FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Lokalizační platforma se zaměřením na mobilní aplikace
Student:	Vojtěch Hořánek
Vedoucí:	Ing. Tadeáš Sosín
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem této práce je vytvořit prototyp lokalizační platformy s důrazem na propojení s externími službami (Firebase Database apod.) pro uložení textů a primárním zaměřením na překlad mobilních aplikací.

Proveďte analýzu případných konkurenčních řešení. Proveďte analýzu funkčních i systémových požadavků a na jejich základě vytvořte low-fidelity prototyp. Navrhněte nástroje a jiné produkty třetích stran potřebné pro realizaci aplikace a vnitřní architekturu aplikace. Navrhněte způsob testování a následného nasazení do produkce. Navrhněte prostředí pro podporu provozu aplikace, které umožní její další rozvoj a podporu stávajících uživatelů. Po dohodě s vedoucím práce realizujte prototyp frontendu a backendu této platformy.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 17. ledna 2024 v Praze.

F8

Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Lokalizační platforma se zaměřením na mobilní aplikace

Vojtěch Hořánek

Květen 2024
Vedoucí práce: Ing. Tadeáš Sosín

Poděkování / Prohlášení

Chtěl bych poděkovat vedoucímu práce, Ing. Tadeáši Sosínovi, za cenné rady poskytnuté při vývoji i psaní bakalářské práce. Zároveň za jeho plnou důvěru. Poděkování patří také mé rodině a přátelům, kteří mi při studiu poskytnuli nezbytnou podporu.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

.....

Abstrakt / Abstract

Bakalářská práce se věnuje tvorbě platformy usnadňující překlad softwaru. Určená je nejen vývojářům, ale i pro překladatelům. Před implementací byla provedena rešerše existujících služeb a z ní vyvozeny požadavky na nové řešení. Záměr celé platformy byl ulehčit lokalizaci především mobilních aplikací, nicméně je možno ji využít i při vývoji jiného softwaru. Předložené řešení je rozděleno na backend a frontend, obě využívající technologii Kotlin Multiplatform, která umožnila sdílení tříd entitních modelů. Komunikace mezi nimi probíha pomocí REST API. Byla využita služba Firebase Authentication pro autentizaci uživatelů. V roli databázového úložiště posloužila NoSQL dokumentová databáze Firebase Firestore. Logování a automatické hlášení chyb probíha v platformě Sentry. Výsledkem je veřejně dostupná webová aplikace s názvem „Ribbon“, ve které si uživatelé mohou spravovat překlady textů v jejich softwaru, zvát překladatele a využít funkcionalit importu a exportu frází ve formátech Android XML a Apple Strings. Hlavním přínosem této práce je možnost integrace přímo do softwaru vyvíjeného uživatelem, speciálně do mobilní aplikace, pomocí zmíněného API.

Klíčová slova: lokalizační platforma; Kotlin Multiplatform; Firebase; překlad mobilních aplikací; JetBrains Compose.

The bachelor thesis focuses on creating a platform that facilitates software localization, targeting both developers and translators. Prior to implementation, research was conducted on existing services, from which requirements for the new solution were derived. The aim of the entire platform was to streamline the localization process, primarily for mobile applications, although it can also be utilized in the development of other software. The proposed solution is divided into backend and frontend, both utilizing the Kotlin Multiplatform technology, enabling the sharing of entity model classes. Communication between them is facilitated through a REST API. Firebase Authentication service was utilized for user authentication, while NoSQL document database Firebase Firestore served as the database storage. Logging and automatic error reporting are handled by the Sentry platform. The outcome is a publicly accessible web application named „Ribbon“, where users can manage translations of texts in their software, invite translators, and utilize import and export functionalities for terms in Android XML and Apple Strings formats. The main contribution of this work is the possibility of direct integration into the user-developed software, particularly mobile applications, using the aforementioned API.

Keywords: localization platform; Kotlin Multiplatform; Firebase; mobile app translation; JetBrains Compose.

Title translation: A localization platform with a focus on mobile applications

Obsah /

Obrázky /

Výpisy kódu / Tabulky

Kapitola 1
Úvod

Lokalizace, zejména překlad softwaru, je jeden ze základních předpokladů pro jeho rozšíření na globální trh. Pokud si uživatelé mohou software přizpůsobit do jejich jazyka, mají z používání lepší zážitek a jsou si při práci v něm jistější [1, 2].

