ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Skála** Jméno: **Jan** Osobní číslo: **465854**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Systém pro správu uživatelů projektu EduARd

Název bakalářské práce anglicky:

User rights management system for EduARd project

Pokyny pro vypracování:

1) Seznamte se s aktuálním stavem projektu EduARd (rozšířená realita pro školy).
2) Navrhněte a implementujte backend pro systém integrující jednotlivé části (klientské-java a webové) aplikace do jednoho celku. Spolupracujte při návrhu s kolegy řešícími další části systému (systémy pro editaci dat na webu a jejich prohlížení na mobilním zařízení).
3) Řešte primárně autentizaci a správu uživatelů, systém přihlašování, role, přístupnost a práva.
4) Vygenerujte základní UI pro implementované části systému.

Seznam doporučené literatury:

[1] Fanie Reynders, Modern API Design with ASP.NET Core 2, Apress 2018
[2] Robert C. Martin, Clean Architecture, Personal Education 2017

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. David Sedláček, Ph.D., katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **23.01.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Ing. David Sedláček, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra počítačů

Systém pro správu uživatelů projektu EduARd

Bakalářská práce

Autor:
Jan Skála

Vedoucí projektu:

Ing. David Sedláček,

Ph.D.

Bakalářský program:
Otevřená informatika

Obor:

Software

2019

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci na téma Systém pro správu uživatelů projektu EduARd jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Praze dne ....

Jan Skála

Poděkování

Tímto bych chtěl poděkovat Ing. Davidu Sedláčkovi PhD za možnosti podílet se na projektu. Zároveň také za vedení a cenné rady během formování konceptu celé aplikace. Dále pak Tomáši Hercegovi, za konstruktivní kritiku během oponentury.

V Praze dne ....

Jan Skála

Abstrakt

V moderním světě je tendence maximálně digitalizovat veškerý možný obsah a celkově využívat moderní technologie k zobrazování obsahu. Naučné stezky nejsou výjimkou.

Tato práce se zaobírá vytvořením webové služby, se kterou pomocí definovaného rozhraní komunikují dílčí aplikace projektu EduARd. Je zaměřena primárně na správu uživatelů, institucí, autentizaci a autorizaci. Kromě webové služby je jejím výstupem také portál pro systémové administrátory projektu EduARd.

Klíčová slova API, web, autentizace, autorizace, portál, webová aplikace, .Net Core, CQRS, REST, Swagger, OPEN API, SPA, Vue.js

Abstract

In modern world, it is more and more popular to make digital version of available content and leverage modern technology for content viewing. Nature trails are no exception.

This work's goal is to create a web service, which can communicate with other apps that are part of EduARd project using a defined interface. It is focused on user management, institution management authentication and authorization. Aside the web service, the output of this thesis is also a portal for system administrators of EduARd project.

Key words API, web, authentication, authorization, portal, web app, .Net Core, CQRS, REST, Swagger, OPEN API, SPA, Vue.js

Odkaz na tuto práci Skála, Jan. Systém pro správu uživatelů projektu EduARd. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta elektrotechnická, 2019.

OBSAH

1 Úvod

Projekt EduARd si klade za cíl vytvořit systém pro tvorbu učebních pomůcek tzv. virtuálních naučných stezek, které umožní přesunout výuku vybraných témat ze školních lavic ven, za podpory moderních univerzálních elektronických pomůcek jako jsou tablety. Systém by měl sloužit pro základní a střední školy při podpoře výuky hlavně v oblasti přírodních věd, ale i humanitních věd jako jsou dějepis a občanská nauka. Motivací pro realizaci systému je stav, kdy vzdělávací instituce pořídily pro podporu výuky velké množství elektronických pomůcek, ke kterým potřebují kvalitní obsah. Ten může být buď vyroben specializovanou firmou anebo přímo učiteli na míru jejich potřebám, k čemuž ale potřebují právě systém pro tvorbu obsahu. V rámci projektu proto vytvoříme jak systém pro tvorbu obsahu tohoto typu učebních pomůcek, tak aplikace umožňující prohlížení obsahu právě na tabletu nebo mobilním zařízení. Konkrétně se bude jednat o tvorbu projektů a úloh jak referencovaných vůči GPS pozici, pro použití ve školních lavicích, tak s využitím rozšířené reality. Tyto aplikace kombinují interakci s reálným světem prostřednictvím obrazu snímaného kamerou nebo přímo průhledným displejem, s doplňkovou informací zapojenou do reálného světa tabletem. Jsou proto vhodnější pro demonstraci a ukázky než pouze např. webové stránky.

Dalším využitím výstupů projektu bude příprava podkladů pro tvorbu podobných učebních pomůcek v učebnách. Pro tyto projekty a úlohy se využijí výše zmíněné vlastnosti systému kromě GPS lokalizace použitelné pouze pro vnější lokalizaci. Půjde tak tedy připravit učební pomůcka pro vnější i vnitřní použití. V rámci projektu počítáme také s možností sdílení vytvořených úloh a projektů mezi školami. Tato funkce bude součástí webového systému pro tvorbu obsahu. Zároveň tímto vznikne prostor pro možné zapojení dalších partnerů, kteří budou moci vyrábět specializovaný obsah. (1)

Cílem této práce je návrh a realizace systému, který dokáže komunikovat s ostatními dílčími aplikacemi projektu EduARd a umožnit integraci nejen s aplikacemi, které běží ve webovém prohlížeči, ale i s těmi které běží na mobilním telefonu nebo jiné platformě. Dále je potřeba vytvořit uživatelské rozhraní, které je schopno funkcionalitu backendu zabývající se správou institucí používat. Během návrhu se snažit o maximální přehlednost a čistotu architektury. Dodržovat principy a paradigmata softwarového inženýrství jakými jsou DRY a SOLID. Vybudovat udržitelnou a testovatelnou aplikaci, kterou lze nasadit bez složité konfigurace. Většinu funkcionality aplikační vrstvy otestovat pomocí unit testů a některé variace případů užití pokrýt pomocí integračních testů. Zachovat aplikační logiku nezávislou, aby ji bylo možné bez problému vyjmout a použít jinde. Během toho zajistit včasné doručení testovací verze ostatním studentům, aby měli dostatek času k otestování a implementaci svojí funkcionality.

Nejprve se problematika popíše pomocí systémových entit, systémových rolí a případů užití. Následuje návrh řešení, který popisuje schéma autentizace a porovnává metody, které se při implementaci autentizace používají. Další sekce detailně popisuje navrženou architekturu celé aplikace. Pro každou vrstvu architektury se zvolí vhodná technologie nebo knihovna a odůvodní se proč se tak stalo. Po dokončení návrhu architektury se navrhne rozhraní jak pro backend, tak pro frontend. V kapitole implementace je popsáno, kde běží testovací instance obou aplikací. V kapitole testování je popsáno, jakým způsobem probíhalo testování a automatický proces nasazování a aktualizace aplikace.

Motivací práce je dát školám prostředky k efektivní modernizaci výuky a možnost přenést výuku ven z lavic. Propojit všechny dílčí aplikace, aby spolu mohly bezpečně komunikovat a ochránit obsah, který si instituce (školy) vytváří. Ukázat, že mobilní telefony nebo tablety mají smysluplné využití i ve výuce a že plošné zákazy jejich používání nejsou správnou cestou.

2 ANALÝZA

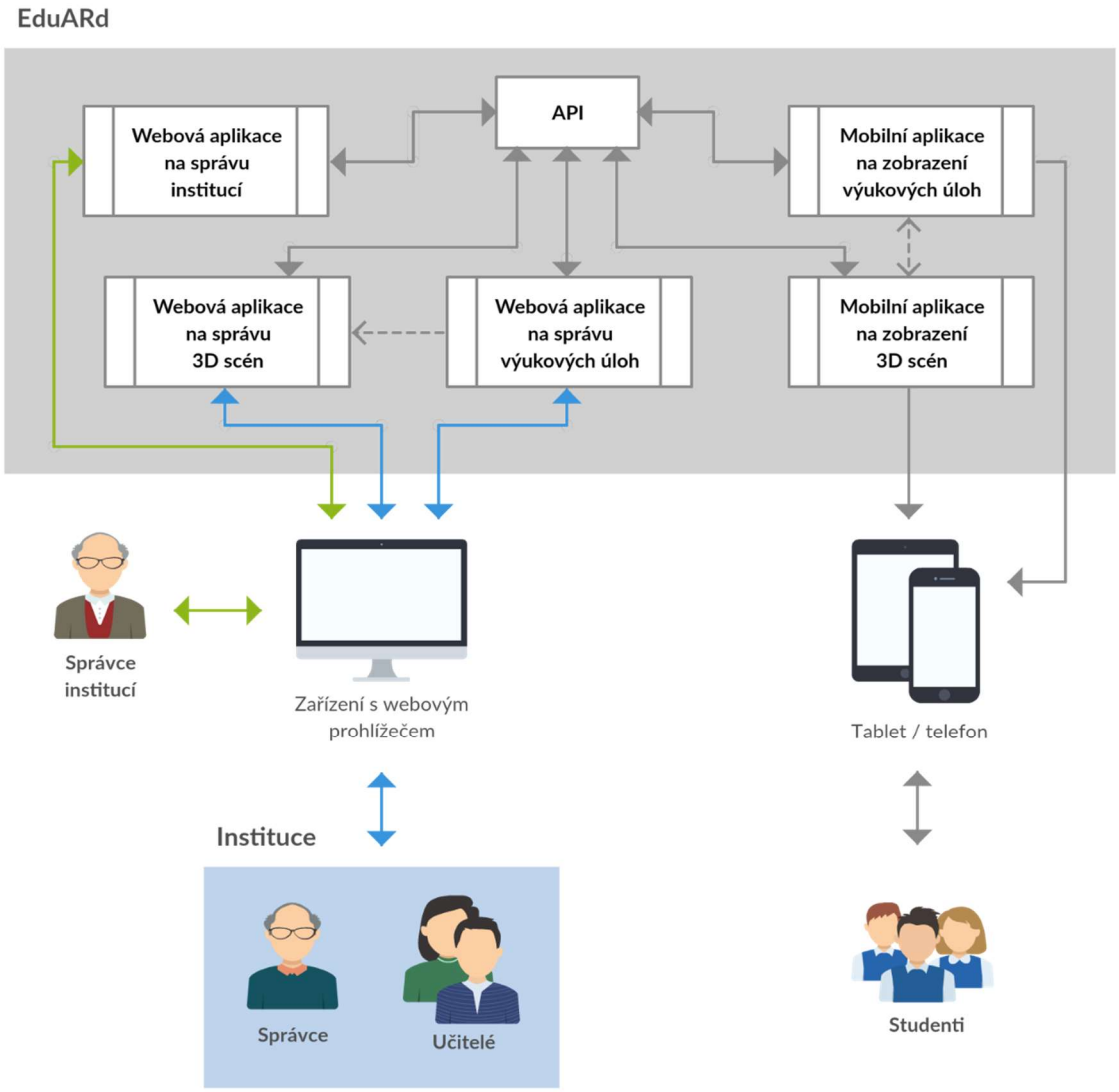
Můj rozsah práce spočívá v seznámení se s aktuálním stavem projektu EduARd a jeho dílčími aplikacemi. Na základě toho identifikovat jaká data potřebují nahrát na server a jaká data naopak ze serveru získat. Následně navrhnout backend pro systém integrující jednotlivé části (dílčí aplikace). Při tom spolupracovat a komunikovat s ostatními kolegy, kteří na těchto částech pracují. Zejména s Martinem Sklenárem, který implementuje část webové služby zajišťující ukládání, filtrování a stahování obsahu. Úkolem je řešit primárně autentizaci, správu uživatelů a institucí. Poslední části je implementovat základní uživatelské rozhraní, které umožňuje používání výše zmíněné funkcionality skrz webový prohlížeč – zejména správu institucí.

Projekt EduARd se skládá z několika dílčích aplikací. Část z nich je soustředěná primárně na tvorbu obsahu a jeho nahrávání na server, zatímco druhá je orientovaná na konzumaci obsahu a jeho následnému zobrazení uživateli. Všechny aplikace a vztahy mezi nimi jsou zaneseny v diagramu (Obrázek 2-1 Diagram dílčích aplikací). Vzhledem k tomu, že dílčích aplikací je několik a patří sem jak mobilní, tak webové aplikace, je nutné definovat takové rozraní, se kterým dokážou bez větších obtíží komunikovat. Je potřeba myslet na to, že kolegové svoje aplikace implementují také v rámci svých bakalářských prací a nelze tedy dokončit projekt na poslední chvíli, byť v termínu, a poté až integrovat s jinými systémy. Dokumentace a testovatelnost rozhraní backendu je nutností a samozřejmostí.

Infrastruktura backendu se buduje od nuly, a to nám dává několik výhod. Netrápí nás kompatibilita se starým kódem a při implementaci se můžeme držet moderních trendů v architektuře softwaru. To se bude týkat zejména návrhu autentizace a autorizace. Největším problémem je, že mobilní aplikace pro zobrazování obsahu mají být uzpůsobené tak aby fungovaly i bez internetu. Obsah je tedy ze serveru potřeba stáhnout a uložit do perzistentního úložiště telefonu. Ten ale může nabývat na velikosti zejména u 3D scén. V takovém případě je potřeba stahovat na pozadí. Jenže proces na pozadí, který obsah stahuje by neměl disponovat jinými dalšími právy, natož heslem uživatele. U autentizace je tedy nutností počítat s delegací přístupu.

Mimo to, jedna z variant mobilních aplikace na zobrazení výukových úloh nemá přihlašovací formulář pro uživatele. Slouží k zobrazování jen určitého výběru obsahu instituce. Je tedy potřeba této aplikaci zajistit delegovaný přístup jen k danému obsahu, který může stahovat. Zároveň musí být možnost toto oprávnění kdykoli v budoucnu zrušit.

Uživatelé budou mít možnost se přihlásit a zůstat přihlášeni, aby nemuseli opakovaně vyplňovat heslo. Zajistit to takovým způsobem, aby sama klientská aplikace nemusela heslo uživatele někde uchovávat. Zároveň pokud se přihlásí v jedné části systému, musí mít přístup i do další části, ke kterým mají potřebná oprávnění. Přihlášení by mělo fungovat, i pokud backend poběží na více instancích jednoho, či více strojů. Kromě toho je potřeba umožnit přístup i aplikacím, které fungují pouze jako jednoúčelové prohlížeče a nemají přihlašovací dialog.



