojít k nekonečné smyčce mezi referencemi. Při návrhu DTO bylo uplatněno zjednodušení relací, konkrétně aplikování pouze jednostranných vazeb, čímž bylo zabráněno možnému zacyklení.

4.4.4 Dotazovací modely

Pro účely filtrování bylo potřeba navrhnout řešení předávání dat určených k filtrování. Prvotně nebylo zapotřebí implementovat speciální řešení pro filtrování, protože bylo dostačující pouze naplnit DTO entitu a umožnit její předání pro HTTP operace typu GET. Toto řešení umožňovalo specifikovat například ID autora u vyhledávání, či filtrování záznamů, nebo vyhledávat nahrané CSV soubory podle shody obsahu jména záznamu.

Toto řešení bohužel nebylo dostačující pro účely předání informací, které nejsou obsaženy v základním DTO modelu. Mezi takové parametry můžeme zařadit například informaci o počtu záznamů, které je třeba přeskočit, nebo maximální počet záznamů, který by měl být z databáze získán a následně vrácen klientské aplikaci.

Pro dosažení takové rozmanitosti a ideálně co nejobecnější implementace, která bude lehce znovupoužitelná i na jiných endpointech, vznikl základní dotazovací model pojmenovaný *„QueryableRequest"*. Tento model je definován rozhraním *„IQueryableRequest"*.

Zdrojový kód 5 - Ukázka rozhraní IQueryableRequest modelu

```
1 public interface IQueryableRequest
2 {
3     int? Skip { get; set; }
4     int? Limit { get; set; }
5 }
```

Zdroj: vlastní zpracování (2023)

Tento model aktuálně přidává možnost klientské aplikaci předat informaci o počtu záznamů, které mají být přeskočeny a limitovat počet vrácených záznamů, nicméně je možné ho rozšířit o další možné nastavení. Tento model je následně děděn v rámci konkrétnějších dotazovacích modelů, které jsou navrženy pro filtrovaní či konkrétnější dotazování existujících databázových modelů, které jsou vypsané níže:

- LoginRequest
- PersonRequest
- PersonGroupRelationsRequest
- RecordCanvasRequest
- RecordRequest

Protože účel dotazovacích modelů je rozšířit či jinak upravit generovaný dotaz pro databázi, bylo třeba implementovat zpracování dotazovacích modelů na úrovní repozitářů, které obsahují business logiku pro práci s databází. Jelikož aktuální implementace dotazovacích modelů umožňuje přidávat filtry jen pro selektování dat z databáze, byl navrženo generické rozhraní *„IQueryableRepository"*, které definuje metodu pro selektování dat z databáze včetně možnosti předání dotazovacího modelu.

Zmíněné generické rozhraní očekává dva datové typy. První předaný datový typ se následně používá jakožto datový typ, ve kterém je očekáván výsledek celé metody, zatímco druhý datový typ se používá jakožto typ, ve kterém je předáván parametr obsahující žádané filtry s hodnotami.

Zdrojový kód 6 - Rozhraní IQueryableRepository

```
1 public interface IQueryableRepository<T, R>
2 {
3     IEnumerable<T?> GetAll(R? filter);
4     Task<IEnumerable<T>> GetAllAsync(R? filter);
5 }
```

Zdroj: vlastní zpracování (2023)

Z tohoto generického rozhraní se následně dědí ve rozhraních pro jednotlivé repozitáře, které toto dotazování podporují. Takovým repozitářem je například *„IPersonRepository"*.

Zdrojový kód 7 - Ukázka dědění IQueryableRepository v IPersonRepository

```
1 public interface IPersonRepository : IBaseRepository<Person>,
2                  IQueryableRepository<Person, PersonRequest>
3 {
4     IEnumerable<Person> GetByLoginId(uint id);
5 }
```

Zdroj: vlastní zpracování (2023)

K aplikování filtrů je třeba v poslední řadě implementovat zpracování předaných dotazovacích modelů na úrovni repozitářů. Pro tento účel byla vytvořena pomocná statická generická třída *„QueryableHelper"*, která obsahuje statickou generickou metodu *„ApplyQuery"*. Protože je jak třída, tak její metoda statická, je možné zavolat metodu odkudkoliv bez nutnosti inicializování třídy. Funkce očekává návratový datový typ, předanou instanci databázového kontextu (více v kapitole 4.4.5.1 Databázové kontexty), která bude nadále modifikována a instanci dotazovacího modelu obsahující informace k filtrování. Návratovou hodnotou je upravená instance databázového kontextu, ze kterého je následně generován SQL kód.

Zdrojový kód 8 - Pomocná metoda ApplyQuery pro aplikování dotazovacích modelů

```
1   public static IQueryable<T> ApplyQuery(
2     IQueryable<T> ctx,
3     IQueryableRequest? filter
4   )
5   {
6       if (!(filter?.Skip is null) && filter.Skip > 0)
7       {
8         ctx = ctx.Skip((int) filter.Skip);
9       }
10
11      if (!(filter?.Limit is null) && filter.Limit > 0)
12      {
13        ctx = ctx.Take((int) filter.Limit);
14      }
15      return ctx;
16  }
17 }
```

Zdroj: vlastní zpracování (2023)

Některé repozitáře pak mají své vlastní privátní metody určené ke zpracování a aplikování rozšířených dotazovacích modelů, které jsou už vázány ke struktuře daného databázového modelu.

4.4.5 Entity Framework Core

Pro přístup k databázi je v použita ORM (zkratka z anglického *Object Relational Mapping*) knihovna Entity Framework Core, zkráceně EF Core. Tato knihovna umožňuje používat programovací jazyk C# k dotazování databáze pomocí dotazovacího jazyka SQL. Jednou z hlavních výhod EF Core je, že celý codebase API je v jazyce C# a nedochází tak ke kombinaci několika jazyků uvnitř jedné aplikace.

Knihovna byla především použita z důvodu zjednodušení mapování mezi relacemi databázových entit a převádění z databázových entit (tabulek) do modelů navržených v jazyce C#. Pro možné použití EF Core je třeba vytvořit a zaregistrovat databázový kontext jakožto dostupnou servisu v rámci frameworku .NET.

Při registrování kontextu je třeba nutné předat takzvaný *„ConnectionString“*, který je nosičem informace, na které adrese je možné databázový server najít a jaké přihlašovací údaje se mají k připojení k databázovému serveru použít.

4.4.5.1 Databázové kontexty

V aplikaci jsou připraveny dva databázové kontexty. Hlavním kontextem je *„BaseDbContext"*, který zastává hlavní funkcionalitu. Tento hlavní kontext zpřístupňuje veškeré databázové entity. Tyto entity jsou v kontextu a ve zbytku API reprezentovány již zmíněnými databázovými modely. Kontext rozšiřuje děděnou metodu z databázového kontextu, který je přímo zaimplementován v knihovně Entity Framework Core.

Zmíněná metoda je nazvána *„OnModelCreating"* a umožňuje definovat jak relační vazby databázových entit, jejich klíčové hodnoty, indexy, omezení, chování a mnoho dalšího. Pro zajištění správného chodu databáze navržené pro diplomovou práci, se uvnitř zmíněné metody definují především všechny vazební tabulky v databázi. Tyto definice obsahují především určení cizích klíčů, typ relace a nastavení způsobu chování relací a entit při mazání záznamů.

Zdrojový kód 9 - Ukázka definování relací uvnitř funkce "OnModelCreating"

```
1       modelBuilder.Entity<PersonGroupRelations>()
2           .HasIndex((x) => new {
3               x.PersonID,
4               x.PersonGroupID,
5               x.Deleted_at
6            })
7           .IsUnique();
8
9        modelBuilder.Entity<Record>()
10          .HasOne(x => x.RecordGroup)
11          .WithMany(x => x.Records)
12          .OnDelete(DeleteBehavior.NoAction);
13
14       modelBuilder.Entity<PersonGroupRelations>()
15          .HasOne(x => x.Person)
16          .WithMany(x => x.PersonGroups)
17          .HasForeignKey(x => x.PersonID)
18          .OnDelete(DeleteBehavior.NoAction);
19
20       modelBuilder.Entity<PersonGroupRelations>()
21          .Property(x => x.State)
22          .HasDefaultValue(PersonGroupRelationState.waiting);
23
24       modelBuilder.Entity<PersonGroupRelations>()
25          .HasOne(x => x.Group)
26          .WithMany(x => x.Members)
27          .HasForeignKey(x => x.PersonGroupID)
28          .OnDelete(DeleteBehavior.NoAction);
```

Zdroj: vlastní zpracování (2023)

4.4.5.2 Limitovaný kontext

Databázové kontexty je i možné použít pro tvorbu možných oprávnění. Pro potřeby diplomové práce bylo třeba navrhnout řešení, jak bezpečně a konzistentně zajistit znemožnění uživatelům s nízkým oprávněním provádět jen určité operace. Ačkoliv je toto možné provést na úrovní kontrolerů, nebo repozitářů, je bezpečnější vytvořit omezení dle oprávnění na té nejnižší možné úrovni, ideálně uvnitř samotné databáze.

Na základě vytvoření řešení pro tento problém, byl navržen databázový kontext, který globálně omezuje databázové operace generované uvnitř API. Kontext byl pojmenován podle úrovně oprávnění, pro kterou byl určen, a to konkrétně *„PersonDbContext“*. Protože je potřeba, aby i tento kontext, stejně jako hlavní databázový kontext, umožňoval přistoupit k veškerým databázovým entitám a obsahoval informace

o cizích klíčích, indexech, relačních vazbách a chování entit mezi relačními vazbami, dědí tento kontext přímo z hlavního kontextu.

Hlavním rozdílem je však rozšíření děděné již zmíněné funkce *„OnModelCreating"*. V rámci tohoto kontextu jsou nastaveny takzvané globální filtry. Jedná se o filtry, které jsou nastaveny pro každou potřebnou databázovou entitu a omezují manipulaci záznamů v databázových entitách pouze na záznamy, které jsou spjaté s přihlášeným uživatelem.

Zdrojový kód 10- Ukázka rozšířené "OnModelCreating" funkce o globální filtry

```
1     protected override void OnModelCreating(ModelBuilder mb)
2     {
3         mb.Entity<Record>()
4             .HasQueryFilter(b => b.PersonID == _personId);
5         mb.Entity<RecordData>()
6             .HasQueryFilter(b => b.Record != null
7                 && b.Record.PersonID == _personId
8             );
9         mb.Entity<Person>()
10            .HasQueryFilter(b => b.ID == _personId);
11        mb.Entity<Contact>()
12            .HasQueryFilter(b => b.PersonID == _personId);
13        mb.Entity<Login>()
14            .HasQueryFilter(b => b.Persons != null
15                && b.Persons.FirstOrDefault(
16                    (y) => y.ID == _personId
17                ) != null
18            );
19        mb.Entity<PersonGroupRelations>()
20            .HasQueryFilter(b => b.Group!.PersonID == _personId
21                || b.PersonID == _personId
22            );
23
24        base.OnModelCreating(mb);
25    }
```

Zdroj: vlastní zpracování (2023)

Globální filtry jsou automaticky nastaveny vždy před finálním sestavením databázového dotazu. Díky tomu je možné mít sjednocenou business logiku určenou k vytváření databázových dotazů uvnitř repozitářů. V momentě, kdy uvnitř repozitáře dochází k vytvoření dotazu a jeho následného provedení, je při vytváření dotazu aplikován i globální filtr. V případě nutnosti je však možné při sestavování databázového dotazu pomocí Entity Framework Core globální filtr deaktivovat. Tato možnost je použita při sestavování dotazů pro zobrazení vyfiltrovaných seznamů týmových pozvánek.

Zdrojový kód 11 - Ukázka deaktivování globálních query filtrů

```
1        if (filter?.State is not null)
2        {
3          // Ignore global filter in this usecase;
4          ctx = ctx.IgnoreQueryFilters();
5          ctx = ctx.Where((x) => x.State == filter.State);
6        }
```

Zdroj: vlastní zpracování (2023)

4.4.6 Autentifikace a autorizace

Proces autentifikace se provádí uvnitř řadiče „*AuthController*“. Tento řadič umožňuje pouze přijetí a zpracování HTTP metody POST. Metoda očekává na vstupu instanci datového typu, který je identický s databázovým modelem „*Login*“. Konkrétně se kontroluje naplnění modelových vlastností *Username* a *Password*. V rámci návrhu aplikace se *Username* očekává ve formátu emailu. Všechna hesla jsou šifrována pomocí šifrovacího algoritmu SHA256.

Kombinace emailu a hesla je následně porovnána s databází a pokud byla nalezena shoda, je vygenerován autorizační token, konkrétně JWT. Tento token se následně posílá spolu se informacemi o přihlášeném uživateli zpět klientské aplikaci.

4.4.6.1 JSON Web Token

JWT (zkráceně z anglického *JSON Web Token*) je standardizovaný formát tokenu, který se využívá k autentizaci a autorizaci. JWT je dělen na tři části, konkrétně jeho součástí je: [72]

- Hlavička
- Tělo
- Podpis

Hlavička obsahuje informace o použitém algoritmu pro podepsání tokenu a informaci o jaký typ tokenu se jedná. Do těla jsou zakódovány informace, o přihlášeném uživateli a další aditivní data, jako je například čas expirace tokenu. Na základě těla, hlavičky a tajného klíče je vypočten podpis, který slouží jako nositel, že data nebyla během přenosu upravena. Pokud upravena byla, nový kontrolní výpočet nebude odpovídat originálnímu podpisu. [72]

Pro generování JWT se v diplomové práci byla navržena třída *„JWTManager"*, která je registrována jako interní služba. Tato třída umožňuje generovat již zmíněné tokeny, na základě předaných informacích o přihlášeném uživateli. Do těla tokenu se ukládá především následující informace:

- ID uživatelských přihlašovacích údajů
- ID uživatelského účtu, ke kterému se váže veškerá funkcionalita aplikace
- Role oprávnění uživatele
- Přihlašovací jméno uživatele (email)
- Metadata

Zmíněná metadata obsahují data například o entitě, která token vygenerovala, podepsala, za jakou dobu tokenu propadne platnost a podobně. V aktuálním nastavení je každý token platný dvě hodiny. Důležitou aktuální limitací systému je však možnost nastavení pouze jedné role oprávnění. Ačkoliv systém jako takový, umožňuje přidat uživateli vícero rolí oprávnění, je možné předat pouze jednu úroveň oprávnění do těla JWT, jak možné vidět na ukázce zdrojového kódu níže, konkrétně na řádcích 13 až 17.

Zdrojový kód 12 - Implementace metody GenerateToken v JWTManger službě

```
1   public JwtSecurityToken GenerateToken(Login user)
2   {
3       var jwt = _configuration.GetSection("Jwt").Get<Jwt>();
4       if (jwt is null) throw new NullReferenceException(
5           "Could not load `JWT` section in appsetings"
6       );
7       List<Claim> claims = new List<Claim>()
8       {
9           new Claim(JwtRegisteredClaimNames.Sub, jwt.Subject),
10          new Claim(JwtRegisteredClaimNames.Jti,
11              Guid.NewGuid().ToString()
12          ),
13          new Claim(ClaimTypes.Role,
14              user.Persons?.FirstOrDefault()?.Roles?
15                  .FirstOrDefault()?.Name
16              ?? ""
17          ),
18          new Claim(ClaimTypes.Name, user.Username),
19          new Claim(ClaimTypes.Sid, user.ID.ToString()),
20          new Claim(ClaimTypes.PrimarySid,
21              user?.Persons?.FirstOrDefault()?.ID.ToString() ?? ""
22          ),
23      };
24      SymmetricSecurityKey key = new SymmetricSecurityKey(
25          System.Text.Encoding.UTF8.GetBytes(jwt.Key)
26      );
27      SigningCredentials signIn = new SigningCredentials(
28          key,
29          SecurityAlgorithms.HmacSha256
30      );
31      JwtSecurityToken token = new JwtSecurityToken(
32          jwt.Issuer,
33          jwt.Audience,
34          claims,
35          expires: DateTime.Now.AddHours(2),
36          signingCredentials: signIn
37      );
38      return token;
39  }
```

Zdroj: vlastní zpracování (2023)

Tato limitace je však snadno opravitelná. Omezení totiž vychází z předpokladu, že se role nebudou u uživatelských účtu často měnit. V ideální implementaci by neměla být předávána informace o roli skrze JWT, ale měla by být vždy čerstvě načtena z databáze, při příchozí žádosti o provedení operace na straně API. V aktuální implementaci se tak

neděje, ale „naivně“ se očekává, že uživatel má stále stejnou roli jako měl při jeho posledním přihlášení, a tedy při posledním generování JWT pro tohoto uživatele.

Po úspěšném vygenerování JSON Web tokenu a odeslání tohoto tokenu zpět klientské aplikaci, je teď pro klientskou stranu možné dotazovat endpointy API, které jsou dostupné jen přihlášeným uživatelům. Pro zjištění, jestli dotaz vzešel z ověřené klientské aplikace se očekává, že každý dotaz posílaný pomocí HTTP bude ve své hlavičce obsahovat platný vygenerovaný JWT.

Tento JWT token je v API vždy ověřen před zpracováním dotazu. Pokud byl token neplatný nebo uživatel nemá oprávnění k přístupu k zabezpečenému endpointu, je automaticky vrácena chybová hláška *„Unauthorized"* s HTTP kódem 401, označující neoprávněný přístup.

4.4.6.2 Zvolení databázového kontextu

Při každém přijetí HTTP požadavku se provádí zvolení databázového kontextu na základě přijatého JWT. Pokud žádný token přijat nebyl, nebo není validní, vybere se automaticky základní databázový kontext *„BaseDbContext“*. Pokud však validní JWT byl přijat, získají se z přijatého tokenu uložené informace o uživatelském účtu a jeho roli. Pokud role v tokenu nenese hodnotu „Admin“ a lze validně zpracovat informace o uživatelském účtu, je inicializován limitovaný databázový kontext *„PersonDbContext“*. Pokud je v JWT uložena role „Admin“, je aplikovaný základní databázový kontext, které žádné limitace nepřidává.

Zvolený kontext je následně registrován jako dočasná služba (z anglického *Transient service*) a jako všechny ostatní služby, je přidána do kontejneru závislosti.

Zdrojový kód 13 - Funkce zvolení správného databázového kontextu

```
1     // Add specific DbContext as a service
2     builder.Services.AddTransient<BaseDbContext>(serviceProvider =>
3     {
4         var context = serviceProvider.GetService<IHttpContextAccessor>();
5         var options = new DbContextOptionsBuilder<BaseDbContext>();
6         options.UseSqlServer(
7           builder.Configuration.GetConnectionString("DefaultConnection")
8         );
9
10        if (builder.Environment.IsDevelopment()) {
11          options.LogTo(Console.WriteLine, LogLevel.Debug);
12          options.EnableDetailedErrors();
13          options.EnableSensitiveDataLogging();
14        } else {
15          options.LogTo(Console.WriteLine, LogLevel.Warning);
16        }
17
18        if (context?.HttpContext?.User?
19          .Identity?.IsAuthenticated == true
20        )
21        {
22            // Try to find PrimarySID from JWT claims (PersonId)
23            string? jwtPersonId = context?.HttpContext?.User.Claims?
24              .FirstOrDefault(c => c.Type == ClaimTypes.Sid)?.Value;
25            string? roleName = context?.HttpContext?.User.Claims?
26              .FirstOrDefault(c => c.Type == ClaimTypes.Role)?.Value;
27            uint personId;
28            if(uint.TryParse(jwtPersonId, out personId)
29               && roleName != "Admin"
30            )
31            {
32                return new PersonDbContext(options.Options, personId);
33            }
34        }
35
36        return new BaseDbContext(options.Options);
37    });
```

Zdroj: vlastní zpracování (2023)

4.4.7 Nahrávání a zpracování souborů

Pro nahrávání a zpracování CSV souborů byl navržen zabezpečený řadič *„RecordController“*. Zmíněný řadič definuje a implementuje metodu *POST*, která přijímá vstup datového typu *RecordDTO* a vstupní parametr *„file“* v datovém typu *IFormFile*. Vstupní parametr *„file“* je pro tuto metodu povinný a pokud nebyl metodě předán, je klientské aplikaci navrácena HTTP chybná odpověď *„400 Bad Request“*.

Po zvalidování vstupů je následně soubor dočasně nahrán do složky *„Temp“*, ze kterého je následně načten a uložen do paměti. Dalším krokem je zpracování obsahu souboru. Pro tento proces byla vytvořena pomocná třída *FileHandlerProviderFactory*. Tato třída je schopna na základě přípony nahraného souboru najít a inicializovat třídu určenou pro zpracování souborů s touto příponou.

Zdrojový kód 14 – Ukázka pomocné třída IFileProviderFactory

```
1   public FileHandlerProviderFactory()
2   {
3       Type fileHandlerProviderType = typeof(IFileHandlerProvider);
4       _fileHandlerProviders =
5         fileHandlerProviderType.Assembly.ExportedTypes
6         .Where(x => fileHandlerProviderType.IsAssignableFrom(x) &&
7                     x is { IsInterface: false, IsAbstract: false })
8         .Select(x =>
9         {
10          var parametlessCtor = x.GetConstructors().SingleOrDefault(
11            c => c.GetParameters().Length == 0
12          );
13          return parametlessCtor is not null
14            ? Activator.CreateInstance(x)
15            : throw new ArgumentException("Missing arguments");
16        })
17        .Cast<IFileHandlerProvider>()
18        .ToImmutableDictionary(x => x.FileExtension, x => x);
19  }
```

Zdroj: vlastní zpracování (2023)

Tato třída je obzvláště zajímavá použitím nástroje *„Reflection“*. Tento nástroj umožňuje získávat informace o typech a jejich vlastnostech za běhu programu. Po úspěšném nalezení kompatibilní třídy, se přejde ke zpracování obsahu CSV souboru nalezenou třídou. Pro účely diplomové práce byla naimplementována pouze třída určená ke zpracování CSV souborů, nicméně touto abstrakcí je velmi jednoduché rozšířit aktuální implementaci i o jiné tabulkové formáty, jako je například XLSX.

Po úspěšném zpracování souboru je smazán originální soubor z diskového uložiště serveru a zpracovaný soubor je následně nahrán do databáze, včetně jeho obsahu. Metoda je ukončena vrácením odpovědi s informacemi o zpracovaném souboru, případně vrácením chybové hlášky.

4.5 Klientská aplikace

Jak již bylo zmíněno výše, klientská webová aplikace je napsána primárně v kombinaci jazyků Javacript a Typescript (ve verzi 4.8.2). Zároveň byla použita knihovna React ve verzi 18.2.0.

Pro stylování aplikace byly použity jazyky CSS a SCSS. V průběhu vývoje aplikace byla ještě přidána knihovna *Tailwind CSS*, která umožňuje deklarovat CSS styly pomocí tříd, které odpovídají jednotlivým vlastnostem. Příkladem může být třída *„flex“*, která automaticky nastaví HTML prvku *„display: flex“* vlastnost. Tailwind urychluje a zjednodušuje tvorbu responzivních aplikací.

4.5.1 Výhody funkcionálního přístupu

Jelikož byla k návrhu a vývoji aplikace použitá knihovna React, byl i samotný návrh komponent, funkcí a všeobecně zdrojového kódu ve funkcionálním paradigma. Tento přístup přináší velké množství výhod. Mezi typické výhody tohoto paradigma patří především jednodušší správa zdrojového kódu, což se následně odráží ve snížené složitosti při navrhování a psaní automatizovaných testů, nebo případnému rozšiřování aplikace. Jednodušší správa zdrojového kódu nastává díky myšlence čistých funkcí. Takové funkce neobsahují žádný kód produkující vedlejší efekty a jejich výsledek je díky tomu předvídatelnější. Výsledek funkce v takovém případě ovlivňují pouze vstupní parametry.

Protože nedochází uvnitř čistých funkcí k vedlejším efektům, je možné takové funkce relativně jednoduše paralelizovat a aplikovat výhody asynchronního programování. Což je velikou výhodou v momentě, kdy je třeba aplikaci škálovat, protože funkce tak nemají závislosti, kvůli kterým by bylo škálování aplikace méně efektivní.

Aplikování čistých funkcí a všeobecně funkcionální paradigma je v knihovně React časté, právě protože React využívá komponentní architekturu, kdy každá komponenta by měla být generická, a tak v ideálním prostředí znovupoužitelná i v jiném use-case.

4.5.2 Kontext v knihovně React

V rámci realizace aplikace byla použita zabudovaná funkcionalita v knihovně React, konkrétně funkce kontextu. Hlavní využití kontextů je sdílení stavů mezi komponenty. V rámci celé klientské aplikace je nutné držet několik informací neustále dostupné

pro všechny existující komponenty. Jedná se zejména o informace přihlášeného uživatele, aktuální nastavení barevného schématu pro aplikaci a aktuální stav bočního navigačního panelu.