Obrázek 2-1 Diagram dílčích aplikací

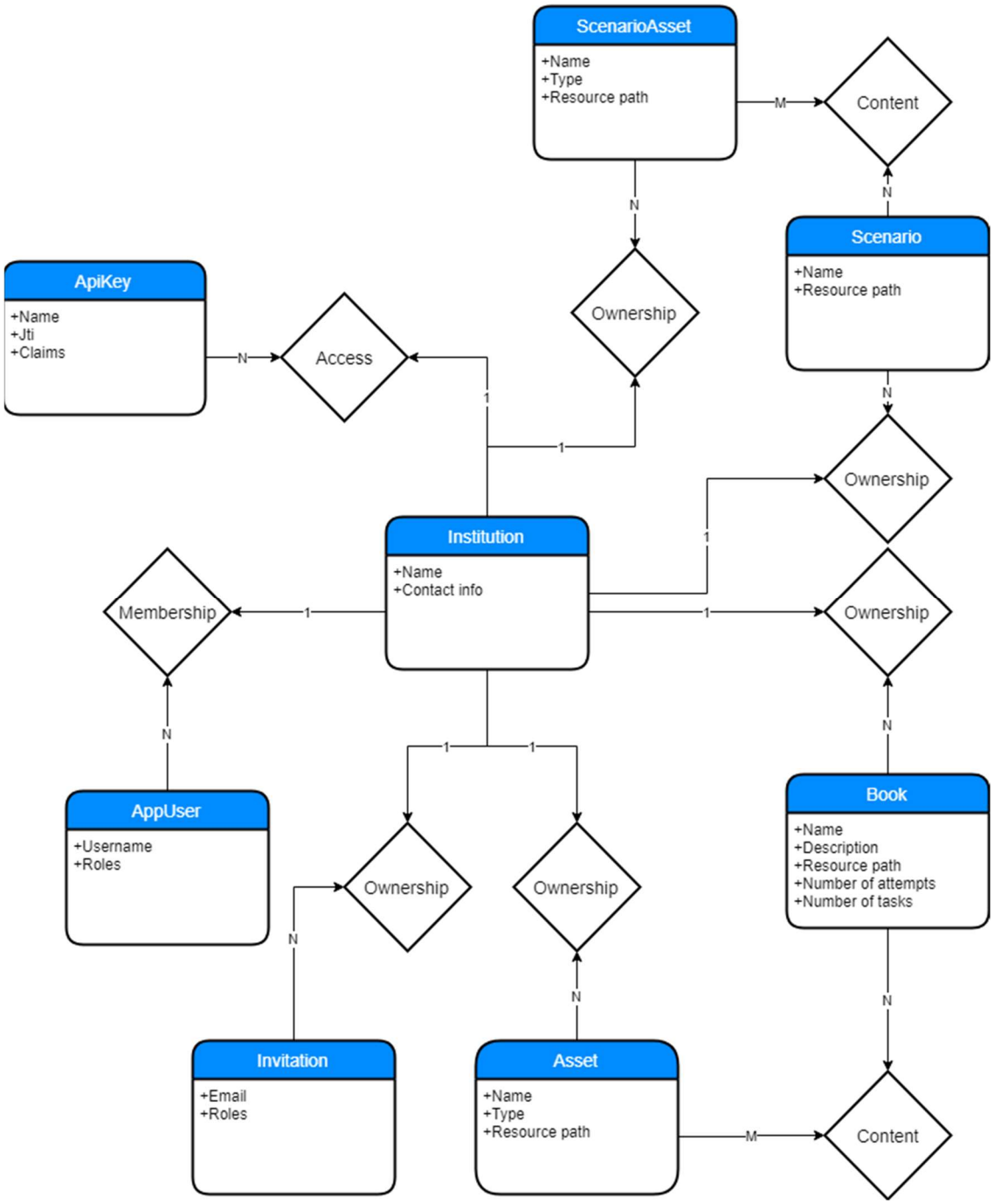
Autor obrázku: Anastasia Surikova

2.1 ENTITY V SYSTÉMU

1. **Instituce (Institution)**
   Obsahuje název, kontaktní údaje. Má seznam členů a pozvánek. Vlastní učebnice, scénáře, přílohy a přílohy ke scénářům.
2. **Uživatel aplikace (AppUser)**
   Obsahuje data o uživateli. Uživatelské jméno i to do jaké role patří.
3. **Uživatelská role (AppRole)**
   Data o uživatelské roli. Název a identifikátor.
4. **Učebnice (Book)**
   Obsahuje metadata o virtuální naučné stezce: název, popis, počet úkolů a umístění odkud lze naučnou stezku v podobě xml stáhnout.
5. **Příloha (Asset)**
   Obsahuje metadata o příloze k učebnici: název, umístění odkud lze přílohu stáhnout, typ přílohy.
6. **Scénář (Scenario)**
   Může být použit ve virtuální naučné stezce. Obsahuje metadata o 3D scéně: název, umístění odkud lze 3D scénu v podobě jsonu[1] stáhnout.
7. **Příloha scénáře (ScenarioAsset)**
   Obsahuje metadata o příloze ke scénáři: název, umístění odkud lze přílohu stáhnout, typ přílohy.
8. **Pozvánka (Invitation)**
   Eviduje informaci o tom, kdo byl pozván, do jaké instituce a jakou uživatelskou roli obdrží až pozvánku přijme.
9. **Aplikační klíč (ApiKey)**
   Umožňuje autorizaci bez použití uživatelského jména a hesla. Dává přístup ke konkrétním učebnicím a scénářům. Využívají ho jednoúčelové aplikace pro zobrazování statického obsahu. Každá pozvánka má unikátní token, který slouží k identifikaci. Každá pozvánka může být přijata pouze jednou.

Všechny entity a jejich relace jsou znázorněny na diagramu doménového modelu (Obrázek 2-2 Konceptuální schéma).

[1] JavaScript Object Notation - Jedná se o jednoduchý datový formát. Využívá se pro přenos strukturovaných dat. Oproti XML neobsahuje silné typové definice, díky tomu je přenos dat menší.



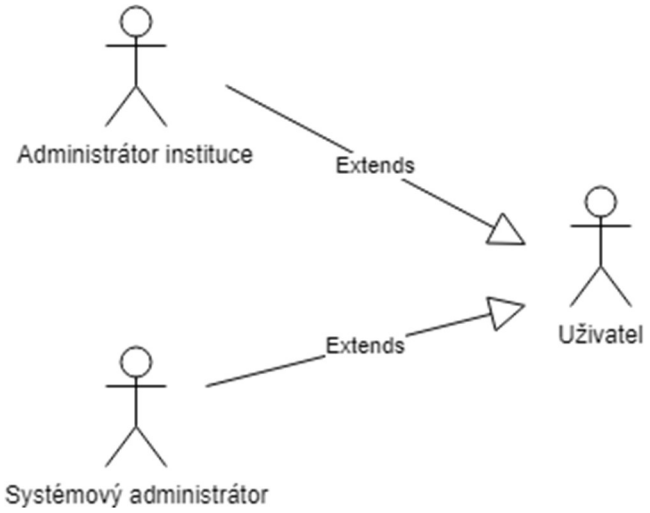
Obrázek 2-2 Konceptuální schéma

2.2 ROLE V SYSTÉMU

1. **Uživatel**
   Spadá pod jednu instituci. Každý uživatel má přístup k obsahu, který vlastní nebo sám vytvořil. Zároveň má práva na obsah, který vlastní jeho instituce.
2. **Editor**
   Může v rámci instituce spravovat její obsah, tvořit a editovat materiál.
3. **Administrátor instituce**
   Spravuje uživatele dané instituce a jejich role. Může přizvat uživatele do instituce nebo je z instituce odebrat.
4. **Systémový administrátor**
   Zakládá jednotlivé instituce a může upravovat data o institucích.

2.3 PŘÍPADY UŽITÍ

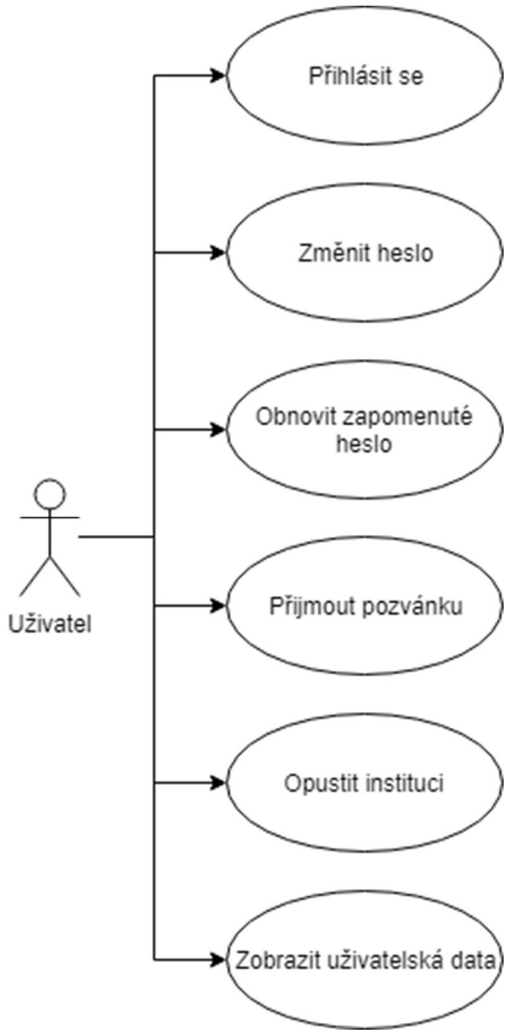
Nejsou zmíněny všechny funkcionality systému, protože sekce týkající se ukládání obsahu a jeho zobrazování implementoval Martin Sklenár v rámci svojí práce. Aktéři a jejich relace jsou zakresleny v diagramu (Obrázek 2-3 Aktéři).



Obrázek 2-3 Aktéři

2.3.1 Všichni uživatelé

Případy užití společné pro všechny uživatele (Obrázek 2-4 Případy užití pro všechny uživatele).



Obrázek 2-4 Případy užití pro všechny uživatele

Přihlásit se
Každý uživatel se může do aplikace přihlásit prostřednictvím svého uživatelského jména a hesla.

Změnit heslo
Každý uživatel si může změnit heslo. Musí zadat staré heslo a poté dvakrát nové heslo pro ověření.

Obnovit zapomenuté heslo
Každý uživatel si může obnovit zapomenuté heslo zadání svého emailu, který je současné uživatelským jménem. Následně obdrží email, který ho přesměruje na formulář pro resetování hesla.

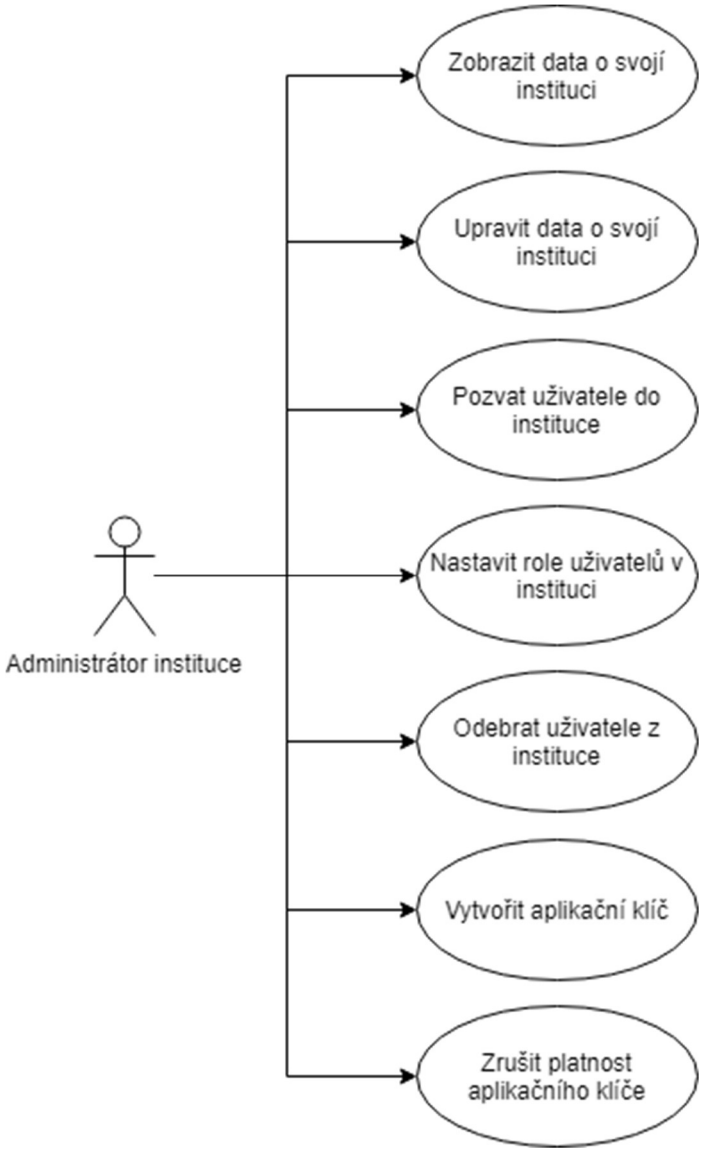
Přijmout pozvánku
Každý uživatel, který obdržel pozvánku, ji může přijmout a stát se členem instituce. Nesmí však již být členem instituce.

Opustit instituci
Každý uživatel smí opustit instituci. Opuštěním instituce mu zaniká účet.

Zobrazit uživatelská data
Každý uživatel si může zobrazit data, která o něm systém shromažďuje.

2.3.2 Administrátor instituce

Případy užití, které může vykonávat jenom administrátor instituce (Obrázek 2-5 Příapdy užití - administrátor instituce).



Obrázek 2-5 Příapdy užití - administrátor instituce

Zobrazit data o svojí instituci

Administrátor instituce si může zobrazit detail svojí instituce. Název, seznam členů včetně jejich rolí, seznam pozvánek a kontaktní údaje.

Upravit data o svojí instituci

Administrátor instituce může upravit název nebo kontaktní údaje svojí instituce.

Pozvat uživatele do instituce

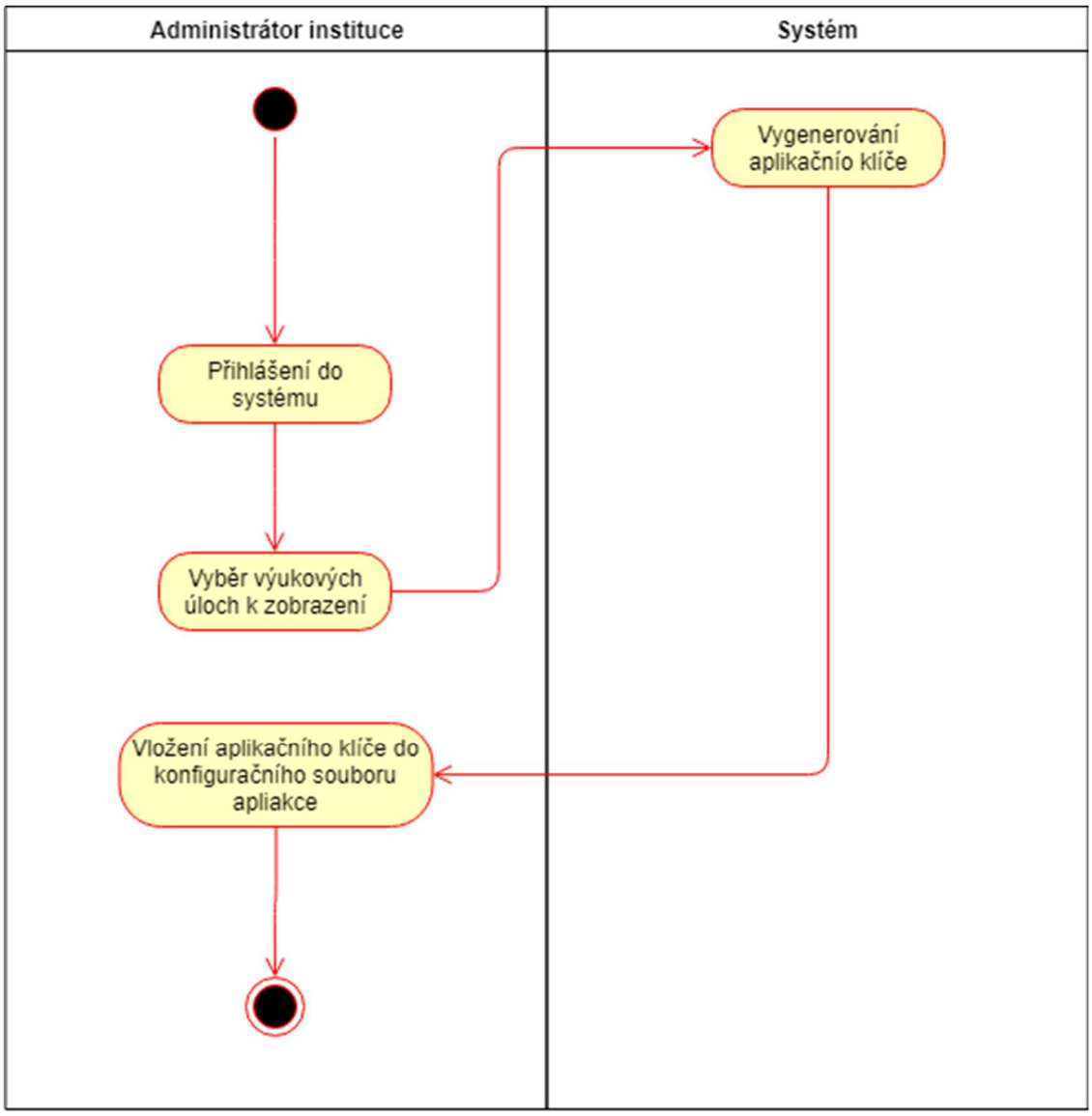
Administrátor instituce může emailem pozvat další lidi do své instituce. Součástí pozvánky je i seznam rolí, kterými budou disponovat. Jakmile je pozvánka přijata, vznikne pozvanému účet a stane se členem instituce.