Tabulka 16 - Seznam kontextů klientské aplikace

Kontext	Popis kontextu
AuthContext	Kontext obsahující informace přihlášeném uživateli a aktivního JWT tokenu.
ColorModeContext	Kontext umožňující nastavit světlý nebo tmavý režim aplikace. Funkcionalitu je zároveň možné rozšířit o další barevné profily.
SidebarContext	Kontext obsahující informaci o stavu bočního navigačního panelu. Mezi ukládané stavy patří informace, jestli je panel minimalizován nebo v plné šířce a aktivní navigační záložka.

Zdroj: vlastní zpracování (2023)

Pro každý kontext bylo třeba připravit takzvaný *Provider*, funkci, která umožňuje manipulovat s přiřazeným kontextem. Vytvoření poskytovatelé jsou registrování v souboru *App.tsx*. Provider komponenty jsou pojmenovány podle jim přiřazených kontextů následovně:

- AuthProvider
- ColorModeProvider
- SidebarProvider

Každý poskytoval obsahuje implementaci určenou pro získání dat z kontextu a nastavení hodnoty v kontextu. Pro vyšší míru abstrakce, a především co nejjednodušší způsob používání poskytovatelů uvnitř komponent, byl ke každému poskytovateli vytvořen „*hook*“.

4.5.3 Hooky

Při implementaci funkcí klientské aplikace byly použity funkce zvané Hooky. Tyto funkce se používají primárně pro sdílení stavů mezi komponentami. Po zavolání hook funkce se vygeneruje objekt, který umožňuje přistoupit ke stavům mezi komponentami, případně provádět jiné operace.

V rámci aplikace se používají různé interní hooky zabudované přímo v knihovně *React*, nebo jiných knihovnách, jako je například knihovna *ReactQuery*. Při návrhu a realizace řešení bylo zapotřebí vytvořit pár hook funkcí na míru.

4.5.3.1 Komunikace s API

Jelikož je API vzdálená aplikace, která může být provozována na jiném serveru, než je klientská aplikace, je třeba zajistit způsob komunikování klientské aplikace s API. Jazyk Javascript obsahuje základní funkci určenou pro komunikaci pomocí protokolu HTTP, funkci *"fetch"*. Pro úspěšnou komunikaci s API je nutné nastavit hlavičky popsané v tabulce níže.

Tabulka 17 - Seznam nutných HTTP hlaviček pro komunikaci mezi klientem a API

Parametr	Popis
Content-Type	Označení formátu a znakové řady obsahu zasílaného z klienta na server.
Accept	Označení formátu a znakové řady, který klient přijímá od serveru.

Zdroj: vlastní zpracování (2023)

Při pokusu o nastavení a předání zmíněných hlaviček do funkce *„fetch"* docházelo při vývoji k problémům, při kterých nebyly zmíněné hlavičky validně nastaveny. Dalším zásadním problémem je chybějící automatický převod odesílaných dat do JSON formátu, který je v diplomové práci používán pro komunikaci s API. Problémy byly vyřešeny použitím populární knihovny *„Axios"*.

Zmíněná knihovna nevyužívá interně přímo funkci *„fetch"*, ale funkci *„XMLHttpRequest"*. Knihovna *„Axios"*, oproti zmíněné metodě *„fetch"*, obsahuje automatický převod odesílaných dat do textového formátu JSON, podporu pro nastavení aditivních HTTP hlaviček, nastavení konfigurace pro vysílání požadavků a mnoho dalšího.

Pro zjednodušení, byla naimplementována pomocná funkce ve formě hooku, jmenovitě hook „*useAxios"*. Použitím zmíněného hooku dojde k automatickému nastavení zmíněných hlaviček *„Content-Type"* a *„Accept"*.

Protože většina funkcionality na straně API vyžaduje oprávnění, je nutné předat serveru informaci, který přihlášený uživatel v klientské aplikaci vytváří dotazy pro manipulaci dat na straně serveru. Tato informace je uvedena v serveru vygenerovaném JWT tokenu, který je třeba předat jako aditivní HTTP hlavičku při vytváření dotazu posílaného do vzdálené API aplikace. Token je nastavován v rámci hlavičky *„Authorization"*. Tato hlavička je nastavována pouze v případech, kdy je JWT dostupný. Pokud token dostupný není, hlavička se nenastavuje. Tímto chováním je zabráněno odesláním hlavičky *„Authorization"* obsahující neplatný token ve formě prázdného řetězce.

Zdrojový kód 15 - Ukázka nastavení „headers" uvnitř hooku useAxios

```
1 export const setAxiosHeaders = (token?: string) => {
2   const headers: AxiosHeadersType = {
3     'Content-Type': 'application/json; charset=utf-8',
4     'Accept': 'application/json; charset=utf-8',
5   };
6   if (token) headers.Authorization = `Bearer ${token}`;
7   return headers;
8 };
9
10 export interface CustomAxiosResponse<T> extends AxiosResponse<T> {
11   detail?: string,
12 }
13
14 const useAxios = () => {
15   const { token } = useContext(AuthContext);
16
17   const post = <T>(url: string, data: string | FormData,
18                    config?: AxiosRequestConfig):
19     Promise<CustomAxiosResponse<T>> => {
20     return axios.post(baseUrl + url, data, {
21       ...config,
22       method: 'POST',
23       headers: setAxiosHeaders(token)
24     });
25   }
```

Zdroj: vlastní zpracování (2023)

4.5.3.2 Manažer barevného schéma

Do aplikace byla naimplementována možnost změny barevného schématu. Základními připravenými schématy jsou:

- Světlý barevný režim
- Tmavý barevný režim

Pro jednoduché nastavení režimu byla naimplementována hook funkce *„useColorMode"*, která generuje objekt umožňující přístup a manipulaci s daty uložených v kontextu *„ColorModeContext"*.

4.5.3.3 AuthHook

Pro jednouchou správu informací o přihlášeném uživateli byl do aplikace naimplementován hook *„AuthHook"*. Tato funkce generuje objekt, který umožňuje manipulovat s *„Auth"* kontextem. Objekt vygenerovaný hookem představuje interní API objekt, který interně přistupuje k přiřazenému provideru obsahující business logiku pro hookem používané operace. Zmíněný hook definuje objekt s následujícími vlastnostmi.

Tabulka 18 - Seznam vlastností objektu generovaného v "useAuth" hooku

Název vlastnosti	Popis vlastnosti
user	Objekt obsahující informace o přihlášeném uživateli.
signIn	Funkce umožňující uložit informace o uživateli a JWT.
signOut	Funkce, která odebere uložené informace o uživateli.

Zdroj: vlastní zpracování (2023)

Aby nedošlo ke ztrátě informací o přihlášeném uživateli při obnovení stránky, ukládají se informace o uživateli, včetně jeho JWT do lokálního uložiště. Toto lokální uložiště se nachází v prohlížeči na klientském zařízení, kde je aplikace spuštěna.

Zdrojový kód 16 - Ukázka metody pro uložení JWT a uživatelských informací

```
1   const [token, setToken] = useState<string | undefined>(getToken());
2   const [user, setUser] = useState<IPerson | undefined>(getUser());
3
4   const persistToken = (value?: string) => {
5     localStorage.setItem('token', value || '');
6     setToken(value);
7   };
8
9   const persistUser = (value?: IPerson) => {
10    localStorage.setItem('user', JSON.stringify(value));
11    setUser(value);
12  };
```

Zdroj: vlastní zpracování (2023)

4.5.4 Komponenty

Veškerý renderování obsah v knihovně React se skládá je komponentou, a to včetně celých stránek. Proto je důležité se zamyslet nad způsobem, jakým by měly být jednotlivé komponenty navrhovány a rozdělit komponenty do několika úrovní. V rámci diplomové práce tak vznikly tři základní úrovně komponent.

- Atomické komponenty
- Kontejnery
- Stránky

Atomické komponenty lze chápat jakožto základní stavební kameny UI. Tyto komponenty jsou ty nejmenší možné renderování prvky. Typicky mezi ně patří vstupní pole ve formulářích, tlačítka, nadpisy, ale i prázdné divy aplikující styly, které budou následně naplněny dalšími komponentami.

Hlavním účelem atomických komponent je, aby byly co nejvíce generické, a tak použitelné ideálně kdekoliv v aplikaci. Zároveň je tímto způsobem možné mít připravené elementy, které mají identické stylování napříč celou aplikací. Pokud se tak vznesl například požadavek na změnu barvy, či jiného parametru, pro textový vstup, je možné tuto změnu aplikovat pro celou aplikaci úpravou jediného souboru. Tímto způsobem je možné elegantně sjednotit grafické kaskádové styly jednotlivých elementů pro celou aplikaci, a to nejen těch interaktivních.

Atomické komponenty se následně skládají do takzvaných kontejnerů. V rámci realizace diplomové práce, byly kontejnery použity spíše pro tvorbu jednotných rozhraní pro administraci. Tyto rozhraní obsahují primárně boční navigační panel a horní lištu. Obsah stránky je předáván a renderování jakožto potomek kontejneru. Potencionálním dalším využitím by bylo zabalovat komplexní obsah stránek do separátních kontejneru, například při rozsáhlých formulářích. Tato praktika však nebyla v rámci použita, protože většina formulářů v diplomové práci je relativně jednoduchá. Mezi vytvořené kontejnery patří především:

- LoginContainer
- BaseContainer
- FilespaceContainer

Přihlašovací kontejner byl navržen pro aplikování kaskádových stylů pro stránky dostupné veřejnosti. Používá se převážně na přihlašovací stránce a stránce určené k registraci.

Základní kontejner (*BaseContainer*) přidává již zmíněnou horní lištu a boční navigační panel do všech stránek v administraci. Jediné místo, kde se nepoužívá je na stránce výpisu obsahu nahraných CSV souborů. Na této specifické stránce se používá zmenšená varianta základního kontejneru, pojmenovaného *FilespaceContainer*.

Poslední zmíněnou úrovní komponent jsou stránky. Uvnitř těchto komponent se tvoří struktura stránek určených pro finální zobrazení. Na takové komponenty se přímo odkazuje v rámci směrování. Uvnitř takových komponent se typicky tvoří stromové struktury komponent, obsahující i zmíněné kontejnery. Stránky je možné označit za nejvyšší možnou vrstvu, či úroveň komponent.

4.5.4.1 MUI

Protože atomických komponent může být veliké množství, byla do diplomové práce nainstalována komponentová knihovna MUI. Tato knihovna obsahuje velké množství již připravených komponent, které je možné použít, rozšiřovat či upravovat.

Většina komponent, použitých z knihovny MUI byla zaobalena do vlastních komponent, které upravují chování, či nastavují třídy CSS pro danou komponentu. Mezi takové komponenty se řadí například *LabeledInput*, která obsahuje rozšiřuje MUI

textový input o parametry knihovny *Informed*. Zmíněná knihovna se v aplikaci používá pro jednodušší kontrolu formulářů a jejich vstupů.

Příkladem použití MUI komponenty tak, jak byla navržena bez přidané vlastní úpravy je například komponenta *Button*.

4.5.4.2 Tailwind CSS

Většina komponent, které byly v průběhu vývoje aplikace navrženy, využívají knihovnu Tailwind CSS. Tato knihovna umožňuje zapisovat kaskádové styly pomocí již připravených CSS tříd. Protože jsou všechny třídy již připraveny a není třeba je definovat manuálně, byl vývoj komponent značně urychlen. Knihovna navíc umožňuje tvořit vlastní třídy, tvořit barevné palety a upravovat již existující CSS třídy, čímž umožňuje vysokou míru flexibility při tvorbě vlastního designu.

Hlavní nevýhodou, na kterou se při realizaci diplomové práce narazilo byla velmi slabá podpora pro přechody a animace. Knihovna umožňuje aplikovat základní přechody a animace, nicméně jelikož knihovna neobsahuje připravené CSS třídy, které by ovlivňovaly potomky elementu, je tvorba animací a přechodů velmi náročná. Pro takové případy je snadnější a mnohem rychlejší vytvořit a importovat CSS či SCSS soubory.

4.5.4.3 SCSS

Některé starší komponenty byly vytvořeny původně se styly napsanými v jazyce SCSS. Tento skriptovací jazyk snižuje komplexitu kaskádových stylů psaných za pomocích jazyka CSS. Od tvorby SCSS souborů se v průběhu vývoje opustilo, jelikož jejich jediná výhoda byla aplikace již zmíněných složitějších animací a přechodů, které ovšem nebyly prioritou při tvorbě UI.

4.5.5 Směrování

Pro směrování v rámci klientské aplikace byla použita knihovna *react-router-dom*. Tato knihovna umožňuje směrování pro SPA knihovny, jakou je například právě React. Stránkování v tomto případě není opravdové stránkování, ale dochází pouze k záměně obsahu webové stránky podle aktuální URL adresy. Díky tomu nedochází ani ke ztrátě

mezipaměti prohlížeče, protože aplikace není nikdy reálně obnovena a nedochází k znovunačtení webové stránky.

Pro manipulaci URL adres v prohlížeči, využívá knihovna API umožňující měnit historii prohlížeče pomocí Javascriptu. Aby tento koncept mohl fungovat, je třeba nadefinovat sady cest dostupné uvnitř aplikace a které komponenty (stránky) mají být vykreslované na těchto cestách. Tento seznam cest je definován uvnitř komponenty v souboru *Routes.tsx*.

Zdrojový kód 17 - Ukázka souboru Routes.tsx

```
1  const Paths = () => {
2    return (
3      <Routes>
4
5        <Route path="/" element={<Login/>}/>
6        <Route path="/registration" element={<Registration/>}/>
7
8        <Route path="/dashboard" element={
9          <ProtectedRoute>
10           <Dashboard/>
11         </ProtectedRoute>
12       }/>
13
14       <Route path="/records" element={
15         <ProtectedRoute role={{name: 'admin'} as IRole}>
16           <Records/>
17         </ProtectedRoute>
18       }/>
```

Zdroj: vlastní zpracování (2023)

Tato komponenta se následně registruje v souboru *App.tsx*, který slouží jako hlavní soubor definující celou kostru aplikace. V hlavním souboru se exportuje stejnojmenná komponenta, *App*, která se renderuje v souboru *index.tsx*. Tento souboru se používá jakožto vstupní bod aplikace, jehož účel je inicializovat knihovnu React a vytvořit kostru aplikace. Tato kostra je následně vykreslována v prohlížeči klientského stroje. Na ukázce zdrojového kódu níže je možné vidět zmíněnou funkci *App*.

Zdrojový kód 18 - Ukázka souboru App.tsx

```
 1 function App() {
 2   const queryClient = new QueryClient();
 3   const { mode } = useColorMode();
 4
 5   const darkTheme = createTheme({
 6     palette: {
 7       mode,
 8     },
 9   });
10
11   return (
12     <ThemeProvider theme={darkTheme}>
13       <QueryClientProvider client={queryClient}>
14         <div className="app">
15           <AuthProvider>
16             <BrowserRouter>
17               <SidebarProvider>
18                 <Paths />
19               </SidebarProvider>
20             </BrowserRouter>
21           </AuthProvider>
22         </div>
23       </QueryClientProvider>
24     </ThemeProvider>
25   );
26 }
```

Zdroj: vlastní zpracování (2023)

4.5.5.1 Zabezpečení cesty

Pro používání aplikace je nutné si vytvořit uživatelský účet a přihlásit se. Protože knihovna *react-router-dom* neobsahuje možnost řešení autorizace, bylo třeba tuto možnost doimplementovat. Tato nutnost dala ke vzniku komponentě pojmenované *ProtectedRoute*. Tato komponenta očekává dva vstupní parametry. Prvním je objekt role oprávnění, kterému je povoleno přesměrovat a zobrazit obsah žádané stránky. Druhým parametrem jsou takzvaní potomci (z anglického *children*). Tito potomci jsou komponenty, které jsou umístěny uvnitř jiné komponenty. V tomto je potomkem stránka, kterou je třeba vykreslit v případě úspěšné autorizace.

Zdrojový kód 19 - Ukázka komponenty ProtectedRoute

```
 1 const ProtectedRoute: FC<ProtectedRouteType> = ({ role, children }) => {
 2   const { user } = useAuth();
 3 
 4   // TODO: We want to check if two arrays have at least 1 role in common
 5   if (role && !user?.roles?.some(
 6     (x) => x.name.toLowerCase() === role.name.toLowerCase()
 7   )) return (<Navigate to="/" />);
 8 
 9   if (!user || !user.id) return (<Navigate to="/" />);
10   return (
11     <>
12       {children}
13     </>
14   );
15 };
16 
17 export default ProtectedRoute;
```

Zdroj: vlastní zpracování (2023)

Pokud nedojde k úspěšné autorizaci, je uživatel přesměrován na stránku s přihlašovacím formulářem.

Zdrojový kód 20 - Ukázka použití komponenty ProtectedRoute

```
 1       <Route path="/invitations" element={
 2         <ProtectedRoute>
 3           <Invitations/>
 4         </ProtectedRoute>
 5       }/>
 6 
 7       <Route path="/admin/teams"
 8         element={
 9           <ProtectedRoute role={{name: 'admin'} as IRole}>
10             <TeamsTable/>
11           </ProtectedRoute>
12         }/>
```

Zdroj: vlastní zpracování (2023)

4.5.6 Grafické zpracování

Klientská webová aplikace obsahuje dvě základní grafická rozhraní. První rozhraním je uživatelské rozhraní, které je dostupné všem přihlášeným uživatelům bez ohledu na roli oprávnění. V tomto rozhraní je možné zobrazit a operovat se záznamy vázanými striktně

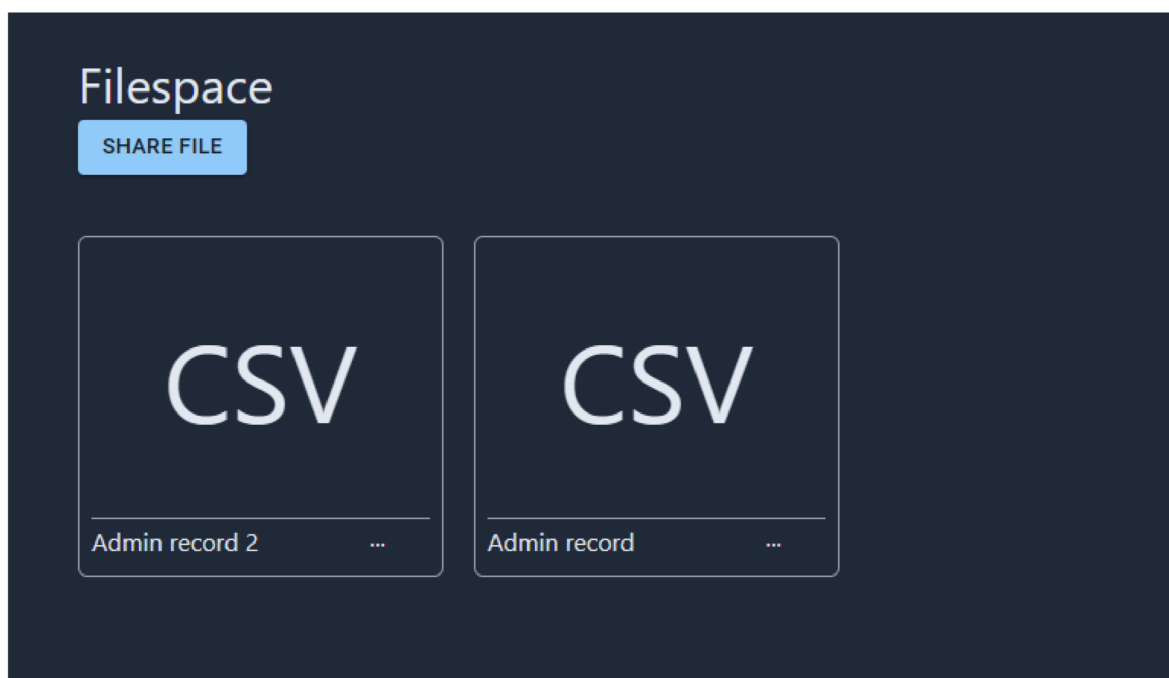
k účtu přihlášeného uživatele. Všechna data, které jsou v tomto rozhraní viditelné jsou omezeny API pouze na přihlášeného uživatele, bez ohledu na nastavení filtrování v rámci klientské aplikace. Díky tomu není možné, aby jakákoliv klientská aplikace, ať už se jedná o aplikaci vytvořenou v rámci diplomové práce, nebo jakoukoliv jinou klientskou aplikaci, není možné načíst data, ke kterým nemá uživatel oprávnění.

Druhým rozhraním je administrátorské rozhraní, které je dostupné pouze uživatelům s rolí administrátora. Je důležité podotknout, že obsah stránek určených pro administrátory je tvořen ze stejných komponent, jako jsou tvořeny stránky určené pro uživatele. V rámci tohoto rozhraní je možné zobrazit všechna aktivní data v databázi. Aktivními daty jsou zde myšleny všechny záznamy, které nejsou označené za smazané.

4.5.6.1 Desktopové zobrazení

Webová klientská aplikace byla vyvíjena primárně pro desktopové zobrazení, tedy pro zobrazení panely používané často v kombinaci se stolním PC. Toto rozhodnutí bylo aplikováno z důvodu charakteru očekávané práce s aplikací. Očekávaná práce je nahrávání, zobrazování a všeobecná operace nad tabulkami. Pro charakter takové práce se očekává použití stolního počítače, případně notebooku. Uživatelské rozhraní tak bylo nastylováno za použití flexboxů. Díky flexboxům bylo dosaženo základní responzivity pro rozlišení menší než FullHD (1920 x 1080 pixelů).

Obrázek 1 - Zobrazení seznamu nahraných souborů ve FullHD rozlišení



Zdroj: vlastní zpracování (2023)

4.5.7 Administrace

Rozhraní určené pro administrátory se skládá primárně ze pěti stránek. Tyto stránky následně odkazují na další stránky, obsahující většinou formuláře určené k vytváření nových záznamů v databázi. Stránky určené jen pro administrátory jsou popsány v tabulce níže.

Tabulka 19 - Seznam stránek dostupných jen pro administrátory

Stránka	Adresa stránky
Seznam přihlašovacích účtů	/users
Detail uživatelského účtu	/user/{id}
Tvorba uživatele	/users/add
Seznam nahraných souborů	/records
Seznam týmů	/admin/teams

Zdroj: vlastní zpracování (2023)

4.5.7.1 Seznam přihlašovacích účtů

Hlavní část obsahu seznamu přihlašovacích účtů, tvoří tabulka s výpisem všech uživatelů v databázi. Tato tabulka obsahuje všechny přihlašovací účty, ke kterým je následně možné

připojit provázané uživatelské účty. Každý jeden řádek v tabulce tedy je jeden přihlašovací účet, což je záznam z databázové tabulky *Login*. Tabulka obsahuje čtyři sloupce:

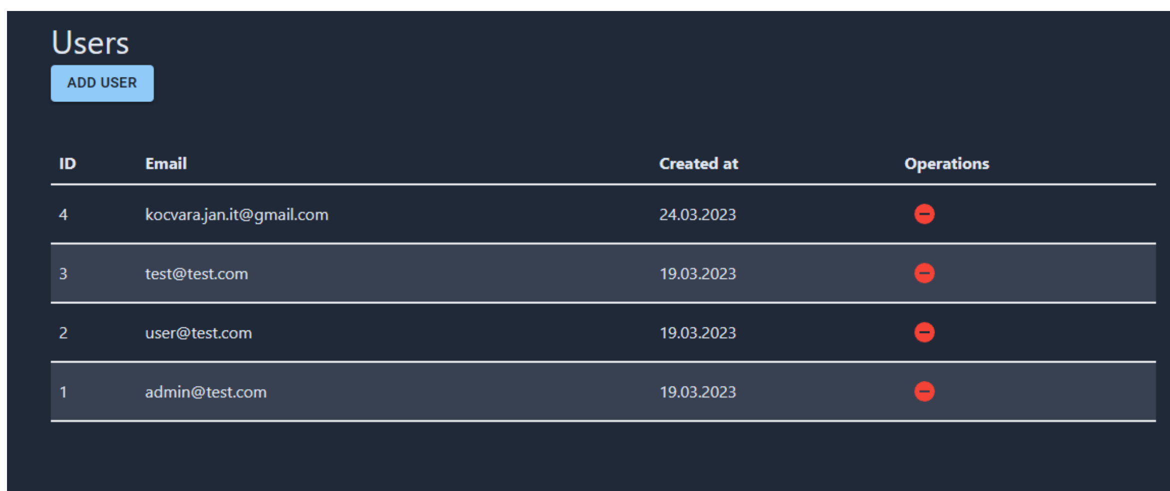
Tabulka 20 – Seznam sloupců tabulky uživatelských přihlašovacích účtů

Název sloupce	Popis
ID	Unikátní číselný identifikátor záznamu.
Email	Email použitý k přihlášení
Created_at	Časové razítko udávající čas vytvoření záznamu v databázi.
Operations	Sloupec určený pro možné operace se záznamem.