Nastavit role uživatelů v instituci

Administrátor instituce může editovat role ostatních členů včetně té své v rámci svojí instituce.

Odebrat uživatele z instituce
Administrátor instituce může odebrat libovolného člena ze své instituce.

Vytvořit aplikační klíč
Administrátor instituce může pro svoji instituci vytvořit aplikační klíč. Pro každý klíč může nastavit jaké knihy s ním lze zobrazit. Některé mobilní aplikace neumožní uživateli přihlásit se, ale nechají ho zobrazit některé výukové úlohy. Získají k nim přístup právě pomocí aplikačního klíče, který budou mít ve svém konfiguračním souboru. Případ užití je zakreslen pomocí diagramu aktivit (Obrázek 2-6 Vytvoření aplikačního klíče).

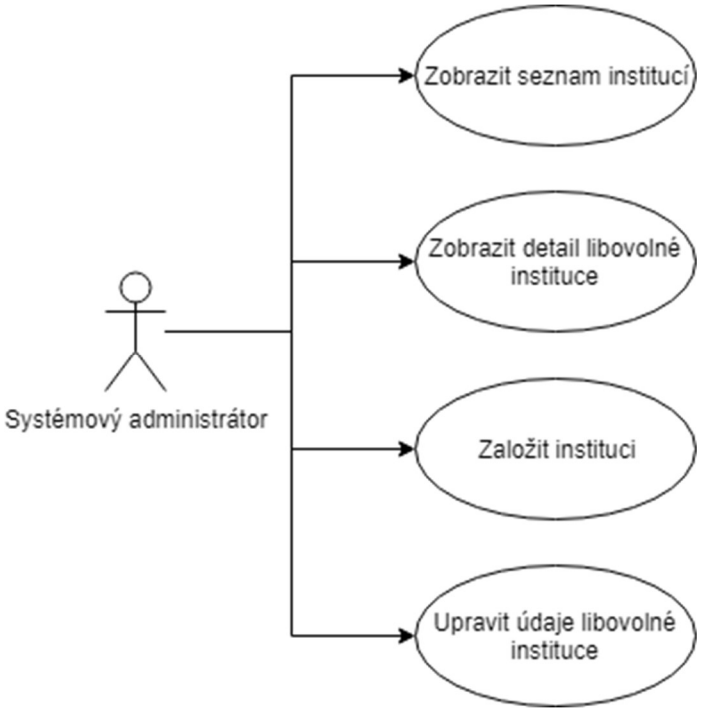


Obrázek 2-6 Vytvoření aplikačního klíče

Zrušit platnost aplikačního klíče
Administrátor instituce může zrušit platnost aplikačního klíče ve své instituci.

2.3.3 Systémový administrátor

Případy užití, které může vykonávat jenom systémový administrátor (Obrázek 2-7 Případy užití - systémový administrátor).



Obrázek 2-7 Případy užití - systémový administrátor

Zobrazit seznam institucí

Systémový administrátor může zobrazit seznam všech institucí. Seznam obsahuje název, počet členů a počet knih.

Zobrazit detail libovolné instituce

Systémový administrátor může zobrazit detail libovolné instituce. Název, seznam členů včetně jejich rolí, seznam pozvánek a kontaktní údaje.

Založit instituci

Systémový administrátor může založit instituci. K založení instituce vyplní název, kontaktní údaje a email administrátora, který je následně pozván do instituce. Přijetím pozvánky mu vznikne účet a stane se členem instituce.

Upravit údaje libovolné instituce

Systémový administrátor může upravit název a kontaktní údaje libovolné instituce.

3 NÁVRH ŘEŠENÍ

Existuje několik přístupů, jak v dnešní době koncipovat architekturu aplikací. Některé směry jsou populární, jiné méně. Není vždy rozumné okamžitě skočit po aktuálním trendu jen proto, že je nový. Kdybychom však neinovovali metodiky, nikam bychom se neposouvali. U projektu EduARd se zatím spekuluje, zda poběží na lokální infrastruktuře nebo zda se využije poskytovatel cloudových služeb. V takové situaci je vhodné zvolit kompaktní řešení nezávislé na platformě, které nevyžaduje složitou konfiguraci.

Jedním se zavedených architektonických trendů jsou tak zvané monolitické aplikace. Funckionalita je jeden celek zapouzdřená do aplikace. Od uživatelského rozhraní až po kód na pozadí. Aplikace obstarává veškeré úkony potřebné k tomu, aby byl splněn požadavek od uživatele. Zřídkakdy volá nějakou další aplikaci. Nevýhodou tohoto přístupu je, že takovéto aplikace nejsou modulární. Nelze aktualizovat část funkcionality, bez nutnosti zastavení celé aplikace. Zároveň je problematické škálování. Nelze například škálovat nezávislá část s uživatelským rozhraním a část, která zajišťuje výpočty.

Opačným přístupem jsou mikroslužby. Jedná se o koncepčně malé části funkcionality, které fungují jako samostatné aplikace. Mikroslužby by na sobě měly být co nejvíce nezávislé. Výhodou tohoto přístupu je možnost nasadit a udržovat každou funkcionalitu zvlášť a zároveň škálovat podle potřeby. Nevýhoda spočívá v komunikaci mezi službami. Mnohdy totiž neběží na stejném stroji a je potřeba, aby mezi sebou udržovaly spojení, například komunikací pomocí REST [2]protokolu. Složitý problém je zajištění konzistence napříč touto množinou služeb.

Výstupem mojí práce bude webová služba (API[3]), která poběží na webovém serveru, v cloudu nebo jako samostatná aplikace. Tato služba bude přístupná ostatním dílčím aplikacím jako REST API. Rozhraní bude vystavovat Open API definici, která je strojově čitelná a umožní tak integraci s jinými systémy nebo klientskými aplikacemi. Zároveň bude sloužit jako dokumentace a testovací prostředí pro konzumenty API.

Jedním z konzumentů tohoto API bude právě portál pro systémové administrátory. Kromě funkcionality přihlašování bude využívat hlavně aplikační logiku, která zajišťuje správu institucí. Dalšími konzumenty budou mobilní aplikace, které zajišťují zobrazování virtuálních naučných stezek nebo portál sloužící k jejich editaci.

[2] Representational State Transfer - Implementuje základní operace (CRUD) pomocí metod HTTP protokolu (GET, POST, PUT, DELETE).

[3] Application Programming interface - Rozhraní, které je vystaveno pro komunikaci. Zapouzdřuje složitější funkcionality za jednodušší metody a procedury, typicky CREATE, READ, UPDATE, DELETE.

3.1 OPEN API DEFINICE

Jedná se o standardizovanou strojově čitelnou definici Web API. Podrobný popis toho, jaké má API endpointy a co na nich lze volat za HTTP metody. Obsahuje definici všem modelů, operací, a dokonce i schématu autentizace, kterých může být více než jedno. Ke všem metodám je definovaný formát odpovědi a jsou popsány všechny chybové stavy, které mohou nastat. Zároveň lze dokumentovat jaké chyby může API vracet a za jakých podmínek. Bývá standardem z Open API definice vygenerovat funkční obslužný kód, který je schopný webovou službu volat. (2)

Na podobném principu fungoval protokol SOAP (Simple Object Access Protocol). Je více komplexní než REST a definuje více standardů. Má dokonce vestavěnou retry logiku. (3) V kombinaci s WSDL, který slouží jako popis služby a definuje metody, které umí služba vykonávat, bylo možné sestavit zdokumentovanou službu podobně jako toho lze dosáhnout s REST a Open API. Nicméně tento přístup vyžadoval, aby se komunikovalo jen pomocí xml. Komunikace s pomocí formátu json nebyla možná. To má za následek nižší propustnost a celkovou rychlost služby i to kolik požadavků dokáže odbavit za minutu. (4) Navíc je velice obtížné volat službu využívající SOAP přímo z internetového prohlížeče, a proto je pro tuto práci nevhodný.

3.1.1 Code first

Nejprve se vytvoří funkční webová služba a implementují se všechny controllery a metody na nich. Později se využije framework[4], který analýzou napsaného kódu vygeneruje Open API definici. Následně mohou konzumenti vygenerovat obslužný kód pro tuto webovou službu.

3.1.2 Definition first

Nejprve se definuje, jak budou vypadat API endpointy a modely, které bude služba využívat. Definují se všechny odpovědi a následně se napíše Open Api definice. Z té se pak následně vygeneruje jak server, tak klientské aplikace. Ve vygenerovaných aplikacích se později implementují definovaná rozhraní.

3.2 AUTENTIZACE

Pro použití většiny funkcionalit API se uživatel bude muset autentizovat. Tradiční způsob řešení autentizace, který se využívá v konvenčních a často monolitických aplikací, spočívá v odeslání dat na server přes přihlašovací formulář. Na straně serveru se dotazem do databáze ověří, zda jsou přihlašovací údaje správné. Pokud ano, načtou se všechna potřebná data uživatele a uloží se do vytvořené session, což znamená že se na straně serveru vytvoří unikátní řetězec, který se zároveň odešle uživateli. Ten si jej může uložit například do cookies v prohlížeči. Při další komunikaci je společně s každým požadavkem odesílán i zmíněný řetězec a server pak načítá dříve uložená data.

Vhledem k tomu, že ne všechny dílčí aplikace projektu EduARd běží v prohlížeči, není vhodné využívat cookies. Chceme-li mít ale funkční autentizaci je potřeba při každém požadavku odesílat informace pro ověření identity. Například v hlavičce požadavku nastavit hodnotu Authorization. Existuje několik způsobů. Bez ohledu na to, jaký typ autentizace se používá, by všechny měli komunikovat přes HTTPS a nikoli přes nezašifrované HTTP. Ideálně použít moderní kryptografický protokol, například TLS 1.2, případně 1.3.

3.2.1 Základní autentizace (basic)

Jedná se o nejjednodušší a zároveň nejmíň bezpečný postup. V hlavičce požadavku se do hodnoty Authorization uloží uživatelské jméno a heslo ve formátu base64. Při každém požadavku stačí deserializovat data z hlavičky a ověřit identitu uživatele porovnáním obdržených hodnot s hodnotami v databázi. Poté ověříme, zda má přístup k požadovanému zdroji. Pokud nebyl úspěšně autentizován nebo nemá patřičná práva, vrátíme http kód 401 (unauthorized). Na první pohled je zde bezpečnostní

[4] Softwarový balík usnadňující programování. Má větší rozsah než knihovna. Typicky zapouzdřuje a řeší komplexní problém.

hrozba v podobě odesílání jména a hesla v nezašifrovaném formátu. Tento problém částečně vyřeší použití TLS, ale pokud se přes to podaří útočníkovi data odposlechnout získá heslo uživatele. Získá tak nejenom přístup k daném zdroji ale i k celé uživatelské identitě. Zároveň při obnovení stránky nebo při zavření prohlížeče se ztratí data z paměti, a proto je nutné je někde uchovávat a klientské aplikaci, opět v nezašifrované podobě.

3.2.2 Autentizace s použitím tokenu (stateless)

Když se uživatel úspěšně přihlásí pomocí uživatelského jména a hesla, obdrží ze serveru přístupový token. Při každém dalším požadavku se token odesílá v hlavičce požadavku. Server ověří pravost tokenu a v některých případech i identitu držitele tokenu. Ušetří se tak ověřovací dotaz do databáze, a navíc se heslo nevyskytuje v každém požadavku. Zároveň se aplikace stává lépe škálovatelná, protože si nemusí držet žádný stav. Zvýšením bezpečnosti můžeme dosáhnout tak, že k tokenu přidáme jeho životnost. Jakmile skončí platnost tokenu, server jej již nepovažuje za platný. V takovém případě musí uživatel získat nový přístupový token.

3.2.3 JWT (JSON Web Token)

Jedná se o otevřený standard, využívající JSON jako nosič informací přístupových tokenů. Token obsahuje identitu uživatele a často je doplněn o další potřebné informace (role, práva atd.). V kombinaci s JSON Web Signature (JWS) a JSON Web Encryption (JWE) můžeme docílit ještě větší bezpečnosti. JWS nám umožňuj tokeny podepisovat a zabránit tak jejich padělání, zatímco JWE skryje informace v tokenu před uživatelem, a dokáže je přečíst pouze server. Často však chceme v tokenu zobrazit nějaké informace, aby si o ně klientská aplikace nemusela explicitně žádat. V našem případě uživatelské jméno, název nebo identifikátor instituce.

JWT se skládá ze tří části:

- Header – obsahuje typ tokenu a typ hashovacího algoritmu
- Payload – obsahuje data o uživateli (uživatelské jméno, role, email atd.)
- Signature – podpis tokenu, hash dvou přechozích částí a tajného klíče, který klient nezná

3.2.4 Přístupový token

Obsahuje veškerá data, který server potřebuje o uživateli znát. Pomocí těchto informací server rozhodne, zda má uživatel nebo zařízení přístup ke zdroji, který požaduje. Obvykle se jim dává krátká doba platnosti. Ve chvíli, kdy dojde k odposlechnutí tokenu, útočník má jen omezenou dobu přístupu. Její délka se liší podle typu aplikace. Bankovní aplikace mají typicky velice krátkou dobu platnosti, oproti méně kritickým aplikacím, jakou je například obrázková galerie. Pohybuje se v řádech desítek minut s ohledem na dobu, po kterou uživatel aplikaci používá.

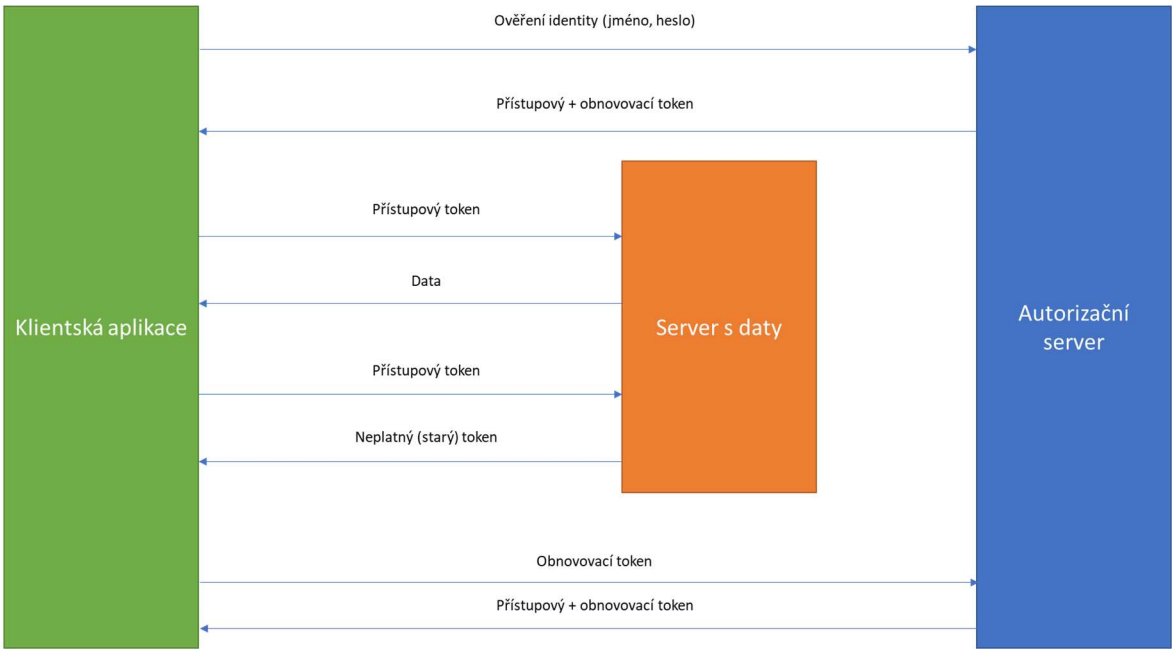
3.2.5 Obnovovací (refresh) token

Tento token se využívá k získání nového tokenu bez nutnosti vyplňování jména a hesla. Jakmile přístupovému tokenu vyprší platnost, vyžádá si klientská aplikace obnovení tokenu. Spolu s obnovovacím tokenem odešle i token, kterému vypršela platnost. Částečně se tím zvýší bezpečnost, protože útočník pro získání nového tokenu potřebuje i starý token. Oproti přístupovému tokenu je obnovovací token uložen na serveru, aby mohl být později zneplatněn. Je evidentní že obnovovací token musí mít větší životnost.

Při ukládání obnovovacího tokenu můžeme docílit zvýšené bezpečnosti tím způsobem, že jej neukládáme v nezašifrované podobě, ale zacházíme s ním jako s heslem. Na server tedy uložíme pouze jeho hash. Pokud tedy útočník získá data z databáze, nebude znát pravou hodnotu tokenu. Zároveň při prvním použití obnovovací tokenu je ze serveru odstraněn. I kdyby tak útočník obnovovací token později získal, už mu nebude fungovat. (5)

3.2.6 Schéma autorizace

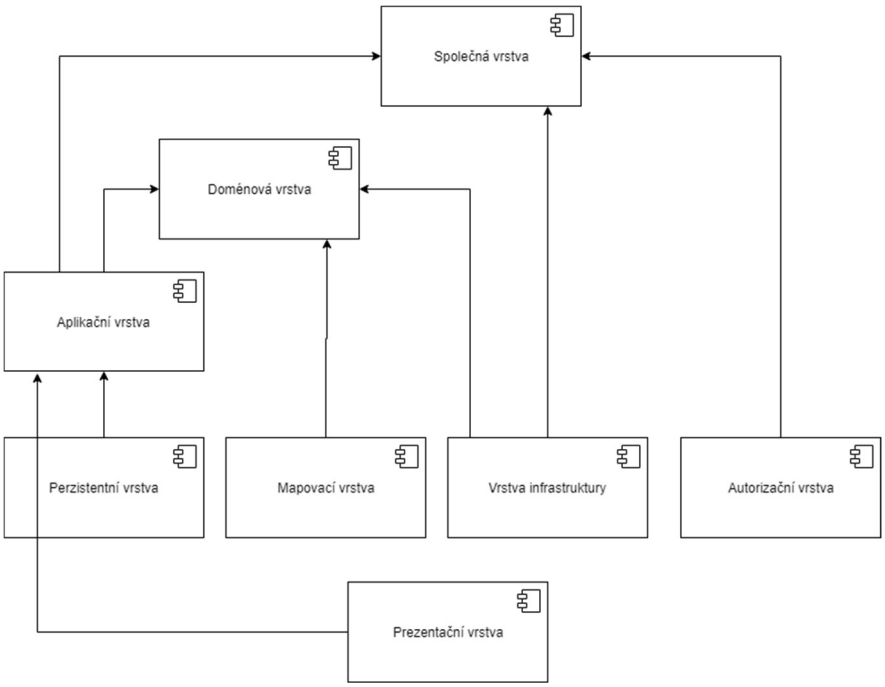
Klientská aplikace nejprve odešle přihlašovací údaje na autorizační server. Ten ověří jejich správnost. Pokud nejsou chybná vrátí přístupový a obnovovací token. Poté se klientská aplikace může dotazovat na data, ke kterým jim umožňuje token přístup. Buď si sama hlídá životnost tokenu nebo je upozorněna na jeho neplatnost serverem. Někdy se uplatňuje i tak zvaná sliding expirace. To znamená, že server token uzná i přes to že je nějakou dobu neplatný. Mezní hodnota bývá v řádech jednotek minut. Využívá se toho u aplikací, které jsou pod velkou zátěží. Požadavek může být ve frontě tak dlouho, že token, který s ním přišel, může ztratit platnost, než bude zpracován. Pro obnovení tokenu musí klientská aplikace opět komunikovat s autorizačním serverem. Pojem server zde nutně neznamená fyzický hardware. Někdy se může autorizační server i server s daty nacházet na stejném zařízení (Obrázek 3-1 Schéma autorizace).



Obrázek 3-1 Schéma autorizace

3.3 ARCHITEKTURA

Architektura webové služby (API) je rozdělena na několik vrstev. Závislosti na jednotlivých vrstvách jsou znázorněny v diagramu (Obrázek 3-2 Diagram component). Každá vrstva odpovídá za určitou část funkcionality. Celý návrh je koncipován tak, aby aplikační vrstva byla maximálně nezávislá. Definuje rozhrání, která potřebuje k fungování a ostatní vrstvy jej implementují. Funkcionalitu ostatních vrstev volá výhradně použitím rozhraní. Stává se tak snadno testovatelnou. V budoucnu je tak možné snadno vyměnit podpůrné vrstvy od mapování až po prezentaci.



Obrázek 3-2 Diagram component

3.3.1 Společná vrstva

Nezávislá na ostatních vrstvách. Obsahuje definici společných rozhraní, která slouží ke komunikaci mezi jednotlivými vrstvami. Kromě toho obsahuje konstanty a objekty, které pouze udržují informaci o konfiguraci nebo nastavení. Dále obsahuje pomocné třídy pro repetitivní úkony.

3.3.2 Doménová vrstva

Reprezentuje jednotlivé entity v systému jako třídy. Kompletně nezávislá vrstva. Kromě entit obsahuje DTO[5], na která se jednotlivé entity mohou mapovat.

3.3.3 Autorizační vrstva

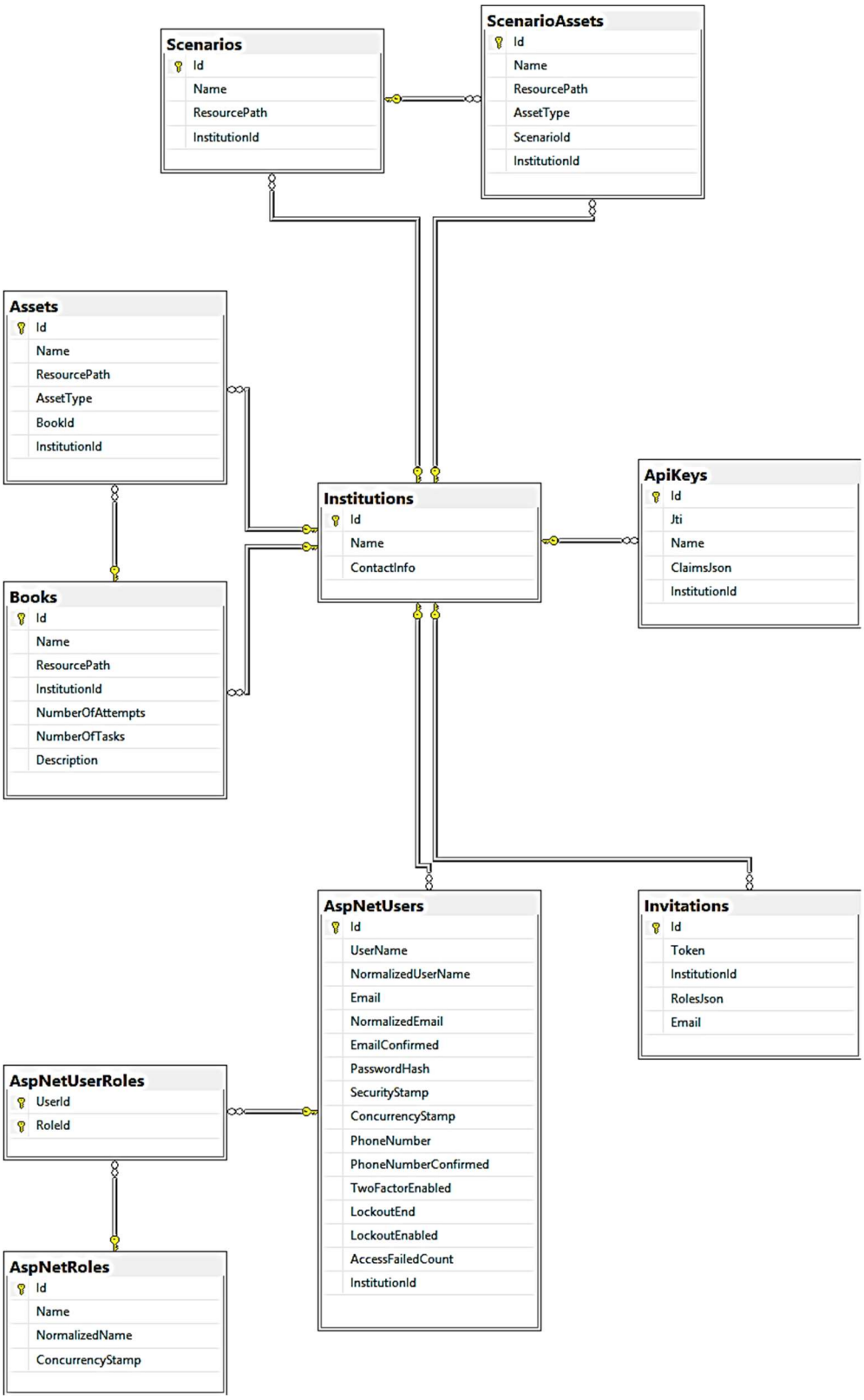
Vrstva poskytující implementace pro rozhraní definované společnou vrstvou. Zajišťuje vystavení přístupového a obnovovacího tokenu, jeho validaci a případnou obnovu.

3.3.4 Perzistentní vrstva

Následující sekce popisuje schéma tabulek, které udržují údaje o entitách v aplikaci a relace mezi nimi. Tabulky, které jsou vázané na specifickou technologii, kterou jsem požil při implementaci, začínají předponou „AspNet“. Schéma tabulek, které zajišťují práci s knihami, scénáři a přílohami implementoval Martin Sklenář v rámci svojí práce. Jedná se o tabulky: Books, Assets, Scenarios, ScenarioAssets. Jsou ponechány v diagramu, pro celkovou přehlednost (Obrázek 2-2 Konceptuální schéma). Vrstva zajišťující komunikaci s databází využívá mapovací vrstvu na mapování dat z databáze přímo na objekty doménového modelu a naopak. Z databází se nikde nekomunikuje přímo a vždy se využívá abstrakce v podobě perzistentní vrstvy a ORM[6].

[5] Data transfer object – Objekt, který pouze přenáší data mezi procesy nebo vrstvami aplikace. Nemá žádnou vnitřní funkcionalitu.

[6] Objektově relační mapování – Zajišťuje automatickou konverzi dat mezi (relační) databázi a objekty z aplikační domény.



Obrázek 3-3 Relační schéma

3.3.5 Vrstva infrastruktury

Infrastruktura implementuje rozhraní, která jsou nezbytná pro fungování aplikační logiky. Zajišťuje funkcionalitu pro odesílání emailů, vygenerování textu zpráv pro uživatele nebo ukládání souborů. Aplikační logika je na této vrstvě nezávislá a její funkcionalitu používá pouze prostřednictvím rozhraní.

3.3.6 Aplikační vrstva

Pro implementaci požadavků jsem se rozhodl využít návrhový vzor CQRS. Command Query Responsibility Segregation. Striktně rozdělí všechny operace na ty, které vyžadují pouze čtení (queries) a ty, které mohou způsobit změnu v databázi (commands). Obvyklý přístup dekompozice funkčních požadavků je rozdělit je na 4 operace: vytvoření, čtení, úprava, mazání. Někdy se také označují zkratkou CRUD [7](create, read, update, delete). Avšak s přibývajícími požadavky tato abstrakce naráží na svá omezení. Mnohdy potřebujeme na data nahlížet jiným pohledem. Buď se jedná o jejich filtrování nebo mapování, anebo dokonce o spojování informací z vícero míst do jednoho modelu. Naopak při vkládání chceme vytvořit pravidla, které nám dovolí uložit jen určitou kombinaci parametrů nebo odvodit data, která chceme uložit a liší se od dat, která systém obdrží. (6)

Nevýhoda tohoto přístupu spočívá v množství kódu navíc které vytváří. Je to převážně způsobeno charakteristikou jazyka C# a také protože mediátor využívá dependency injection[8], aby získal závislosti pro všechny komponenty, které volá. Oproti běžnému přístupu rozpadnutí funkčních požadavků na CRUD operace a jejich následné spojení například ve fasádě, připadá na každý funkční požadavek několik objektů. Každý z nich navíc obsahuje právě kvůli dependency injection konstruktor privátní atributy tříd a tím tak celkově roste množství kódu. Nicméně ani to neubírá na udržitelnosti celé aplikační logiky, protože při používání trochu sofistikovanějších vývojářských nástrojů lze takové věci jako konstruktor a atributy bez problémy vygenerovat i při nějakém složitějším refactoringu[9].

Každý požadavek odeslaný klientskou aplikací se namapuje na příslušný command nebo query. Následně je odeslán do mediátoru, který nalezne odpovídající handler, předá data a handler pak už povede požadovanou operaci. Kdykoli je použit libovolný command nebo query, tak se vždy vykoná validace, autorizace a ostatní definované kroky ještě před tím, než se spustí samotný handler. Celkově to tak přispívá k bezpečnosti, přehlednosti a udržitelnosti. Pokud se bude pro volání aplikační logiky používat mediátor, nemůže se stát, že by se validace nebo autorizace neprovedla. Zároveň pokud by bylo v budoucnu nutné aplikační logiku přenést nebo změnit prezentační vrstvu lze tak učinit, aniž by se roztříštila a bylo zapotřebí většího refactoringu.

Command, query

Jsou to prosté objekty, které mají za úkol jen přenášet data, nemají v sobě žádnou funkcionalitu. Vstupním parametrem každého commandu nikdy není entita ale DTO. To samé platí při vracení výsledků nějaké query.

Command/Query Handler

Klíčovou aplikační logiku zajišťují objekty, které se nazývají Handler. Každý handler umí zpracovat jeden command nebo query. Obsahují jen funkční kód bez validace nebo autorizace a komunikují s datovou vrstvou.

Command/Query Validator

Třída, která dědí z abstraktní třídy *AbstractValidator.* Zajišťuje validaci pro přidružený command nebo query. Při nevalidním vstupu se jako odpověď na požadavek vrátí kód 400 (bad request) a důvod proč byl požadavek nevalidní. Tím že je validace součástí aplikační logiky, lze celou aplikační logiku použít i z jiné prezentační vrstvy.

Command/Query Authenticator

Implementuje rozhraní *IRequestPreProcessor.* Vždy se spustí před handlerem, který má zpracovat command nebo query. Pokud selže autorizace, vyhodí výjimku. Výjimka se zpracuje třídou

[7] Zkratka pro Create Read Update Delete

[8] Vkládání závislostí – Jedná se o implementace návrhového vzoru IoC. Pokud chceme v aplikaci vytvořit instanci třídy, nepoužíváme konstruktor, ale zavoláme kontejner, který už ví, jaké závislosti třída potřebuje.

[9] Vylepšení vnitřní struktury kódu bez vlivu na vnější chování.

ErrorHandlingMiddleware a klient jako odpověď dostane 401 (unauthorized) a důvod, proč se požadavek nezdařil. Protože je autorizace součástí aplikační logiky, lze ji celou používat i z jiné vrstvy bez újmy na bezpečnosti.

RequestLogger
Implementuje rozhraní *IRequestPreProcessor.* Spouští se před každým handlerem a loguje data aktuálně spuštěného commandu nebo query pro snazší odhalování chyb.