Zdroj: vlastní zpracování (2023)

Sloupec *Operations* obsahuje vždy jedinou možnost, a to možnost smazat uživatele. Je třeba mít na paměti, že veškeré odstraňování záznamů se provádí pouze pomocí označení záznamu za smazaný a je tedy možné smazané záznamy zpětně obnovit.

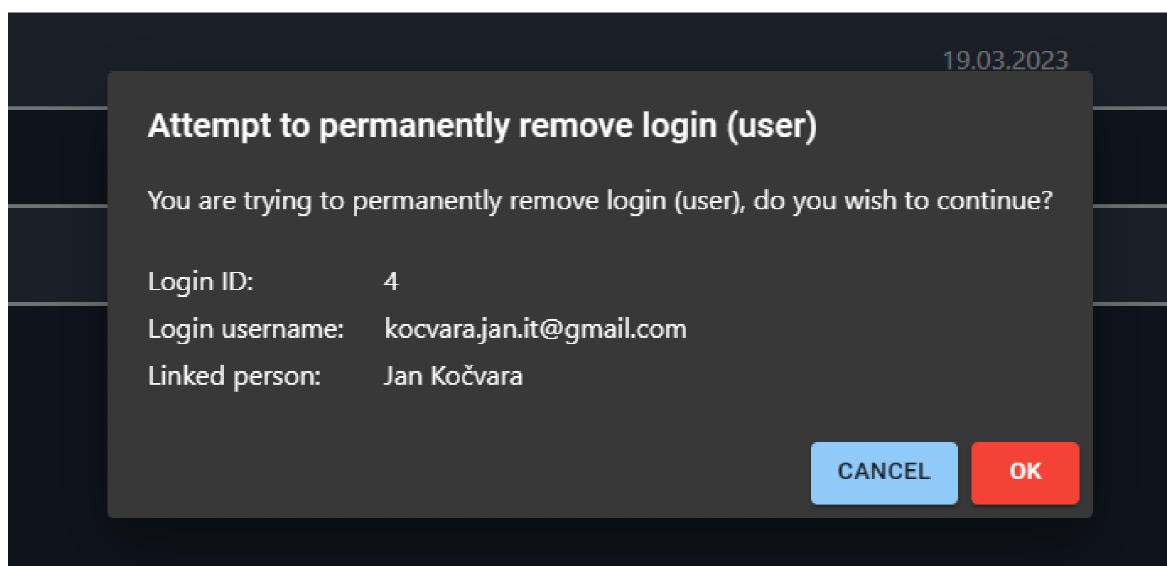
Obrázek 2 - Náhled tabulky obsahující seznam přihlašovacích účtů



Zdroj: vlastní zpracování (2023)

I přesto, že je možné obnovit smazané záznamy, bylo z důvodů zlepšení UX, přidáno dialogové okno, které umožní uživateli si svou operaci ještě jednou promyslet. Uvnitř dialogového okna jsou vidět opětovně informace o účtu, který se pokouší administrátor odebrat. V dialogovém okně je uvedena zpráva, tvrdící, že se jedná o permanentní smazání záznamu. Tato informace je v tento moment takto vypisována, ačkoliv není pravdivá, z důvodu ulehčení náročnosti realizace diplomové práce.

Obrázek 3 – Dialogové okno při smazání uživatelova přihlašovacího účtu



Zdroj: vlastní zpracování (2023)

Klientská aplikace v tento moment neumožňuje obnovit smazané záznamy, z tohoto důvodu je uživateli tvrzeno, že smazané záznamy není možné obnovit. Taková operace je naimplementována a připravena na straně serveru, nikoliv však na straně klienta.

Všechny smazané záznamy nejsou v tabulce viditelné. API automaticky odebírá veškeré záznamy označené za smazané a tyto záznamy tak nejsou v klientské aplikaci dostupné.

Při kliknutí na řádek, mimo sloupec *Operations*, je administrátor přesměrován na detailní zobrazení uživatele.

4.5.7.2 Detail uživatelského účtu

Pro zobrazení nahraných souborů vázaných přímo s uživatelem, případně týmů, které jsou s uživatelem vázány, je možné použít stránku pro detail uživatelského účtu. Na této stránce jsou zobrazeny navigační komponenta s třemi „taby" (do českého překladu přenést „tab" jako „karta"). Každá jedna karta zobrazuje jednu tabulku. Možné je tak zobrazit tři tabulky.

První tabulka umožňuje zobrazit seznam všech nahraných CSV souborů právě prohlíženým uživatelským účtem. Každý jeden záznam lze otevřít. V druhé tabulce je možné prohlížet seznam uživatelských týmů, které prohlížený účet vytvořil. Otevřením poslední karty je možné vykreslit tabulku obsahující seznam všech skupin, kterých je uživatel

součástí. Tato tabulka tedy zobrazuje týmy, které on sám nevytvořil, ale ke kterým se přidal přijetím pozvánky.

Data pro každou z tabulek jsou žádána ze serverové aplikace až v momentě potřeby, tedy až po rozkliknutí karty. Toto rozhodnutí je čistě z důvodu optimalizace. Nikdy tak není načítáno více dat, než je reálně v daný moment potřeba.

4.5.7.3 Tvorba uživatele

Další stránkou dostupnou pro administrátory je stránka obsahující formulář pro tvorbu nového uživatele. Tento formulář je oddělen na dvě části, a to z důvodu vytváření dvou databázových záznamů uvnitř jednoho formuláře. Jelikož jsou uživatelské účty rozdělené na dvě databázové entity, konkrétně na entitu *Login* a *Person*, je formulář rozdělen mezerou na tyto dvě entity. Toto rozdělení je převážně uskupením vstupních polí než grafického rázu. Vizuálně je formulář oddělen pouze mezerou.

Po odeslání formuláře se odesílá pouze jeden požadavek na API, ačkoliv dochází k modifikacím dvou databázových entit, a to i přestože API využívá *repository pattern*, kdy každý jeden endpoint je určen pro práci s jednou databázovou entitou. Při odeslání formuláře se odesílá HTTP metoda CREATE na endpoint *Login*. Při tomto požadavku se využívá praktiky *„upsert"*, kdy dochází k vytvoření nového záznamu do databázové tabulky *Login* a zároveň se k ní vytvoří a sváže nový záznam v tabulce *Person*.

4.5.7.4 Seznam nahraných souborů

Obdobně jako u seznamu uživatelů, i stránka *Records*, zobrazující seznam nahraných souborů, využívá jako hlavní komponentu tabulku. Data pro tabulku jsou načítána z API endpointu */records*. V tento moment, tabulka obsahuje všechny záznamy o nesmazaných nahraných CSV souborech a umožňuje tyto soubory odstranit, nebo je otevřít. Tabulka obsahuje pět sloupců:

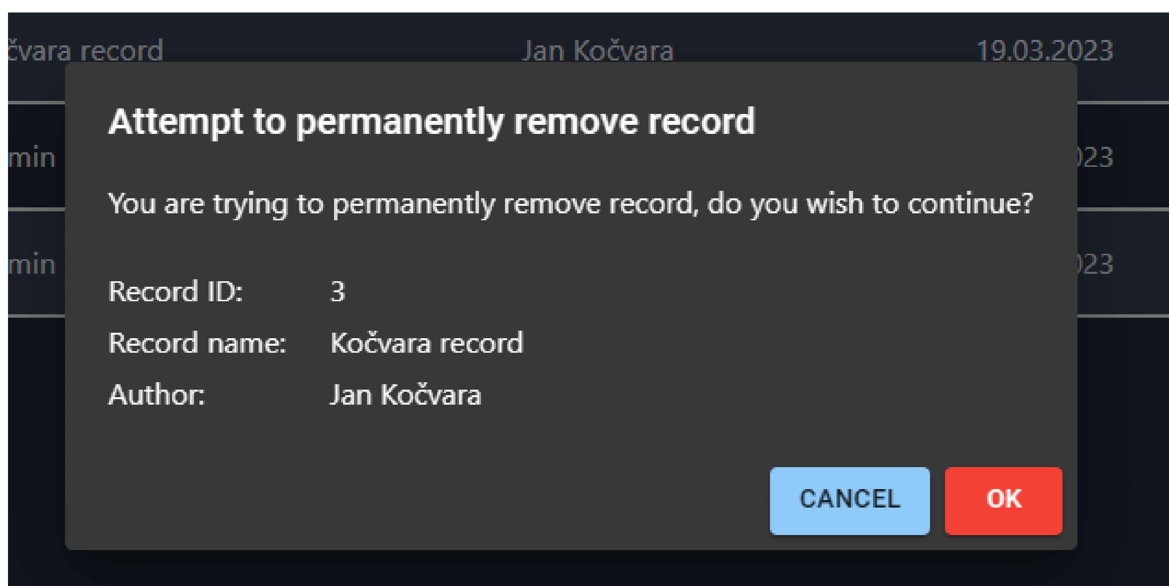
Tabulka 21 - Seznam sloupců tabulky nahraných souborů

Název sloupce	Popis
ID	Unikátní číselný identifikátor záznamu.
Name	Uživatelem nastavený název záznamu.
Author	Křestní jméno a příjmení uživatelského účtu, který záznam vytvořil.
Created_at	Časové razítko udávající čas vytvoření záznamu v databázi.
Operations	Sloupec určený pro možné operace se záznamem.

Zdroj: vlastní zpracování (2023)

Obdobně jako u možnosti smazání uživatelova přihlašovacího účtu, jak bylo popsáno v kapitole *4.5.7.1 Seznam přihlašovacích účtů*, i zde obsahuje sloupec *Operations* možnost pro odstranění záznamu. Tento záznam je při odstranění označen v databázi za smazaný a není dále předáván klientské aplikaci. Celý proces je podobný stejný, jako v kapitole *4.5.7.1 Seznam přihlašovacích účtů*.

Obrázek 4 - Dialogové okno při smazání nahraného souboru



Zdroj: vlastní zpracování (2023)

4.5.7.5 Seznam týmů

V rámci stránky zobrazující seznam týmů se vykresluje tabulka naplněná všemi existujícími, nesmazanými skupinami uživatelů v databázi. Zmíněná tabulka obsahuje následující sloupce:

Tabulka 22 - Seznam sloupců tabulky uživatelských týmů

Název sloupce	**Popis**
ID	Unikátní číselný identifikátor týmu.
Team name	Uživatelem nastavený název týmu
Author	Křestní jméno a příjmení uživatelského účtu, který tým vytvořil.
Created_at	Časové razítko udávající čas vytvoření týmu v databázi.

Zdroj: vlastní zpracování (2023)

Na této stránce není možné vytvářet nové týmy, nebo existující týmy odstraňovat. Při odstraňování týmů by bylo třeba vyřešit, jak se zachovat k členům týmů a všem sdíleným datům.

4.5.8 Uživatelské rozhraní

Zbytek naimplementovaných komponent je použit v základním uživatelském rozhraní. Toto rozhraní je dostupné jak administrátorům, tak uživatelům bez role administrátora. Celé rozhraní je limitováno pouze na data vůči přihlášenému uživateli. V takovém případě nesejde na přiřazené roli, při každém dotazu odeslaném API je zároveň i specifikováno, aby se data omezila na přihlášeného uživatele. V případě role administrátora je toto řízeno stranou klientské aplikace. V případě uživatele bez administrátorského oprávnění je tato limitace kontrolována samotným serverem, aby nedošlo k podvržení.

4.5.8.1 Souborový prostor

Velmi důležitou stránkou v klientské aplikaci je souborový prostor uživatele. Na této stránce je možné vidět všechny soubory, které přihlášený uživatel nahrál. Všechny soubory jsou zobrazené v malých boxech, vykreslovaných vedle sebe. Jelikož se jedná o stromovou strukturu, jsou ihned po otevření stránky vidět pouze nejvyšší soubory a adresáře. Toto zobrazení bylo navrženo právě takto z důvodu jednoduchého rozšíření o funkcionalitu *Drag&Drop*, kterou je tak možné v budoucnu naimplementovat pro možnost přesouvání souborů mezi adresáři.

Zmíněná stromová struktura je ovšem zcela virtuální a všechny nahrané a zobrazené soubory, včetně adresářů, jsou jen záznamy ukládané v databázi. Pro uživatele klientské aplikace by mělo však rozhraní budit dojem, že se jedná o reálně uložené soubory v adresářích na straně serveru.