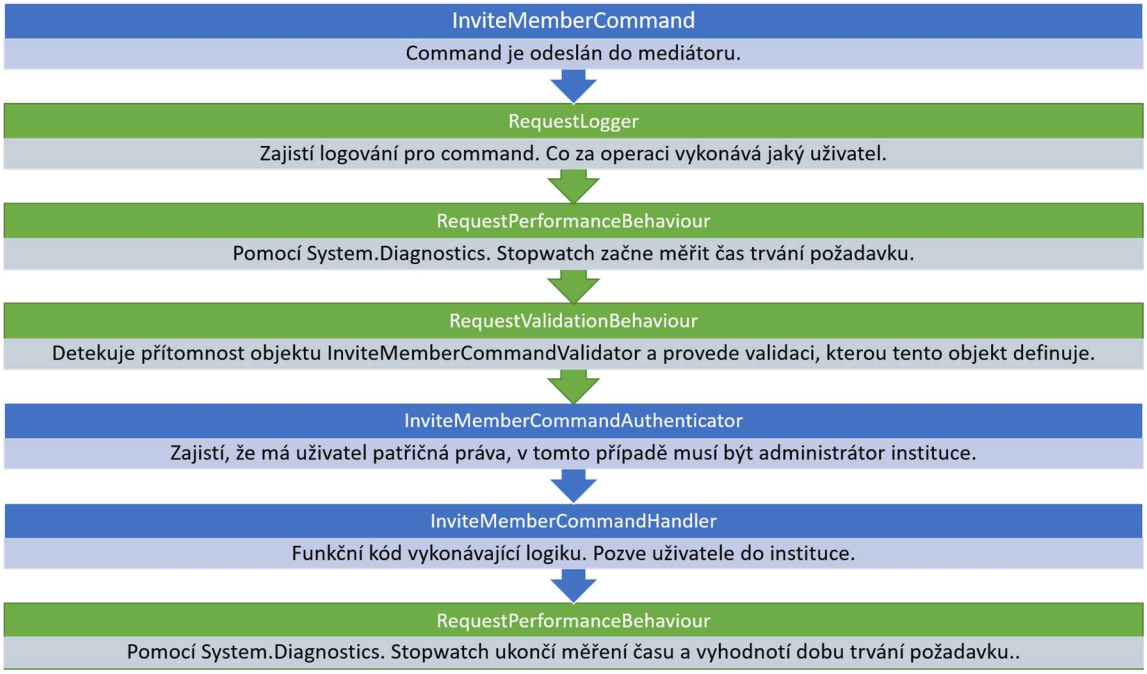
RequestPerformanceBehaviour
Implementuje rozhraní *IPipelineBehavior.* Spouští se před každým handlerem. Pomocí třídy *System.Diagnostics*. Stopwatch změří dobu trvání každého požadavku, který je v aplikaci zpracováván. Měření ukončí, jakmile handler dokončí práci. Pokud je čas větší než konstanta definovaná v konfiguraci, zapíše do logu varování.

RequestValidationBehavior
Implementuje rozhraní *IPipelineBehavior.* Rozpozná všechny validátory k právě spuštěnému commandu nebo query. Spustí validaci a výstup z ní shromáždí. Pokud došlo k chybě, sestaví strukturovanou odpověď, kterou zobrazí uživateli. Zajišťuje, že se validace skutečně provede, než se spustí handler.

TokenRevocator
Implementuje rozhraní IRequestPostProcessor. Stará se o zneplatnění všech obnovovacích tokenů uživatele, pokud dojde k jedné z následujících událostí. Změna hesla (*ChangePasswordCommand*). Resetování hesla (*ApplyResetPasswordCommand*).

Mediátor
Instance handlerů se nevytvářejí přímo. Pokud chce prezentační nebo jiná vrstva zavolat command nebo query, musí jít skrz mediátor. Ten se postará, aby se před handlerem vždy spustila kontrola autorizace a validace. Zároveň detekuje všechny třídy, které implementují *IRequestPreProcessor, IPipelineBehavior nebo IRequestPostProcessor* a zajistí, aby byly rovněž volány ve správném pořadí. Tento přístup umožňuje mít všechny součásti oddělené a zapouzdřené v objektech a celkově tak zlepšuje udržitelnost celé aplikace.

Průběh volání
Pro lepší ilustraci principu fungování aplikační vrstvy detailně rozvedu, jak se zpracovává jeden z požadavku. Konkrétně InviteMemberCommand (Obrázek 3-4 InviteMemberCommand průběh).



Obrázek 3-4 InviteMemberCommand průběh

3.3.7 Mapovací vrstva

Lhostejno zda je aplikace monolitická nebo rozdělená na několik mikro služeb. Nikdy by neměla vracet identický objekt, který získá z databáze nebo jej tam takto vkládat. Vždy by se mělo provádět mapování a zobrazovat uživateli jen ta data, která má vidět. To samé při vkládání dat do aplikace. Opět by se mělo provést mapování na systémové entity, a ne je chtít po uživateli. Jednak je zobrazování celých entit nebezpečné, někdo by mohl například vidět hash hesla uživatele, ale současně je to i neefektivní. Při zobrazování velkého množství dat není nutné vědět kompletní informace o daném záznamu, uživatel si o ně může proaktivně požádat. Přenese se tak méně dat a celá aplikace se mnohonásobně zrychlí. Mapování je samo o sobě poměrně nezajímavá a repetitivní činnost, ale při větším množství entit v systému nebo při snaze o vytvoření udržitelného systému se vyplatí řešit i tuto problematiku netriviálně a sofistikovaně.

Dává smysl nechat mapování řešit jednu samostatnou vrstvu aplikace. Mělo by být definováno na jednom místě a znovupoužito napříč aplikací tam, kde je potřeba. Pro každou entitu existuje DTO, na které ji lze mapovat. Zároveň jiné DTO zase slouží pro vytvoření nebo úpravu entity. Transformovat data z jedné entity do jednoho DTO není nijak složité, lze snadno vyřešit implementací jedné mapovací funkce a na všech místech ji volat. Avšak problém nastane ve chvíli, kdy se snažíme mapovat jednu kolekci na druhou, zejména když jednu kolekci získáváme přímo z databáze. Vrstva přistupující k databázi by neměla vědět o vrstvě zajišťující mapování. Navíc pokud budeme mapovat pomocí funkce, stejně bychom z databáze museli získat všechna data, protože dopředu nevíme, která budeme potřebovat. To je ovšem extrémně neefektivní. A i kdybychom je znali některé operace s daty, jako například agregace nebo filtrování je stejně lepší vykonat přímo v databázi nativně a nikoli v paměti aplikačního serveru.

Řešení, které se mi v minulosti mnohokrát osvědčilo, je vytvořit popis mapování takový, aby se dokázal interpretovat jak v C#, tak v databázi odkud se data načítají. Postačí i pokud se interpretuje ve vrstvě zajišťující přístup k datům. Ke každé entitě je vytvořen mapovací profil, který obsahuje deklarativní popis mapování z jednoho objektu na druhý. Ten lze integrovat s mapovací vrstvou, která převede

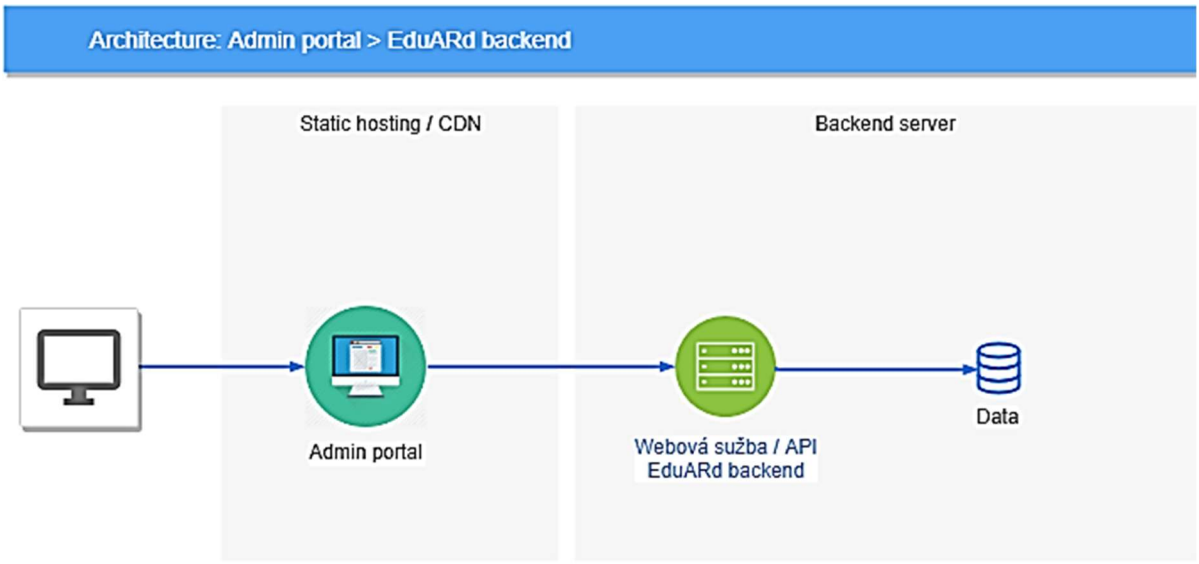
operace s daty na nativní instrukce pro databázi, pokud to půjde, a zbytek provede v paměti aplikačního serveru. Mapování navíc zůstane rozšiřitelné a znovupoužitelné.

3.3.8 Prezentační vrstva

V kontextu architektury zde prezentační vrstva neznamená uživatelské rozhraní, ale spíše sadu controllerů a endpointů, ke kterým ostatní aplikace přistupují. Webová služba (API) získává požadavky od klientské aplikace. Po validaci a základní autorizaci je následně předá aplikační vrstvě ke zpracování. Výsledek zpracování poté vrátí uživateli v odpovědi na daný požadavek. Autorizace probíhá v aplikační vrstvě i ve webové službě, avšak na jiné úrovni. Zatímco aplikační vrstva kontroluje, zda má uživatel příslušnou roli nebo práva, webová služba kontroluje pravost tokenu, jeho životnost a formát. Stejně tak validace v prezentační vrstvě se spíše stará o správnost formátu json. O správnost dat uložených v jsonu se opět stará aplikační vrstva. Rozhodl jsem se využít návrhový vzor MVC (model view controller). Každý endpoint je reprezentován jednou metodou controlleru. Každý požadavek, který se odešle na server, se namapuje na existující model.

3.3.9 Admin portál

Portál pro systémové administrátory není vrstva webové služby. Jedná se o oddělenou webovou aplikaci, kterou lze hostovat samostatně. Veškerá aplikační logika se bude obsluhovat skrz API, které vystavuje webová služba. Komunikuje skrz definované rozhraní stejně jako ostatní dílčí aplikace projektu EduARd. Neobsahuje žádná zadní vrátka k neveřejné funkcionalitě. Veškeré požadavky, které admin portál odesílá na webovou službu, budou validovány a ověřovány stejně jako by pocházely od jakékoli jiné aplikace. Portál tedy bude ve své podstatě jen přenášet data tam a zpět a vykreslovat je. To umožní vyhnout se renderování na serveru. Celá aplikace tak může být hostována staticky. To znamená, že uživatel si pouze stáhne výchozí soubor index.html a následně už s hostingovým serverem nemusí komunikovat. Vztah mezi portálem pro systémové administrátory, webovou službou a databází je znázorněn v diagramu (Obrázek 3-5 Front end - backend - diagram).



Obrázek 3-5 Front end - backend - diagram

3.4 TECHNOLOGIE A PLATFORMA

Řešení jsem se rozhodl vybudovat na platformě .Net Core. Jedná se o open source platformu od společnosti Microsoft. Přestože její předchůdce .Net Framework, byl známý pro vazbu na operační systém Windows, na jiném operačním systému nativně nefungoval, je .Net Core multiplatformní, podobně jako například Java. Jediná potřebná věc ke spuštění je .Net Core Runtime, což je ekvivalent Java Runtime Environment. Ten lze zadarmo stáhnout a nainstalovat. Kromě toho se dá aplikace sestavit

i jako self-contained. Znamená to, že runtime je její součástí a lze jí tedy spudit i na zařízení, kde není nainstalován. Nicméně v tomto případě je nutné cílovou platformu dopředu znát a použít pro to vhodný přepínač při kompilaci.

3.4.1 Webová služba

Pro webovou službu jsem zvolil framework ASP.Net Core, který je vybudován nad .Net Core. Opět oproti jeho předchůdci ASP.Net, který byl spíše robustním řešením, je ASP.Net Core kompaktní a modulární řešení. Umožňuje běh aplikace buď na webovém serveru nebo samostatně, to znamená, že lze aplikaci sestavit tak, že si nese malý webserver s sebou. Lze ji tedy jednoduše spustit jako službu na jakémkoli stroji. To usnadní testování vývojářům a zároveň minimalizuje nároky na nasazení do produkčního prostředí. Podobně jako node.js je ASP.Net Core asynchronní. Veškeré operace, které například využívají přístup k databázi, jsou neblokující a nezpomalují tak aplikaci.

Každý požadavek je zpracováván sérii komponent, kterým se říká middleware. Ty se mohou rozhodnout, zda předají požadavek dále ke zpracování nebo jeho zpracování zastaví. Zároveň mohou vykonat nějakou proceduru před i potom co předají požadavek dál. Middlewary se nastavují při startu aplikace v konfigurační třídě s názvem Startup. Každý požadavek má svůj vlastní kontext, ke kterému lze přistoupit pomocí objektu *HttpContext*. (7)

3.4.2 Admin portál

Pro Admin portál jsem se rozhodl použít Vue.js. Jedná se o framework pro SPA[10]. Vybral jsem si ho hlavně kvůli jeho vývojářský nástrojům, podpoře pro binding a jednoduché syntaxi. Umožní skládat stránku z komponent, které se definují pomocí tří prvků. Template, script a style. Template slouží pro definici html, které má být vykresleno v prohlížeči. Script obsahuje obslužný JavaScript pro ovládání formulářů a manipulaci s daty. Ve Style se nachází css, které umožňuje upravovat vzhled stránky. Lze jej nastavit globálně nebo pouze lokálně pro danou komponentu. Má nativní podporu pro populární framework Bootstrap, který jsem se rozhodl využít k vytváření vzhledu jednotlivých komponent.

3.4.3 Open API definice

Pro tvorbu Open API definice jsem zvolil nástroj Swagger. Je to jedna z implementací standardu Open API. Jedná se o open-source službu podpořenou souborem nástrojů, které umožňují vývojářům designovat, dokumentovat a následně konzumovat REST API. Důležitou součástí je Swagger UI, které vytvoří grafické prostředí a umožňuje prohlížet dokumentaci a zároveň ji testovat bez potřeby nějakého externího nástroje, jakým je například Postman nebo curl. Swagger definici je možné vzít a vložit do nástroje Swagger Codegen, který umožňuje vygenerovat obslužný kód jak pro server, tak pro klienta a vývojáři se tak ve svých aplikacích mohou soustředit pouze na aplikační logiku. (8)

Swashbuckle

Pro řešení vybudované nad ASP.Net Core existuje knihovna, která se nazývá *Swashbuckle.AspNetCore.Swagger*. Analyzuje všechny controllery a jejich anotace (v C# se jim říká atributy) a pomocí nich vygeneruje Swagger definici. Zároveň umožňuje nastavit adresu URL, na které se uživatelům zobrazí uživatelské rozraní, kde mohou dokumentaci prohlížet a testovat API (9)

3.4.4 Databáze

Nástroj zajišťující komunikace s databází se nazývá Entity Framework Core. Jedná se opět o framework vybudovaný nad .Net Core. Funguje na principech ORM. Uživateli frameworku vystaví objekt, který se nazývá *DbContext*. Ten se chová jako UoW [11]a vystavuje jednotlivé kolekce (tabulky) jako repositáře

[10] Single page application – Webová aplikace, která dynamicky překresluje obsah své jediné stránky. Nedochází tak přerušení při navigaci a aplikace celkově působí dojmem jako by byla desktopová.