Nicméně klientská aplikace neumí pracovat se zmíněnými adresáři. Virtuální adresáře jsou funkcionalita, která byla naimplementována na straně serveru a byly udělány všechny přípravy. Při implementaci prvotního řešení však došlo k problémům s možností sdílení adresářů a jejich obsahu a bylo třeba tuto funkcionalitu celou přeimplementovat. Kvůli této chybě nebylo dostatečné množství času řešení sdílení adresářů a tvorba adresářů opravit a doimplementovat.

4.5.8.2 Nahrání souboru

Další stránkou je jednoduchá stránka umožňující nahrávání souborů. Aktuálně je jen podporována přípona CSV. Formulář k nahrání se skládá ze tří vstupů. Prvním vstupem

je textové povinné pole pro pojmenování vytvářeného záznamu. Dalším textovým vstupem je nepovinné pole umožňující zadat popis záznamu.

Posledním vstupem je povinné pole pro vložení souboru s příponou CSV. Tento soubor je následně spolu s ostatními hodnotami odeslán serverové aplikaci, konkrétně na koncový bod *„record“* metodou *CREATE*.

Po úspěšném zpracování záznamu serverovou aplikací, je vrácen klientské aplikaci výsledek obsahující základní informace o vytvořeném záznamu spolu s kódem *201 Created.* Pokud klientská aplikace obdržela odpověď s kódem 201, je uživatel přesměrován na stránku souborového prostoru. V případě chyby je zobrazena chybová hláška a čeká se na další interakci uživatele.

4.5.8.3 Hluboké prohledávání CSV souborů

Na stránce souborového prostoru je možné filtrovat skrze všechny nahrané soubory přihlášeným uživatelem. Díky tomuto filtru je možné vyhledat soubory s žádaným obsahem. Při prohledávání je možné zadat libovolné množství parametrů specifikující prohledávání. Každý nastavený filtr umožňuje zadat tři hodnoty.

První hodnota je vždy vyžadovaná. Specifikuje obsah tabulkové buňky, která musí být v souborech obsažena. Druhým parametrem, který je dobrovolný, je možné určit ve kterém řádku by se měla buňka s hodnotou nacházet. Třetí parametr je rovněž dobrovolný a umožňuje upřesnit filtrování o sloupec, kde by se měla hledaná buňka nacházet.

Tato funkcionalita je užitečná pro nalezení všech nahraných souborů obsahující specifické hodnoty, díky tomu je možné i částečně porovnávat nahrané záznamy.

4.5.8.4 Sdílené soubory

Další stránkou v uživatelském rozhraní je stránka zobrazující sdílené soubory. Tato stránka využívá stejné UI komponenty jako stránka souborového prostoru, proto stránka vypadá téměř identicky. Liší se však obsahem, protože stránka obsahuje záznamy, které sám uživatel nevytvořil, ale byly mu nasdíleny v rámci uživatelských skupin. Uživatel nemá oprávnění tyto soubor odebrat.

4.5.8.5 Seznam týmů

Pro správu uživatelských týmů byla připravena stránka dostupná na adrese */teams*. Stránka se skládá ze dvou tabulek, které se zobrazují podle navigačního panelu složených z karet.

Menu obsahuje dvě karty, první kartou je *„My teams"*, která vykresluje tabulku obsahující všechny týmy vytvořené uživatelem. Druhá karta nese název *„Joined Teams"* a vykresluje tabulku obsahující všechny týmy, ke kterým se uživatel přidal přijetím pozvánky.

Součástí stránky je tlačítko umožňující vytvořit nový tým. Toto tlačítko vede na stránku vykreslovanou na adresse *„/teams/add"*.

4.5.8.6 Vytvoření nového týmu

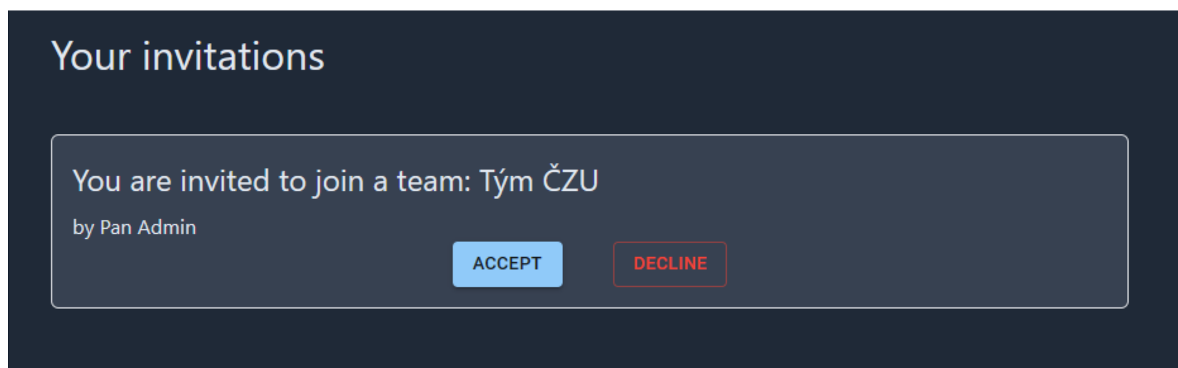
Nové týmy lze vytvářet pomocí stránky dostupné na adrese *„/teams/add"*. Tato jednoduchá stránka obsahuje formulář se třemi HTML elementy. Dva ze tří elementů jsou vstupní pole (z anglického *input*). První vstupní pole je určeno pro zadání názvu týmu, toto textové pole je povinné. Druhým polem je vstupní pole typu *select*. Toto pole interně odesílá HTTP dotazy serverové aplikace a umožňuje zobrazit a filtrovat skrze všechny uživatelské účty v databázi.

Posledním inputem je tlačítko typu *submit*, které zavolá javascriptovou funkci v pozadí formuláře. Tato funkce zmapuje a data zadané ve formuláři a odešle na koncový bod serverové aplikace.

4.5.8.7 Seznam aktivních pozvánek

Poslední zmíněnou stránkou je seznam aktivních pozvánek. Tato stránka umožňuje přijímat a odmítat pozvánky odeslané ostatními uživateli. Jednotlivé pozvánky se zobrazují v boxech pod sebou a obsahují dvě tlačítka, jedno pro přijetí, jedno pro odmítnutí.

Obrázek 5 - Náhled pozvánky



Zdroj: vlastní zpracování (2023)

5 Výsledky a diskuse

5.1 API

V průběhu vývoje byla serverová aplikace dostupná na adrese http://80.211.194.137:5122/. Z dlouhodobého hlediska není zaručeno, že tato adresa bude dostupná, avšak API je možné zprovoznit pomocí přiložených souborů na libovolném serveru.

Aplikace je plně funkční, nicméně pro její používání je třeba klienta, který umožňuje komunikovat s API pomocí protokolu HTTP, případně protokolu HTTPS. Použitý protokol závisí na konkrétní konfiguraci v souboru *appsettings.json*, případně změně implementace ve spouštěcím souboru *Program.cs*. Takovým klientem může být například program *Postman*, či *Insomnia*.

API funguje především jako prostředník mezi klientem a databází. Většina naimplementované funkcionality se váže přímo s prací s databázovým systémem. Řešení API obsahuje funkce pro nahrávání a zpracování CSV souborů, prohledávání nahraných záznamů, správu uživatelských účtů, sdílení nahraných souborů, správu uživatelských skupin i tvorbu virtuálních adresářů. Operace nad nahranými soubory je nezávislá na příponě nahraného souboru a je možné tak pracovat i například se soubory typu .XLS či .XLSX. Pro takové soubory však není v tento moment naimplementována funkce pro vyjmutí jejich obsahu.

Téměř všechny koncové body API vyžadují autorizaci. Té je možné dosáhnout odesláním platného JSON Web Tokenu, který aplikace vygenerovala a vydala uživateli na jeho vyžádání. V JWT jsou uloženy informace jako například ID uživatelského účtu, jeho role oprávnění, nebo jeho email. Po validaci JWT, se při každém dotazu odeslaném klientskou aplikací inicializuje databázový kontext, který v případě uživatelského účtu bez oprávnění administrátora limituje přístup k databázi.

5.1.1 Testování

Serverová aplikace byla průběžně testována při jejím vývoji, nicméně v tento moment neobsahuje žádné automatizované testy. Většina případných testů by byla integračních, protože hlavní funkcionalita a účel API je komunikace a práce s databází.

5.2 Klientská aplikace

Ve výsledné podobě klientské aplikace, byla naimplementována většina požadované funkcionality. Aplikace je plně funkční a dostupná online na webové adrese http://80.211.194.137/, popřípadě je možné zprovoznit aplikaci pomocí přiložených souborů.

V rámci zadání bylo dosaženo vytvoření dvou základních uživatelských rozhraní, konkrétně administrátorského rozhraní a pohledu určeného pro uživatele aplikace. Omezení oprávnění není z důvodu zabezpečení řešeno na úrovní klientské aplikace, ale na úrovni serverové implementace.

Uživateli je umožněno nahrávat a spravovat své CSV soubory, vytvářet týmy a zvát ostatní uživatele systému do svých týmů. V případě týmu je uživateli umožněno odmítat a přijímat pozvánky. Pro ujištění, že nedošlo k mylnému odebrání záznamu, byla k možnosti odebrání záznamů přidána kontrola, žádající o potvrzení smazání.

Další funkcionalitou je prohledávání a filtrování nahraných souborů. Toho je možné dosáhnout použitím prohledávače nahraných záznamů, jehož logika je naimplementována na straně API. Díky tomuto filtrování, je možné zobrazit nahrané soubory, které obsahují hledaný seznam hodnot. Tyto hodnoty lze specifikovat jejich lokací, konkrétně označením sloupce a řádku, kde se má hodnota vyskytovat, tato specifikace je však není vyžadována a byla přidána pouze k umožnění zpřísnění vyhledávacích parametrů. Nahrané soubory není v tento moment možné upravovat. To je z důvodu použití knihovny, která se ukázala jako neoptimální pro editaci tabulkových záznamů. Tato knihovna umožňuje soubory, jak zobrazit, tak i upravit, avšak při všechny upravené hodnoty v záznamu jsou v nepoužitelném formátu. Knihovna při editování záznamu ukládá konkrétní buňky tabulek v podobě HTML elementů. Ačkoliv je možné následně všechny HTML elementy projít a zpracovat do použitelné podoby, například do JSON formátu, jedná se o velké množství kódu, který by velmi zpomaloval aplikaci.

Zmíněná knihovna pro zobrazení a případnou práci s nahranými tabulkovými soubory je navíc znatelně zpomalená při zobrazování rozsáhlejších souborů. Tento jev byl zpozorován při zobrazení CSV souboru obsahující zhruba 120 sloupců a 50 řádků. Pro účely diplomové práce by bylo ideální vytvořit vlastní knihovnu, která by umožňovala správu obsahu tabulkových souborů.

Pro zjednodušení práce se soubory v aplikaci, byly navrženy skupiny určené pro sdružení nahraných souborů. Tyto skupiny mají stromovou strukturu a lze tak zanořovat jednotlivé skupiny do dalších skupin. Efektivně tak lze tyto skupiny vizuálně reprezentovat jako souborové adresáře, ačkoliv se jedná pouze o záznamy uložené v databázi.

Při implementování této funkce však nastal nečekaný problém s UI řešením. Pro příjemné používání této funkce a zajištění tak dobrého UX by bylo třeba navrhnout komponenty obsahující funkce známé jako *drag and drop*. Takové komponenty by bylo možné „přetáhnout" pomocí myši a zanořit tak záznamy do sebe. Bohužel, návrh a implementace takové komponenty by byla velmi komplexní a časově náročná a tato funkcionalita nebyla do klientské aplikace naimplementována, ačkoliv API ji zcela podporuje.

Aplikace také umožňuje vytvářet skupiny uživatelů, které jsou označeny v rámci UI za „týmy". Tyto skupiny byly navrženy a naimplementovány z důvodu jednoduššího sdílení nahraných souborů mezi uživateli. Při vytvoření skupiny je umožněno uživateli přidat další uživatele, tito uživatelé následně obdrží pozvánky.

Aplikace umožňuje zobrazit soubory a virtuální složky, které jsou uživateli nasdíleny z různých skupin. Bohužel, platforma však neumožňuje reálně sdílet soubory mezi uživateli. Tato funkcionalita je připravena a naimplementována na straně serveru, ovšem z důvodu chybného návrhu databázového řešení a implementace této funkcionality, bylo třeba tuto funkci odebrat a znovu navrhnout. Ačkoliv došlo k opravení návrhu a připravení zmíněné funkce na straně API, klientská aplikace na tuto funkcionalitu není připravena. Klientská aplikace tak nenabízí možnost sdílení souborů, či virtuálních složek, ačkoliv je zcela připravena na jejich zobrazení.