[11] Unit of Work - Zabraňuje nekonzistenci v databázi. Série operací se zapouzdří a vykoná se atomicky. Pokud v průběhu selže změny se vrátí do původního stavu.

pro základní CRUD operace. Zároveň umožňuje překládat dotazy napsané v nativním kódu přímo do SQL, dokonce zvolí správný SQL dialekt podle databáze, ke které je připojen. Kromě toho umožňuje evidovat historii schématu databáze pomocí tzv. migrací. Každá migrace se definuje pomocí dvou definic. Definice „up“ vždy upraví schéma databáze, pokud se ze starší migrace přechází na stávající migraci. Definice „down“ se spustí, pokud naopak přecházíme z aktuální migrace na starší. Je tedy vždy možné schéma databáze replikovat napříč jednotlivými technologiemi (MySql, MS Server, PostgreSQL). Mimo to se dá vždy vrátit k historický verzím, pokud by například bylo nutné nasadit starší verzi aplikace.

I v databázích se uplatňují dva směry, kterými lze vytvořit model pro ORM.

Code first
Nejprve se vytvoří třídy, které reprezentují entity. Poté se pomocí zdrojového kódu definují jednotlivé relace mezi nimi spolu s omezeními. Následně se vytvoří výchozí migrace, která popisuje výchozí schéma databáze. To se následně aplikuje na cílovou databázi. Tím vznikne nová databáze.

Database first
Jedná se o opačný přístup. Používá se, pokud již existuje databáze, nebo bylo schéma navrženo v softwaru pro tvorbu relačních databází. Pomocí procedury *scaffold dbcontext* se vytvoří jak třídy, omezení tak i jednotlivé relace mezi nimi. (10)

3.4.5 Autentizace a autorizace
K implementaci autentizačního mechanismu pomocí JWT využiji právě zmíněné middlewary. Požadavky, které nemají patřičná oprávnění se nedostanou ani ke controllerům a tím pádem vůbec nebudou zpracovány. Na klienta se vrátí http kód 401(unauthorized). Pokud požadavek příslušné oprávnění má, bude odeslán dalším middlewarům ke zpracování.

Pro ASP.Net Core existuje *Authentication Middleware*, který ověří každý požadavek. Aby správně fungoval je potřeba registrovat do Dependency Injection několik tříd, které zajišťují vytváření tokenu a ověření správnosti přístupových údajů uživatele.

JwtFactory
Slouží k vytváření JWT. Umožňuje vytvořit JWT pro uživatele, který v těle obsahuje data o uživateli a jeho práva.

TokenFactory
Zajišťuje generování náhodných tokenů. Využívá knihovnu *System.Security.Cryptography*, která má mnohem větší entropii než Systém.Random a je tedy mnohem vhodnější a bezpečnější pro tento účel.

JwtTokenValidator
Umožňuje validovat již expirované přístupové tokeny. Využívá se hlavně při obnově pomocí obnovovacího tokenu. Nelze jít přes standardní validační middleware, neboť ten by token okamžitě zamítl, právě z důvodu expirace.

InMemoryRefreshTokenService
Uchovává obnovovací tokeny pro jednotlivé uživatele přímo v paměti.

Schéma autentizace
Aplikace bude využívat tokeny typu Bearer. Přístupový token se bude předávat v hlavičce *Authorization*.

> *Authorization:*
> *Bearer*
> *eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ*
> *SI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJ*
> *f36POk6yJV_adQssw5c*

Nakonfigurovaný middleware ověří správnost podpisu tokenu. Pokud je token platný a nevypršela mu životnost, proces pokračuje a ověří se, zda má uživatel příslušná práva. Pokud tokenu platnost vypršela, je navrácen http kód 401 (unauthorized) a hlavička odpovědi osahuje *Token-Expired: true*. Veškeré endpointy, které obsahují atribut *Authorize* budou vyžadovat, aby byl uživatel autentizován.

Autorizace
ASP.Net Core umožňuje jednoduchou implementaci autorizačních pravidel pomocí objektu, který se nazývá policy. Ke každému atributu Authorize, lze ještě přidat policy, která musí být splněna. Kdykoli je na controller odeslán požadavek, který nesplňuje nastavenou policy, je vyhodnocen jako neautorizovaný a nebude proveden. Policy může znamenat například nutnost nějaké role, věk větší než 18 let atd.

3.4.6 Mapování
K mapování jsem se rozhodl využít knihovnu AutoMapper. Nabízí funkcionální přístup k mapování a všechny mapovací úkony se dají rozepsat jako lambda funkce. Dokáže automaticky mapovat atributy, které mají stejný název a datový typ. Navíc dokáže mapovat i kaskádově. Pokud jeden objekt obsahuje atribut „InstitutionName“ a druhý objekt obsahuje atribut „Institution“, který má atribut Name, namapuje se. Kromě toho je kompatibilní s Entity Frameworkem Core, který používám pro přístup k databázi. Drtivá většina agregací a filtrování se provede přímo v databázi. Cekově se sníží množství kódu, které je potřeba k mapování, avšak při přejmenování atributů může mapování přestat fungovat. Nicméně to by měli odhalit přítomné unit testy.

Každá entita má svůj mapovací profil, kde je definováno, jak se mapuje na jednotlivá DTO, případně jak na entitu namapovat z nich. Uvažujeme-li mapovací profil jako zobrazení, AutoMapper vyžaduje, aby byl každý profil surjektivní. Při startu aplikace se registrují do dependency injection společně s knihovnou AutoMapper. Následně se zavolá metoda *AssertConfigurationIsValid*, která ověří správnost profilů. Nestane se tak, že se na chybu v mapování přijde až při používání aplikace. Faktickou chybu odhalí výjimka, která vznikne při startu.

3.4.7 Validace
V ideálním případě by validace měla probíhat jak na prezentační vrstvě, tak v aplikační vrstvě. Framework ASP.Net Core má vestavěnou validaci v podobě atributů (v Javě se nazývají anotace). Nicméně ta je značně omezená a neumožňuje například rozšířit o vlastní funkce nebo kritéria. Z toho důvodu jsem se rozhodl použít knihovnu Fluent Validation. Podobně jako AutoMapper je rozhraní spíše funkcionální, což přispívá čitelnosti validační logiky. Validace se kromě mediátoru dokáže zavolat i z prezentační vrstvy. Pomocí balíčku *FluentValidation.AspNetCore* se do dependency injection registrují validátory od poskytovatele knihovny a nahradí se tak výchozí validace. Webová služba pak ihned vrací chybovou hlášku, pokud obdrží nevalidní požadavek.

3.5 ROZHRANÍ WEBOVÉ SLUŽBY (API)

Následující sekce popisuje strukturu webového rozhraní (API), které následně konzumují klientské aplikace. Blíže specifikuje akce, které lze volat na controllerech. Jedná se zejména o controllery zajišťující správu uživatelů, přihlášení, autorizaci, generování aplikačních klíčů a správu institucí.

3.5.1 Admin controller

Umožňuje správu institucí, jejich zobrazení a editaci. Všechny endpointy jsou znázorněny pomocí UML diagramu. Soubor v příloze Diagramy/admin.png.

- Seznam všech institucí (GET /api/Admin)
- Detail instituce (GET /api/Admin/{id})
- Založení instituce (POST /api/Admin)
- Úprava údajů instituce (PUT /api/Admin/{id})

3.5.2 Auth controller

Zajišťuje primárně autentizaci a autorizaci. Umožňuje přihlášení, obnovu tokenu a vygenerování a správu aplikačních klíčů. Jednotlivé endpointy jsou znázorněny v UML diagramu. Soubor v příloze Diagramy/auth.png.

- Registrace uživatele se všemi právy k instituci (POST /api/AuthRegister) Slouží pouze k testování a při ostrém provozu nebude přístupný.
- Seznam aplikačních klíčů (GET /api/Auth/ApiKey)
- Detail aplikačního klíče (GET /api/Auth/ApiKey{jti})
- Vygenerovat aplikační klíč (PUT /api/Auth/ApiKey)
- Zneplatnit aplikační klíč (DELETE /api/Auth/ApiKey{jti})
- Přihlásit se (POST /api/Auth/SignIn)
- Obnovit přístupový token (POST /api/Auth/Refresh)

3.5.3 Institutions controller

Obsahuje endpointy, které umožňují administrátorovi instituce spravovat svoji instituci. Zobrazit detail svojí instituce, upravit název a kontaktní informace a spravovat členy instituce. Endpointy jsou zakresleny v UML diagramu. Soubor v příloze Diagramy/institutions.png.

- Detail instituce (GET /api/Institutions/current)
- Upravit instituci (PUT /api/Institutions/current)
- Spravovat role členů instituce (POST /api/Institutions/manage)
- Odebrat člena instituce (DELETE /api/Institutions/{email})
- Pozvat uživatele do instituce (POST /api/Institutions/Invite)
- Zrušit pozvánku do instituce (DELETE /api/Institutions/{id})

3.5.4 Users controller

Zajišťuje základní úkony společné pro všechny uživatele. Umožňuje změnu hesla, vyvolání procesu k obnovení hesla, přijetí pozvánky do instituce, zobrazení dat o aktuálním uživateli a dovolí uživateli opustit svoji instituci. Endpointy jsou zakresleny v UML diagramu. Soubor v příloze Diagramy/users.png.

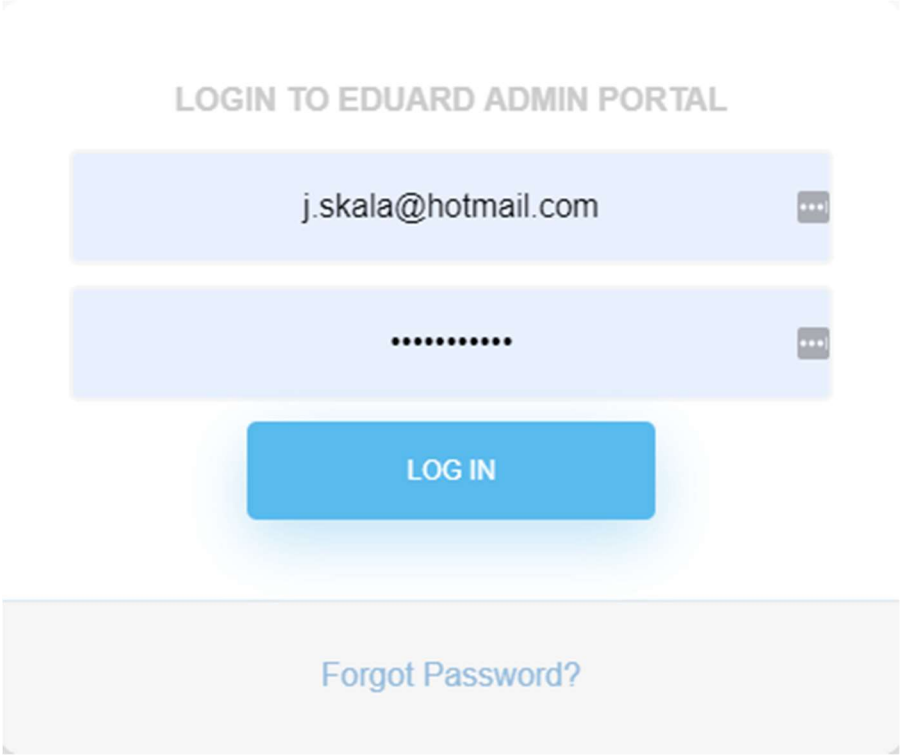
- Získání dat o aktuálně přihlášeném uživateli (GET /api/Users)
- Přijmutí pozvánky do instituce (POST /api/Users)
- Opuštění instituce, ve které se uživatel nachází (DELETE /api/Users/LeaveInstitution)
- (POST /api/ChangePassword)

3.6 Rozhraní Admin portálu

Následující sekce obsahuje popis rozhraní portálu pro systémové administrátory. Rozbor všech komponent, které byly vytvořeny jak po designové, tak funkční stránce.

3.6.1 Přihlášení

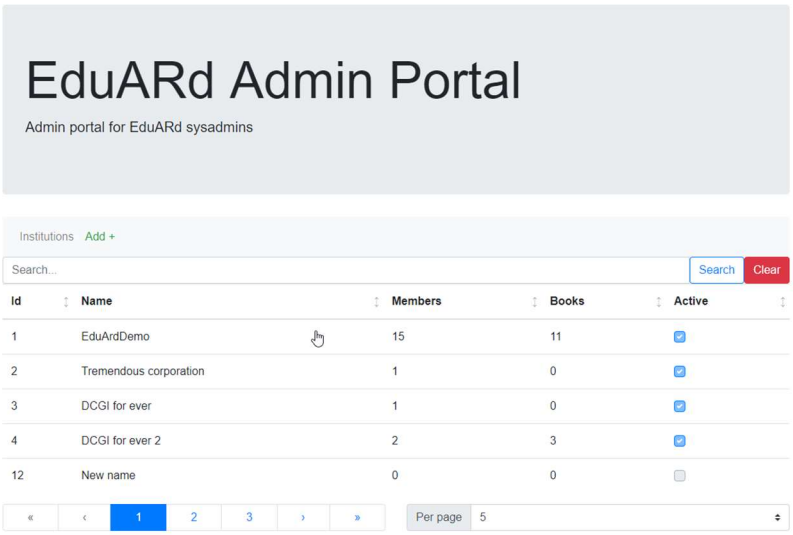
Při vstupu uživatele na administrační portál se zkontroluje, zda má uživatel aktivní token. Pokud ne, pokusí se aplikace o jeho obnovení pomocí obnovovacího tokenu. Pokud obnovovací token není přítomný nebo skončí požadavek chybou, je uživatel přesměrován na přihlašovací dialog (Obrázek 3-6 Dialog pro přihlášení).



Obrázek 3-6 Dialog pro přihlášení

3.6.2 Seznam institucí

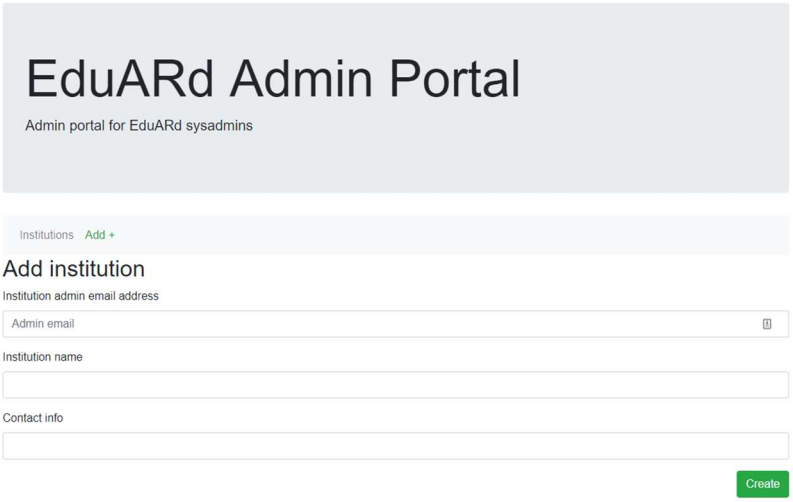
Výchozí stránka aplikace zobrazující aktuální seznam všech institucí, které se v systému EduARd nachází (Obrázek 3-7 Tabulka institucí). Zde může systémový administrátor prohlížet instituce a filtrovat je pomocí fulltextového vyhledávání. Kliknutím na řádek s institucí si může zobrazit její detail.