5.2.1 Testování

Aplikace byla průběžně testována při jejím vývoji a byla tak nalezena a opravena většina chyb. Automatizované testování bohužel nebylo do aplikace naimplementováno. Možná implementace testů pro klientskou aplikaci by šla rozdělit na dvě části, a to na testování grafické části a implementace funkcí.

V případě testování funkcionality, aplikace obsahuje relativně malé množství kódu k testování, mezi takové funkce lze řadit například mapovací funkce, které se používají při tvoření těla HTTP dotazu odesílaného serverové aplikaci, nebo funkce určené

pro zpracování dat pro zobrazení CSV souborů. K takovému testování je například možné použít knihovnu *Jest*.

Zajímavější, mnohem rozsáhlejší a také komplexnější testování by bylo pro grafické prvky. Pro takové testování, lze použít například knihovnu *Selenium*. V tomto případě by bylo třeba označit každou komponentu HTML třídou a následně pomocí knihovny Selenium porovnávat „*snapshoty*“ s aktuálně renderovanou komponentou. Díky takovému porovnávání lze alespoň částečně automatizovat testování grafického zobrazení komponent určených k renderování.

Důvodem neimplementování těchto testů by byla nutná alokace velkého množství času neúměrnému k ostatním částem zadání práce. Testování mapovacích a dalších funkcí nebylo naimplementováno naopak z důvodu použití jazyka TypeScript, který snižuje množství chyb přidaným statickým typováním a kompilací do JavaScriptu. Klientská aplikace navíc neobsahuje žádnou *business logiku*, díky čemuž v aktuální podobě obsahuje jen malé množství funkcí, které svou podstatou jsou spíše mapovací a možnost výskytu chyby tak lze označit za zanedbatelnou.

6 Závěr

V teoretické části práce byly představeny hlavní technologie použité pro návrh a realizace řešení. Nejdříve byl představen model klient-server, který byl následně aplikován. Tímto bylo zajištěno oddělení podnikové logiky systému od aplikace zajišťující vykreslování dat. Dalším důvodem použití zmíněného modelu, byla příprava pro možnost rozšířit navrženou aplikaci z webového prostředí na další platformy, jako jsou například mobilní telefony, či desktopové aplikace.

Další představenou technologií, v teoretické části, byl framework .NET Core, který byl použit k realizaci serverové části aplikace. Tato aplikace, také jinak nazvaná API, obsahuje již zmíněnou podnikovou logiku. Hlavním účelem API je být serverovou částí v již popsané architektuře a umožnit tak sdílení dat uložených v databázi potencionálně napříč různými platformami. Api se využívá především ke čtení dat a manipulaci dat. Serverovou část aplikace lze zároveň použít čistě pro autentizaci, protože umožňuje vytvářet JSON Web Tokeny (zkráceně JWT), které i následně ověřuje v rámci procesu autorizace.

Pro databázové řešení, byl použit databázový systém Microsoft SQL Server. Tento systém byl spouštěn v podobě kontejneru pomocí softwaru Docker. Databázové řešení bylo navrženo v třetí normální formě pro relační databáze. Pro práci s MSSQL byla do serverové aplikace naimplementována knihovna Entity Framework Core.

Poslední dvě témata představená v teoretické části práce byla knihovna React a programovací jazyk TypeScript. Obě technologie byly použity pro vývoj klientské webové aplikace, která slouží jako grafické rozhraní pro práci s databázovým systémem.

Praktická část práce byla rozdělena na tři hlavní části. Jmenovitě na část o návrhu databázového řešení, realizace serverové REST API a tvorbě webové klientské aplikace v knihovně React.

V části návrhu databázového řešení byla popsána struktura databáze a návrh databázových entit, včetně vazeb mezi entitami. U některých tabulek byly taktéž zmíněny návrhy k vylepšení a optimalizaci.

V kapitole REST API byl představen návrh řešení serverové aplikace. Dále zde byla nastíněna její implementace, včetně navržené struktury a návrhových vzorů. Aplikace byla vyvinuta v jazyce C# za použití frameworku .Net Core. Při realizaci bylo použito OOP paradigma s důrazem na velkou míru abstrakce, která byla implementována pomocí

repozitářů. Aplikováním repozitářového vzoru bylo dosaženo oddělení implementace databázových operací a přístupových bodů do dvou oddělených vrstev. Díky tomuto rozdělení je možné rozšiřovat aplikaci o další vrstvy. Jedním z možných rozšíření je přidat cache vrstvu mezi přístupové body API a vrstvu databázového přístupu.

V kapitole o klientské aplikaci byla popsána struktura komponent v projektu, způsob komunikace se serverovou aplikací a jak jsou řešeny oprávnění na úrovni klientské aplikace. V kapitole jsou dále představené použité komponentové knihovny třetích stran, návrh a důvod použití modálních oken a interní pomocné funkce zvané hooky. V rámci práce byla také popsána naimplementovaná míra podpory mobilních zařízení.

7 Seznam použitých zdrojů

[1] INGALLS, Sam. Client-Server Model | A Guide to Client-Server Architecture. *ServerWatch* [online]. 17. listopad 2021 [vid. 2023-03-27]. Dostupné z: https://www.serverwatch.com/guides/client-server-model/

[2] *Client-Server Model - GeeksforGeeks* [online]. [vid. 2023-03-27]. Dostupné z: https://www.geeksforgeeks.org/client-server-model/

[3] How does the client server model work? *IONOS Digital Guide* [online]. 31. leden 2023 [vid. 2023-03-27]. Dostupné z: https://www.ionos.com/digitalguide/server/know-how/client-server-model/

[4] What is Client Server Architecture? - Differences, Types, Example. *Intellipaat Blog* [online]. 24. září 2021 [vid. 2023-03-10]. Dostupné z: https://intellipaat.com/blog/what-is-client-server-architecture/

[5] SUOMELA, Jukka. Answer to „What is the origin of the client server model?" In: *Computer Science Stack Exchange* [online]. 1. prosinec 2012 [vid. 2023-03-28]. Dostupné z: https://cs.stackexchange.com/a/7060

[6] *History Of The Client Server Architecture* [online]. [vid. 2023-03-27]. Dostupné z: https://www.ukessays.com/essays/information-technology/history-of-the-client-server-architecture-information-technology-essay.php

[7] What is Client-Server Architecture? Everything You Should Know | Simplilearn. *Simplilearn.com* [online]. 22. duben 2022 [vid. 2023-03-10]. Dostupné z: https://www.simplilearn.com/what-is-client-server-architecture-article

[8] *Brief History-Computer Museum* [online]. [vid. 2023-03-28]. Dostupné z: https://museum.ipsj.or.jp/en/computer/unix/history.html

[9] *5 Advantages and Disadvantages of Client Server Network | Drawbacks & Benefits of Client Server Network* [online]. [vid. 2023-03-27]. Dostupné z: https://www.hitechwhizz.com/2020/11/5-advantages-and-disadvantages-drawbacks-benefits-of-client-server-network.html

[10] Client Server Architecture Advantages & Disadvantages. *Techwalla* [online]. [vid. 2023-03-28]. Dostupné z: https://www.techwalla.com/articles/client-server-architecture-advantages-disadvantages

[11] Comparison of „peer-to-peer" vs „client-server" Network Models. *Networks Training* [online]. [vid. 2023-03-27]. Dostupné z: https://www.networkstraining.com/peer-to-peer-vs-client-server-network/

[12] *What is Client-Server Networking? Definition, Advantages, and Disadvantages - sunnyvalley.io* [online]. [vid. 2023-03-28]. Dostupné z: https://www.sunnyvalley.io/docs/network-basics/what-is-client-server-network

[13] *Overview of APIs: Value, Benefits, and How to Avoid Vulnerabilities* [online]. 21. září 2021 [vid. 2023-03-28]. Dostupné z: https://www.mossadams.com/articles/2021/09/how-an-api-works

[14] GEWARREN. *.NET (and .NET Core) - introduction and overview* [online]. 15. březen 2023 [vid. 2023-03-27]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/introduction

[15] CHAND, Mahesh. *What Is .NET Core?* [online]. [vid. 2023-03-27]. Dostupné z: https://www.c-sharpcorner.com/article/what-is-dot-net-core/

[16] SINGH, Jatinderbir. *Why We Should And Should Not Use .NET Core* [online]. [vid. 2023-03-28]. Dostupné z: https://www.c-sharpcorner.com/blogs/why-should-and-should-not-we-use-net-core

[17] Why choose the .NET developer platform? *Microsoft* [online]. [vid. 2023-03-28]. Dostupné z: https://dotnet.microsoft.com/en-us/platform/why-choose-dotnet

[18] Episode 1 - A Brief History of .NET Core. *The .NET Core Podcast* [online]. 24. srpen 2018 [vid. 2023-03-27]. Dostupné z: https://dotnetcore.show/episode-1-a-brief-history-of-net-core/

[19] TAYLOR, Jamie. .NET Core History. *A Journey In .NET Core* [online]. 23. únor 2018 [vid. 2023-03-28]. Dostupné z: https://dotnetcore.gaprogman.com/2018/02/23/net-core-history/

[20] SHAREPOINTCAFE. Evolution of .NET Framework to .NET Core. *SharePointCafe.Net* [online]. 19. listopad 2021 [vid. 2023-03-28]. Dostupné z: https://www.sharepointcafe.net/2021/11/evolution-of-dot-net.html

[21] . *NET Core Version History | Release History of .NET Core - Dotnet Stuff* [online]. 20. květen 2022 [vid. 2023-03-28]. Dostupné z: https://www.dotnetstuffs.com/net-core-version-history/

[22] BUGNION, Laurent. A Brief History of .NET Standard. *Xamarin Blog* [online]. 16. duben 2018 [vid. 2023-03-28]. Dostupné z: https://devblogs.microsoft.com/xamarin/history-dot-net-standard/

[23] RICK-ANDERSON. *Overview of ASP.NET Core* [online]. 15. listopad 2022 [vid. 2023-03-27]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core

[24] PROFEXORGEEK. *What is Xamarin? - Xamarin* [online]. 20. září 2022 [vid. 2023-03-27]. Dostupné z: https://learn.microsoft.com/en-us/xamarin/get-started/what-is-xamarin

[25] STRIDELY. Why you should develop your Mobile App with Xamarin. *Stridely Solutions* [online]. 27. červenec 2018 [vid. 2023-03-27]. Dostupné

z: https://www.stridelysolutions.com/insights/blog/why-you-should-develop-your-mobile-app-with-xamarin/

[26] PROFEXORGEEK. *What is Xamarin.Forms? - Xamarin* [online]. 8. červenec 2021 [vid. 2023-03-27]. Dostupné z: https://learn.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms

[27] DAVIDBRITCH. *What is .NET MAUI? - .NET MAUI* [online]. 30. leden 2023 [vid. 2023-03-28]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui

[28] PILUTA, Roman. What is .NET MAUI and Does it Mean the End of Xamarin Forms – NIX United. *NIX United – Custom Software Development Company in US* [online]. 21. prosinec 2022 [vid. 2023-03-28]. Dostupné z: https://nix-united.com/blog/can-maui-fully-replace-xamarin-forms-a-deep-dive-into-net-maui/

[29] Blazor | Build client web apps with C# | .NET. *Microsoft* [online]. [vid. 2023-03-10]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

[30] HEDDINGS, Anthony. What Is Microsoft's Blazor Web Framework, and Should You Use It? *How-To Geek* [online]. 5. srpen 2021 [vid. 2023-03-28]. Dostupné z: https://www.howtogeek.com/devops/what-is-microsofts-blazor-web-framework-and-should-you-use-it/

[31] Blazor University - What is Blazor? *Blazor University* [online]. [vid. 2023-03-28]. Dostupné z: https://blazor-university.com/overview/what-is-blazor

[32] EASTMAN, David. No More JavaScript: How Microsoft Blazor Uses WebAssembly. *The New Stack* [online]. 27. březen 2023 [vid. 2023-03-28]. Dostupné z: https://thenewstack.io/no-more-javascript-how-microsoft-blazor-uses-webassembly/

[33] The Most Complete ASP.NET Core Components | Telerik UI for ASP.NET Core. *Telerik.com* [online]. [vid. 2023-03-28]. Dostupné z: https://www.telerik.com/aspnet-core-ui

[34] *Free Blazor Components | 70+ controls by Radzen* [online]. [vid. 2023-03-28]. Dostupné z: https://blazor.radzen.com/