Obrázek 3-7 Tabulka institucí

3.6.3 Přidat instituci

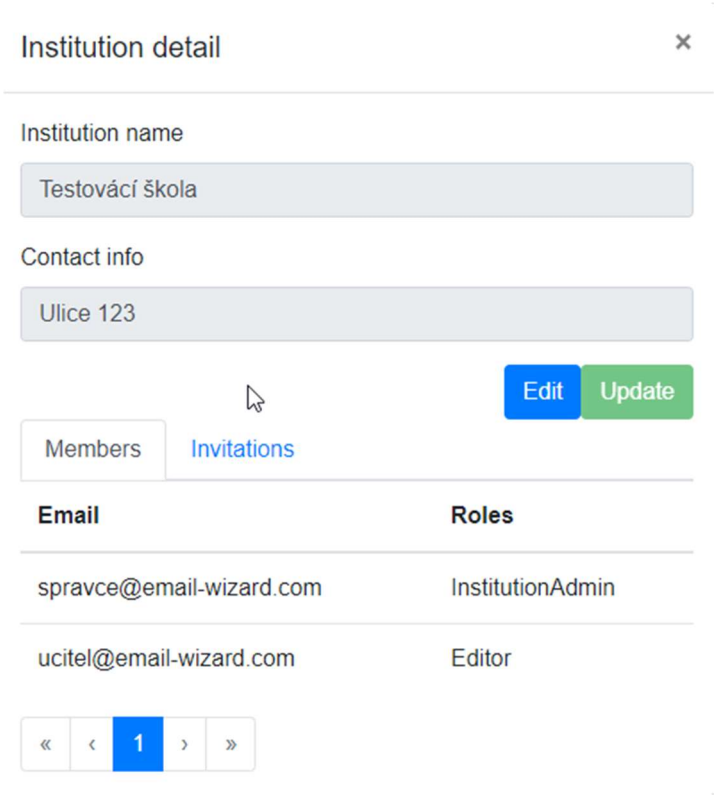
Formulář pro vytvoření instituce (Obrázek 3-8 Vytvořit instituci). Uživatel je povinen vyplnit název instituce, kontaktní údaje a validní email pro uživatele, který obdrží pozvánku do instituce a stane se jejím administrátorem.



Obrázek 3-8 Vytvořit instituci

3.6.4 Detail instituce

Při klepnutí na řádek s institucí v seznamu institucí se uživateli zobrazí detail instituce. Uvidí seznam uživatelů instituce a seznam pozvánek. Může editovat název instituce a kontaktní údaje pomocí formuláře (Obrázek 3-9 Detail instituce). Textová pole si uživatel nejprve odemkne tlačítkem „edit“. Předejde se tam nechtěným změnám, které mohl zapříčinit překlep.



Obrázek 3-9 Detail instituce

4 IMPLEMENTACE

Následující kapitola rozebere implementaci dílčích částí, na kterých jsem pracoval. Je popsáno testovací prostředí, do kterého byla nasazena webová služba a portál pro systémové administrátory. Každá aplikace běží na jiném prostředí, lze je tak škálovat nezávisle.

Kromě toho že bylo potřeba prezentovat výsledky implementace vedoucímu, chtěl jsem i ostatním studentům, kteří ve svých aplikacích využívali webovou službu (API), zajistit testovací prostředí. Jelikož pro CI/CD používám Azure Dev Ops, rozhodl jsem se vybudovat testovací prostředí právě v Microsoft Azure. Je třeba brát ohled na to, že v ostrém provozu nebude s nejvyšší pravděpodobností aplikace nasazena v této cloudové infrastruktuře. Není tedy možné ulehčovat si práci již hotovými službami v Azure, jakým je například Blob Storage, který by obstaral úložiště pro naučné stezky v podobě xml. Nemluvě o autentizaci, kterou lze v Azure také využít. Nicméně použití této cloudové platformy mnohonásobně urychlilo doručení aplikace k testování a k používání. Celkově byl proces doručení mnohem efektivnější a pohodlnější než ji pokaždé manuálně nasazovat na dedikovaný server nebo virtuální stroj.

4.1 WEBOVÁ SLUŽBA (API)

V současnosti existuje běžící prototyp aplikace na adrese https://deveduard.azurewebsites.net/. Je spuštěna v prostředí cloudu Microsoft Azure jako App Service. Původně běžela na sdílené infrastruktuře, ale tarif se musel navýšit na dedikovaný plán, neboť aplikace se uspávala a studenti tak museli při prvním požadavku po aktualizaci dlouho čekat.

Databáze běží ve službě Azure SQL. I ta se bude v ostrém provozu lišit. S největší pravděpodobností se využije PostgreSQL. Díky abstrakci nad databází, jež představuje perzistentní vrstva v podobě *DbContextu*, který poskytuje Entity Framework Core, lze oddělit snadno vyměnit databázový systém. Stačí jen vyměnit poskytovatele DbContextu. Tím může být právě *Npgsql.EntityFrameworkCore.PostgreSQL*, který umožní komunikaci libovolné PostgreSQL databáze s Entity Frameworkem Core. Poskytovatelů je vícero. Existuje dokonce podpore pro SqlLite nebo MySql. Výměna poskytovatele je v zásadě jednoduchá. Požadovaný poskytovatel se nainstaluje pomocí balíčkovacího systému NuGet a v konfigurační třídě *Startup* se zaregistruje do dependency injection příslušná třída.

Pro účely testování byl dočasně přidán endpoint „register“ aby si mohli ostatní vývojáři založit testovací účet (Obrázek 4-2 Registrace uživatele). V ostrém provozu tento endpoint nebude, neboť představuje bezpečnostní riziko.

Jakmile uživatel naviguje na výše zmíněnou webovou adresu, zobrazí se mu automaticky vygenerovaná dokumentace (Obrázek 4-1 Swagger UI). Obsahuje všechny endpointy, jaké druhy metod podporují i definic všech datových modelů, jak pro požadavky, které má posílat klientská aplikace, tak pro odpovědi, které server vrací. Kromě toho lze endpointy ihned vyzkoušet rovnou na webové stránce bez použití dalšího nástroje.

Lze testovat i včetně použití autentizace. Stačí zavolat endpoint pro přihlášení a získat přístupový token typu Bearer (Obrázek 4-3 Přihlášení). Ten následně vložit do kolonky authorize včetně předpony Bearer, která říká cílovému serveru, o jaký typ tokenu se jedná (Obrázek 4-5 Použití přístupového tokenu). Tato informace se musí uvádět, protože kromě typu Bearer existuje ještě token typu MAC, který však nebyl v této práci použit.

Využít Swagger jen na dokumentaci je plýtvání jeho potenciálem. Díky nástroji Swagger CodeGen lze přímo ze Swagger definice vygenerovat obslužný kód, který dokáže popisovanou službu volat. Je na výběr z celé plejády programovacích jazyků od C až po Scalu. Odkaz na Swagger definici je v záhlaví

Swagger UI (Obrázek 4-6 Swagger definice odkaz). Tento postup značně urychlí integraci jakékoli aplikace s touto webovou službou.

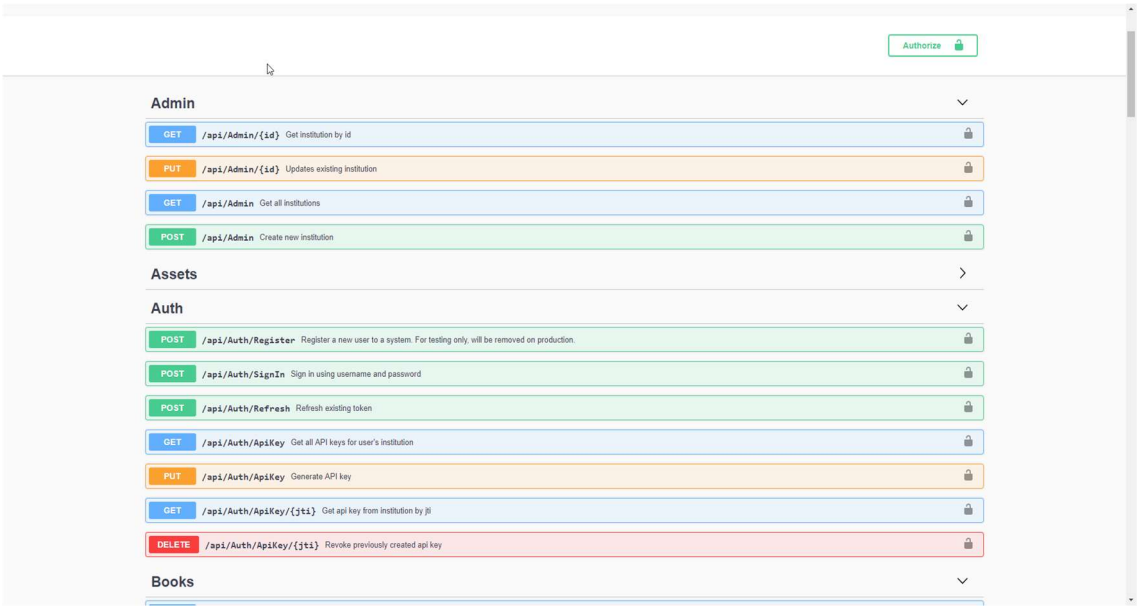
4.2 ADMIN PORTÁL

Současně s webovou službou běží v Azure i portál pro systémové administrátory na adrese https://eduard.z6.web.core.windows.net/. Jelikož se jedna o Single Page aplikace a nemá žádný kód, který by musel běžet na serveru, lze zvolit mnohem úspornější řešení než App Service. Je jím již jednou zmiňovaný Azure Blob Storage, který disponuje funkcionalitou hostovat statické weby, které obsahují pouze HTML, CSS nebo Javascript. Portál komunikuje se službou právě prostřednictvím REST API. Adresa webové služby (API) je uvedena v konfiguračním souboru helpers/constants.js.

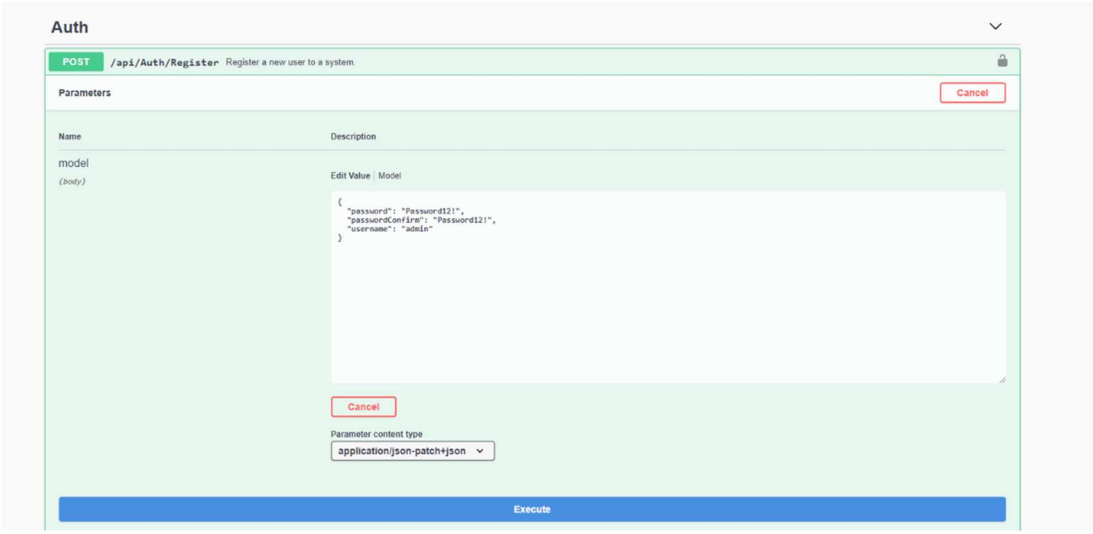
Framework Vue.js, který jsem se rozhodl použít má jedno úskalí. Po sestavení projektu se ve výstupních souborech kromě souborů s příponou .css a .js objeví pouze jeden jediný HTML soubor. Je to očekávané chování z definice Single Page aplikací. Problém nastává ve chvíli, kdy se vývojář rozhodne používat virtuální směrování. Jestliže je směrování na serveru, kde aplikace běží nastaveno špatně a server se snaží podle zadané adresy vrátit cílový soubor, virtuální směrování ve Vue.js nebude fungovat. Pokud například uživatel zadá adresu /strana/1 a server je nastaven špatně, bude se uživateli snažit odeslat soubor, který je ve složce wwwroot/strana/1 a nazývá se index. Žádný takový soubor tam ovšem není a dotaz skončí chybou 404. Tomu se dá zabránit, nastavíme-li, aby při chybějícím soubor server vrátil uživateli soubor wwwroot/index.html. Tím se spustí Vue.js aplikace a doplněk Vue Router přečte adresu, porovná ji se směrovací tabulkou a vykreslí uživateli příslušnou komponentu.

Analogický problém nastal i při použití Azure Blob Storage. Přestože aplikace neběží na webovém serveru, ale pouze v jakémsi kontejneru pro soubory, tak při navigaci pomocí URL adresy docházelo ke stejné chybě. Nicméně i zde pomohlo přesměrovat všechny dotazy končící chybou 404 na soubor index.html nastavení pole „Error document path“ v konfiguraci.