[35] *Ant Design of Blazor - Ant Design of Blazor* [online]. [vid. 2023-03-28]. Dostupné z: https://antblazor.com/en-US/docs/introduce

[36] *React – A JavaScript library for building user interfaces* [online]. [vid. 2023-03-10]. Dostupné z: https://reactjs.org/

[37] *React Native · Learn once, write anywhere* [online]. [vid. 2023-03-10]. Dostupné z: https://reactnative.dev/

[38] JUDE, Marvin. ReactJS, single-page applications(SPA) for the rest of us. *Medium* [online]. 3. duben 2020 [vid. 2023-03-10]. Dostupné z: https://medium.com/@marvinjudehk/reactjs-single-page-applications-spa-for-the-rest-of-us-cd210b107051

[39] HÁMORI, Ferenc. The History of React.js on a Timeline. *RisingStack Engineering* [online]. 4. duben 2018 [vid. 2023-03-10]. Dostupné z: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/

[40] *Design Principles – React* [online]. [vid. 2023-03-11]. Dostupné z: https://reactjs.org/docs/design-principles.html

[41] *React Components* [online]. [vid. 2023-03-11]. Dostupné z: https://www.w3schools.com/react/react_components.asp

[42] React Basics: What Is a React Component & How to Create One. *Telerik Blogs* [online]. 9. březen 2023 [vid. 2023-03-10]. Dostupné z: https://www.telerik.com/blogs/react-basics-what-component-how-create-one

[43] Atomic Design and ReactJS. *Atomic Design and ReactJS* [online]. [vid. 2023-03-11]. Dostupné z: https://danilowoz/blog/atomic-design-with-react/

[44] *Virtual DOM and Internals – React* [online]. [vid. 2023-03-11]. Dostupné z: https://reactjs.org/docs/faq-internals.html

[45] Introduction to React : Real DOM & Virtual DOM. *DEV Community* [online]. 31. srpen 2021 [vid. 2023-03-11]. Dostupné z: https://dev.to/swarnaliroy94/introduction-to-react-real-dom-virtual-dom-363

[46] *Introducing JSX – React* [online]. [vid. 2023-03-11]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html

[47] *JSX In Depth – React* [online]. [vid. 2023-03-11]. Dostupné z: https://reactjs.org/docs/jsx-in-depth.html

[48] *React JSX* [online]. [vid. 2023-03-11]. Dostupné z: https://www.w3schools.com/react/react_jsx.asp

[49] JSX in React – Explained with Examples. *freeCodeCamp.org* [online]. 1. únor 2021 [vid. 2023-03-11]. Dostupné z: https://www.freecodecamp.org/news/jsx-in-react-introduction/

[50] *Most Popular React Tech Stack in 2022 (Based on Data)* [online]. 10. říjen 2022 [vid. 2023-03-28]. Dostupné z: https://profy.dev/article/react-tech-stack

[51] *What is React Router: A Complete Guide [Updated 2023]* [online]. [vid. 2023-03-28]. Dostupné z: https://www.knowledgehut.com/blog/web-development/react-router

[52] *React Redux | React Redux* [online]. [vid. 2023-03-10]. Dostupné z: https://react-redux.js.org/

[53] *Redux Essentials, Part 1: Redux Overview and Concepts | Redux* [online]. 6. březen 2023 [vid. 2023-03-11]. Dostupné z: https://redux.js.org/tutorials/essentials/part-1-overview-concepts

[54] What is TypeScript? *TypeScript Tutorial* [online]. [vid. 2023-03-23]. Dostupné z: https://www.typescripttutorial.net/typescript-tutorial/what-is-typescript/

[55] TypeScript. *Cleverism* [online]. 11. srpen 2016 [vid. 2023-03-23]. Dostupné z: https://www.cleverism.com/skills-and-tools/typescript/

[56] *Benefits of Using TypeScript | DigitalOcean* [online]. [vid. 2023-03-28]. Dostupné z: https://www.digitalocean.com/community/tutorials/typescript-typescript-benefits

[57] REDDY, Jay Krishna. 5 Essential Benefits Of Using TypeScript. *Medium* [online]. 27. červen 2021 [vid. 2023-03-28]. Dostupné z: https://javascript.plainenglish.io/top-5-essential-benefits-of-using-typescript-e55aa52038b4

[58] *How To Use Enums in TypeScript | DigitalOcean* [online]. [vid. 2023-03-28]. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-use-enums-in-typescript

[59] JORDAN, Mark. Typescript (or: How I learned to stop worrying and love web development). *Ingeniously Simple* [online]. 17. září 2018 [vid. 2023-03-28]. Dostupné z: https://medium.com/ingeniouslysimple/typescript-or-how-i-learned-to-stop-worrying-and-love-web-development-deebe44c8794

[60] *Why C# goes well with TypeScript* [online]. 21. prosinec 2021 [vid. 2023-03-28]. Dostupné z: https://nate.org/csharp-and-typescript

[61] *The Complete History of JavaScript, TypeScript, and Node.js* [online]. [vid. 2023-03-23]. Dostupné z: https://www.itmagination.com/blog/the-complete-history-of-javascript-typescript-and-node-js

[62] TypeScript Versions | List of Complete Release Hstory for TypeScript. *EDUCBA* [online]. 26. prosinec 2019 [vid. 2023-03-28]. Dostupné z: https://www.educba.com/typescript-versions/

[63] *An Introduction to TypeScript: Static Typing for the Web — SitePoint* [online]. 7. březen 2018 [vid. 2023-03-28]. Dostupné z: https://www.sitepoint.com/introduction-to-typescript/

[64] Dynamic Static Typing In TypeScript. *Smashing Magazine* [online]. [vid. 2023-03-28]. Dostupné z: https://www.smashingmagazine.com/2021/01/dynamic-static-typing-typescript/

[65] HIWARALE, Uday. Understanding TypeScript's "Compilation Process" & the anatomy of "tsconfig.json" file. *JsPoint* [online]. 1. září 2020 [vid. 2023-03-28]. Dostupné z: https://medium.com/jspoint/typescript-compilation-the-typescript-compiler-4cb15f7244bc

[66] CIANCI, Lewis. Comparing the best TypeScript IDEs. *LogRocket Blog* [online]. 16. září 2022 [vid. 2023-03-28]. Dostupné z: https://blog.logrocket.com/comparing-best-typescript-ides/

[67] *Documentation - Everyday Types* [online]. [vid. 2023-03-28]. Dostupné z: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

[68] *Documentation - Creating Types from Types* [online]. [vid. 2023-03-28]. Dostupné z: https://www.typescriptlang.org/docs/handbook/2/types-from-types.html

[69] RAJGOR, Karan. Basics Of Object Oriented Programming(OOP) With Typescript. *The Code Hubs* [online]. 19. květen 2022 [vid. 2023-03-28]. Dostupné z: https://www.thecodehubs.com/basics-of-object-oriented-programmingoop-with-typescript/

[70] *Is TypeScript OOP? Detailed Explanation with Examples* [online]. 16. srpen 2022 [vid. 2023-03-28]. Dostupné z: https://www.numeric-quest.com/is-typescript-oop/

[71] Object Oriented Programming with TypeScript. *DEV Community* [online]. 5. červen 2021 [vid. 2023-03-28]. Dostupné z: https://dev.to/kevin_odongo35/object-oriented-programming-with-typescript-574o

[72] AUTH0.COM. *JWT.IO - JSON Web Tokens Introduction* [online]. [vid. 2023-03-31]. Dostupné z: http://jwt.io/

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

Obrázek 1 - Zobrazení seznamu nahraných souborů ve FullHD rozlišení 72
Obrázek 2 - Náhled tabulky obsahující seznam přihlašovacích účtů..................................73
Obrázek 3 – Dialogové okno při smazání uživatelova přihlašovacího účtu 74
Obrázek 4 - Dialogové okno při smazání nahraného souboru 76
Obrázek 5 - Náhled pozvánky 81

8.2 Seznam ukázek zdrojového kódu

Zdrojový kód 1 - Ukázka generického rozhraní IBaseRepository......................................41
Zdrojový kód 2 - Ukázka dědění generického IBaseRepository rozhraní a definování aditivní metody...41
Zdrojový kód 3 - Ukázka registrování repozitářů jako služeb 42
Zdrojový kód 4 - Ukázka GET metody implementované v PersonController řadiči 44
Zdrojový kód 5 - Ukázka rozhraní IQueryableRequest modelu 48
Zdrojový kód 6 - Rozhraní IQueryableRepository 49
Zdrojový kód 7 - Ukázka dědění IQueryableRepository v IPersonRepository 49
Zdrojový kód 8 - Pomocná metoda ApplyQuery pro aplikování dotazovacích modelů 50
Zdrojový kód 9 - Ukázka definování relací uvnitř funkce "OnModelCreating" 52
Zdrojový kód 10- Ukázka rozšířené "OnModelCreating" funkce o globální filtry 53
Zdrojový kód 11 - Ukázka deaktivování globálních query filtrů..54
Zdrojový kód 12 - Implementace metody GenerateToken v JWTManger službě...............56
Zdrojový kód 13 - Funkce zvolení správného databázového kontextu 58
Zdrojový kód 14 – Ukázka pomocné třída IFileProviderFactory 59
Zdrojový kód 15 - Ukázka nastavení „headers“ uvnitř hooku useAxios 63
Zdrojový kód 16 - Ukázka metody pro uložení JWT a uživatelských informací 65
Zdrojový kód 17 - Ukázka souboru Routes.tsx...68
Zdrojový kód 18 - Ukázka souboru App.tsx 69
Zdrojový kód 19 - Ukázka komponenty ProtectedRoute...70

Zdrojový kód 20 - Ukázka použití komponenty ProtectedRoute 70

8.3 Seznam tabulek

Tabulka 1 - Struktura databázové entity Person 31
Tabulka 2 - Struktura databázové entity Login 32
Tabulka 3 - Struktura databázové entity Role 32
Tabulka 4 - Struktura vazební tabulky PersonRole 33
Tabulka 5 - Struktura databázové entity Contact... 33
Tabulka 6 - Struktura databázové entity ContactType 34
Tabulka 7 - Struktura databázové entity Record 35
Tabulka 8 - Struktura databázové entity RecordData.. 35
Tabulka 9 - Struktura databázové entity RecordGroup 36
Tabulka 10 - Struktura databázové entity PersonGroup... 37
Tabulka 11 - Struktura databázové entity PersonGroupRelations...................................... 38
Tabulka 12 - Struktura databázové entity PersonGroupRecord 38
Tabulka 13 - Struktura databázové entity PersonGroupRecordGroup 39
Tabulka 14 - Seznam základních kontrolerů 43
Tabulka 15 - Seznam modelů databázových entit 46
Tabulka 16 - Seznam kontextů klientské aplikace... 61
Tabulka 17 - Seznam nutných HTTP hlaviček pro komunikaci mezi klientem a API........ 62
Tabulka 18 - Seznam vlastností objektu generovaného v "useAuth" hooku 64
Tabulka 19 - Seznam stránek dostupných jen pro administrátory...................................... 72
Tabulka 20 – Seznam sloupců tabulky uživatelských přihlašovacích účtů 73
Tabulka 21 - Seznam sloupců tabulky nahraných souborů 76
Tabulka 22 - Seznam sloupců tabulky uživatelských týmů.. 77

8.4 Seznam použitých zkratek

Zkratka	Celý název
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
C#	C Sharp
DB	Database
DDD	Domain-Driven Design
DOM	Document Object Model
DTO	Data Transfer Object
EF	Entity Framework
EF Core	Entity Framework Core
FK	Foreign Key
GO	Go Lang
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
IP	Internet Protocol
IT	Information Technology
JQS	JavaScript XML
JSON	JavaScript Object Notation
JWT	JSON Web Token
LINQ	Language Integrated Query
MSSQL	Microsoft SQL Server
MVC	Model-View-Controller
.NET	Dot Net
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
PHP	Hypertext Preprocessor

PK	Primary Key
REST	Representational State Transfer
SCSS	Sassy Cascading Style Sheets
SEO	Search Engine Optimization
SPA	Single Page Application
SQL	Structured Query Language
TS	TypeScript
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience
VDOM	Virtual Document Object Model
VPS	Virtual Private Server
XML	Extensible Markup Language
XLSX	Microsoft Excel Open XML Format Spreadsheet
Zkratka	Celý název
API	Application Programming Interface

Přílohy

Na USB disku byl přiložen projekt, ve kterém je obsažen zdrojový kód aplikace, struktura databáze a konfigurace pro spuštění aplikace pomocí platformy Docker. Na disku je následně přiložen 3x CSV soubor, které je možné importovat do aplikace.