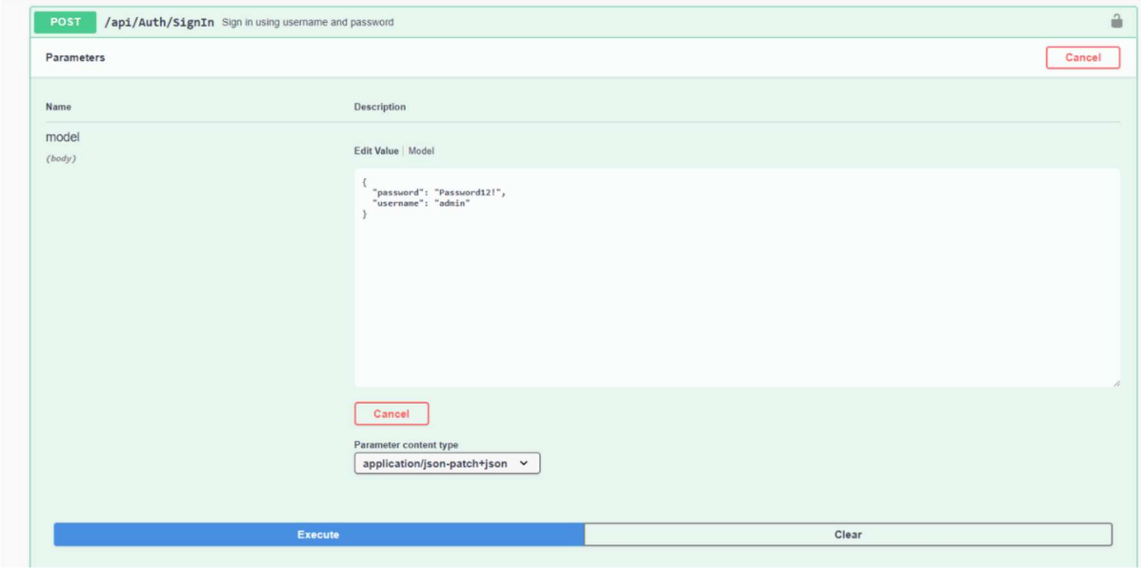
4.3 SCREENSHOTY



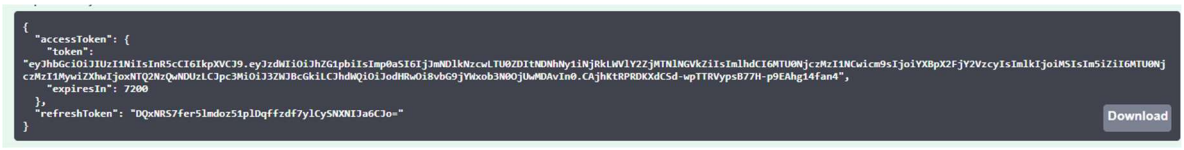
Obrázek 4-1 Swagger UI



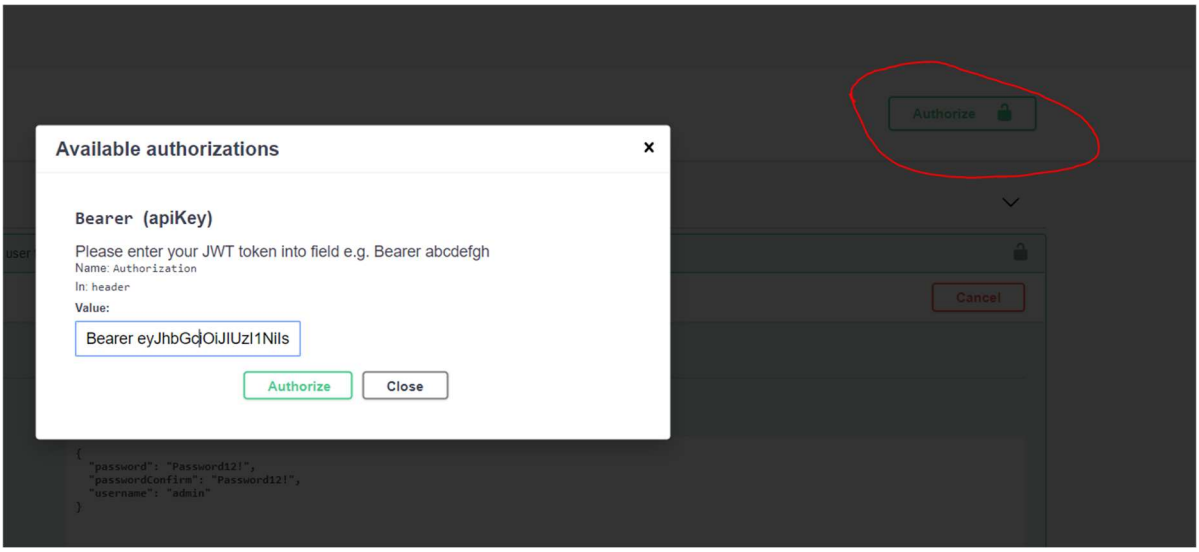
Obrázek 4-2 Registrace uživatele



Obrázek 4-3 Přihlášení



Obrázek 4-4 Odpověď s přístupovým a refresh tokenem



Obrázek 4-5 Použití přístupového tokenu

Obrázek 4-6 Swagger definice odkaz

5 TESTOVÁNÍ

5.1 UNIT TESTY

Pro unit testy jsem se rozhodl použít xUnit. Jedná se o open source testovací framework. Vybral jsem ho hlavně kvůli možnosti parametrizace jednotlivých testů. Zároveň umožňuje nastavit dependency injection pro každou skupinu testů pomocí objektu s názvem fixture. Snadno se tak vytvoří prostředí, ve kterém každý test běží izolovaně a je mnohem snazší nechat testy běžet paralelně, což velmi urychlí celý testovací proces. (11)

Většina aplikační logiky, kterou jsem tvořil je pokryta unit testy. Jmenovitě práce s entitami: Instituce, Uživatel aplikace, Pozvánka a Aplikační klíč. Dále pak autorizace pro entity Kniha, Příloha, Scénář a Příloha scénáře. Zároveň je samostatně testována i autorizační vrstva.

5.2 INTEGRAČNÍ TESTY

Pomocí knihovny *Microsoft.AspNetCore.Mvc.Testing* lze nakonfigurovat in-memory testovací server, který umí spustit aplikaci a emulovat databázi i webový server pouze za použití operační paměti. To umožní i testování na online build agentovi[12] a případně tak realizovat Continuous integration a Continuous deployment. (12)

Série testů, která vznikla už v ranné části implementace pokrývá všechny endpointy, kde se používá metoda GET. Otestují se tak všechny query. Navíc díky tomu, že se v integračních testech zpravidla používají stejné objekty jako při ostrém provozu, lze testovat, zda je správně nakonfigurována mapovací vrstva a že neselže ani dependency injection, například z důvodu chybějící implementace pro nějaké rozhraní.

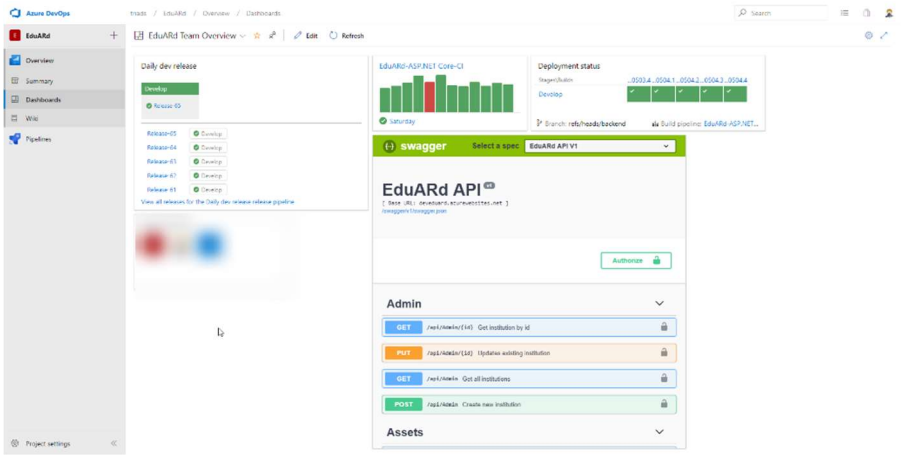
5.3 CONTINUOUS INTEGRATION, CONTINUOUS DEPLOYMENT

Vzhledem k tomu, že k sestavení webové služby (API) stačí pouze .Net Core SDK, lze jí tak jednoduše sestavit i na testovacím agentovi. Pro tento účel jsem vybral službu Azure Dev Ops od společnosti Microsoft. Tato služba je zdarma pro open source projekty a pro malé týmy (do 5 členů). Jelikož mám s touto službou bohaté zkušenosti a splňuji jedno z kritérií, rozhodl jsem se ji používat.

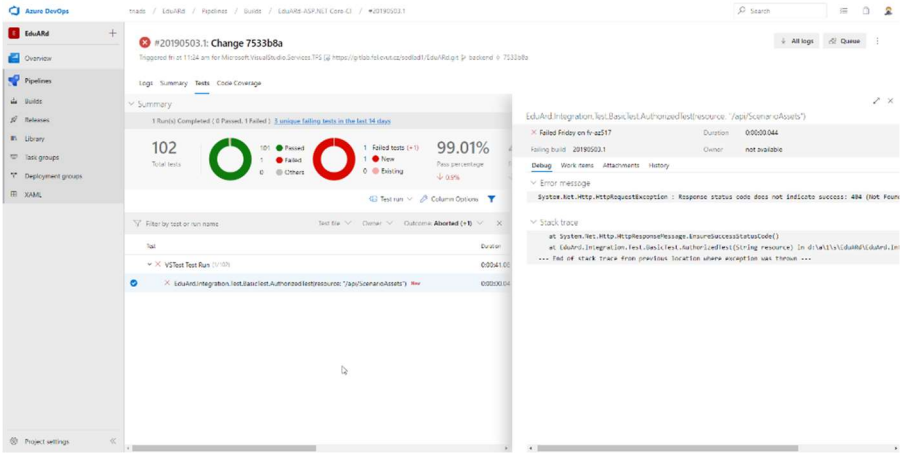
Agent nejprve stáhne všechny závislosti a knihovny, které se v aplikaci využívají a nainstaluje je. Poté se pokusí sestavit zdrojový kód. Výstupem této operace je spustitelný binární soubor. Podaří-li se sestavení následuje spuštění všech testů. Jakmile se aplikace úspěšně sestaví spustí se v Azure Dev Ops série kroků, která se nazývá „Release Pipeline“ a nasadí celou aplikaci na testovací prostředí. Testovací prostředí je detailněji popsáno v kapitole 4.

Celkový stav testů a nasazení je dobře vidět na informačním panelu (Obrázek 5-1 Informační panel. Odsud se lze prokliknout i na detail testu, který selhal (Obrázek 5-2 Detail testu).

[12] Program běžící na pozadí. Překládá zdrojový kód, který se nachází v repositáři, ke kterému je připojen. Používá se pro CI a CD.



Obrázek 5-1 Informační panel



Obrázek 5-2 Detail testu

6 ZÁVĚR

V této práci byla rozebrána problematika autentizace, autorizace, správa uživatelů a institucí systému EduARd. Zároveň integrace, komunikace a napojení na ostatní dílčí aplikace. Byly popsány a porovnány možnosti řešení dané problematiky. Od konzervativních a tradičních přístupů až po moderní architektonické trendy.

Na základě analýzy případů užití, entit a rolí v systému byly vybrány nejvhodnější metody řešení. Komunikace mezi dílčími aplikacemi bude probíhat pomocí http protokolu a standardu REST. API bude ve formě webové sužby a k jeho dokumentaci bude sloužit Open API definice. K jejímu zápisu se využije nástroj Swagger. Ačkoli se původně počítal s tím, že pomocí nástroje Swagger Codegen ostatní studenti vygenerují kód pro obsluhu webové služby, nebyl to většinový případ. Já sám jsem využíval jen část webové služby a pro obsluhu endpointů jsem raději kód napsal, aby byl celkově čitelnější a neobsahoval věci navíc.

Autentizace a autorizace je realizovaná prostřednictvím JWT, který ve většině případů umožní vymanit se z nutnosti udržování nějakého stavu na serveru a celý proces autentizace urychlit a umožnit snadné škálování. Přístup k databázi je realizován výhradně prostřednictví abstrakce a frameworku Entity Framework Core, aby bylo možné konkrétní databázi snadno migrovat nebo celkově změnit technologii. Celá architektura je koncipovaná tak, aby aplikační logika byla nezávislá na okolním prostředí a bylo snadné ji celou vyjmout a znovupoužít.

Přestože byl kompletně automatizován proces nasazení a testování, nepodařilo se odhalit všechny chyby a některé byly stejně objeveny až při používání webové služby. Primárně se jednalo o nemožnost volat testovací instanci webové služby (API) z prohlížeče. Bylo to zapříčiněno tím, že webové aplikace pro editaci obsahu a webová služba (API) byly hostované na jiných doménách. V hlavičce chyběl záznam *Access-Control-Allow-Origin: **, který by sdělil prohlížeči, že má požadavek povolit. Nicméně i to bylo opraveno.

6.1 NĚKTERÁ MOŽNÁ VYLEPŠENÍ

Od začátku je aplikace navržena tak, aby byla snadno rozšiřitelná a jednotlivé funkcionality byly odděleny, zapouzdřeny a otestovány. Díky použitému paradigmatu CQRS lze snadno přidat nový command nebo query a tím tak rozšířit funkcionalitu. Při přidání controlleru nebo endpointu se změna automaticky promítne do vygenerované dokumentace. Potenciál pro vylepšení je velký. Mimo jiné se jedná například o

- Přívětivější design portálu pro systémové administrátory a jeho lepší zobrazení na mobilních zařízeních.
- Podpora pro další poskytovatele identit. Umožnit uživatelům přihlašovat se například pomocí Facebooku nebo jejich školních účtu, pokud jejich škola vlastní Office 365 nebo G Suite.
- Vyjmout controllery, které řeší autorizaci a provozovat je v samostatné webové službě a usnadnit tak škálovatelnost.
- Plně podporovat standard oAuth2 a umožnit tak delegovaný přístup dalším aplikacím, které se budou chtít s webovou službou dále integrovat. Může se jednat například o aplikaci zajišťující sdílení obsahu.

7 SEZNAM OBRÁZKŮ

8 SLOVNÍK POJMŮ A ZKRATEK

Pojem / zkratka	Vysvětlení
CRUD	Zkratka pro Create Read Update Delete
ORM	Objektově relační mapování Zajišťuje automatickou konverzi dat mezi (relační) databázi a objekty z aplikační domény.
API	Application Programming interface Rozhraní, které je vystaveno pro komunikaci. Zapouzdřuje složitější funkcionality za jednodušší metody a procedury, typicky CREATE, READ, UPDATE, DELETE.
REST	Representational State Transfer Implementuje základní operace (CRUD) pomocí metod HTTP protokolu (GET, POST, PUT, DELETE).
Framework	Softwarový balík usnadňující programování. Má větší rozsah než knihovna. Typicky zapouzdřuje a řeší komplexní problém.
DTO	Data transfer object – Objekt, který pouze přenáší data mezi procesy nebo vrstvami aplikace. Nemá žádnou vnitřní funkcionalitu.
Inversion of Control (IoC)	Jedná se o návrhový vzor. Chod aplikace řídí framework. Odděluje rozhraní od implementace, jednotlivé vrstvy spoléhají spíše na rozhraní než na konkrétní implementaci.
Dependency Injection (DI)	Vkládání závislostí. Jedná se o implementace návrhového vzoru IoC. Pokud chceme v aplikaci vytvořit instanci třídy, nepoužíváme konstruktor, ale zavoláme kontejner, který už ví, jaké závislosti třída potřebuje.
JSON	JavaScript Object Notation Jedná se o jednoduchý datový formát. Využívá se pro přenos strukturovaných dat. Oproti XML neobsahuje silné typové definice, díky tomu je přenos dat menší.
UoW	Unit of Work Zabraňuje nekonzistenci v databázi. Série operací se zapouzdří a vykoná se atomicky. Pokud v průběhu selže změny se vrátí do původního stavu.
Build agent	Program běžící na pozadí. Překládá zdrojový kód, který se nachází v repositáři, ke kterému je připojen. Používá se pro CI a CD.
Continuous integration (CI)	Integrace jednotlivých částí systému mezi sebou. Ideálně automaticky.
Continuous deployment (CD)	Funkční části systému jsou automaticky nasazeny do produkce, jakmile jsou otestovány.
SDK	Software development kit – soubor nástrojů pro vývoj software.
SPA	Single page application – Webová aplikace, která dynamicky překresluje obsah své jediné stránky. Nedochází tak přerušení při navigaci a aplikace celkově působí dojmem jako by byla desktopová.

Refactoring	Vylepšení vnitřní struktury kódu bez vlivu na vnější chování.

9 SEZNAM LITERATURY A POUŽITÝCH ZDROJŮ

1. Sedláček, David. *Přihláška projektu EduARd.* 2018.

2. Ralphson, Mike. OpenAPI-Specification. *Github.* [Online] [Citace: 17. 1 2019.] https://github.com/OAI/OpenAPI-Specification.

3. SOAP Specifications. *W3C.* [Online] [Citace: 7. 5 2019.] http://www.w3.org/TR/soap/.

4. Nitzsche, Jörg, Lessen, T. van a Leymann, Frank. WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities.
[Online] 2008. [Citace: 7. 5 2019.] http://dblp.uni-trier.de/db/conf/iciw/iciw2008.html.

5. Nickel, Jochen. *Mastering Identity and Access Management with Microsoft Azure.* Birmingham, UK : Packt Publishing, 2016.

6. Fowler, Martin. CQRS. [Online] [Citace: 17. 1 2019.] http://martinfowler.com/bliki/CQRS.html.

7. Roth, Daniel, Anderson, Rick a Luttin, Shaun. aspnetcore-2.2. *docs.microsoft.com.* [Online] 16. 11 2018. https://docs.microsoft.com/cs-cz/aspnet/core/?view=aspnetcore-2.2.

8. Swagger API. [Online] [Citace: 17. 1 2019.] http://swagger.io/.

9. Boyer, Shayne a Addie, Scott. getting-started-with-swashbuckle. *docs.microsoft.com.* [Online] 18. 12 2018. https://docs.microsoft.com/cs-cz/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-2.2&tabs=visual-studio.

10. Freeman, Adam. *Your First Entity Framework Core Application.* Berkeley : Apress, Berkeley, CA, 2018.

11. xunit. *github.* [Online] 2019. https://xunit.github.io/.

12. Latham, Luke a Smith, Steve. aspnet/core/test/integration-tests. *docs.microsoft.com.* [Online] 11. 1 2019. https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-2.2.

10 PŘÍLOHA

Součástí tištěné verze této práce je též DVD, které obsahuje následující adresáře:

- AdminPortal – zdrojový kód portálu pro systémové administrátory
- Backend – zdrojový kód backendu projektu EduARd
- Diagramy – UML diagramy rozhraní API
- Dokumentace – soubor README s popisem spuštění a nasazení projektu