Vývojáři jsou tak postaveni do role, kdy musí navrhnout software s ohledem na lokalizaci pro různé trhy a regiony. Jeden z hlavních pilířů tohoto procesu je překlad textů zobrazovaných uživatelům. Lokalizace a překlad jsou často mylně zaměňovány nebo považovány za synonyma. Tato bakalářská práce se bude zaměřovat výhradně na překlady a jejich usnadnění, nikoliv na celkovou lokalizaci. Do ní totiž patří i přizpůsobení uživatelského rozhraní, měrných jednotek, právních a regulačních požadavků, a dokonce i úpravy barev či obrázků. Různé společnosti mohou barvy vnímat rozdílně [3]. Tyto úpravy se provádí přímo v kódu softwaru, proto je mimo rozsah této práce o nich pojednávat.

Správa překladů nemusí být jednoduchá. Na projektu většinou pracuje více překladatelů a sjednocení textů není lehký úkol. Software bývá dostupný pro více platforem, které používají vlastní formát pro definici překladů. Často se tak nelze vyhnout převodu mezi nimi. Tyto úkony se zpravidla musí provádět před každým vydáním nové verze softwaru a při jejich manuální realizaci by mohlo dojít k nadbytečným prodlevám a zbytečným chybám. Minimalizace a automatizace těchto procesů je proto nadmíru žádaná.

V současné době již existují platformy, které tento problém řeší a více se jim budu věnovat v kapitole 3. Cílem této bakalářské práce je navrhnout a vyvinout prototyp obdobné lokalizační platformy, která usnadní překladatelům překlad textů a vývojářům integraci při tvorbě softwaru, zejména mobilních aplikací. Výstupem práce bude prototyp webového frontendu spolu s odpovídajícím backendem.

Motivace pro tuto práci byla integrační omezení existujících řešení, která nesplňovala požadavky na jednoduchost použití v existujícím projektu a vysoké náklady související s použitím těchto platforem.

Kapitola 2
Cíl práce

Cílem bakalářské práce je tvorba prototypu platformy pro překladatele a vývojáře usnadňující lokalizaci softwaru a speciálně mobilních aplikací.

Nejdříve se zaměřím na rešerši existujících služeb řešící daný problém, ze které vyvstanou požadavky nového řešení. Navrhnu a popíši technologie vhodné při implementaci, diskutuji vnitřní architekturu systému s ohledem na škálovatelnost a ze všech poznatků navrhnu datové a databázové modely. Zabývat se budu i nástroji třetích stran, vhodných pro podporu chodu platformy.

Součástí předloženého řešení bude webové rozhraní pro uživatele (frontend), které umožní překládat texty v projektech, a serverová část služby sloužící k obsluze webu (backend).

Implementovanou aplikaci nasadím do produkčního prostředí, otestuji automatickými testy a budou provedeny i testy na uživatelích.

V závěru uvedu možná vylepšení a návrhy pro budoucí vývoj.

Kapitola 3
Analýza existujících řešení

Existujících platforem pro překlad softwaru je v dnešní době na trhu veliké množství a z důvodu rozsahu nemohu v této práce všechny porovnat. Proto jsem zvolil jen nejpopulárnější a nejlépe hodnocené služby, nalezené na stránkách sloužících k vyhledávání aplikací a informací o jejich popularitě [4, 5, 6]. V kategorii „Software Localization“ jsem tedy z více než zhruba šedesáti uvedených vybral pět a porovnal jejich funkce, design, ale i možnosti integrace při vývoji softwaru, potažmo přímo mobilní aplikace. Při porovnání cen a plánů uvádím měsíční sazby při ročním předplatném.

3.1 Crowdin

Crowdin je jedna z nejpopulárnějších platforem pro překlad která, vznikla v roce 2009 [7] a uživatelé G2 ji hodnotí 4,6 z 5 hvězd [8].

Než uživatel začne s překladem, je třeba se registrovat a vytvořit projekt. Po jeho vytvoření a výběru podporovaných jazyků lze začít s nastavením frází. Fráze/klíče jsou v projektu rozřazeny do jednotlivých „zdrojů“ (souborů). Uživatel může zvolit nahrání existujících frází nebo jednoduše vytvořit prázdný zdroj a později do něj přes rozhraní editoru přidat nové klíče.

Na stránce překladů jsou ke každému klíči zobrazeny relevantní informace o kontextu a překladu ve výchozím jazyce. Uživatel může přeložit text ručně, nebo využít některé z nabízených návrhů. Ty jsou generovány umělou inteligencí a na základě předchozích překladů.

Do projektu lze přizvat uživatele s různou úrovní oprávnění, jež jsou přiřazeny k pěti rolím. „Manager“ má kontrolu nad celým projektem. „Developer“ může spravovat fráze, vytvářet překlady a používat integrace. „Translator“ překládá fráze. „Proofreader“ dělá korekturu a také překládá fráze. „Language Coordinator“ spravuje tým překladatelů. Tyto role se navíc nevylučují, lze je kombinovat. V případě potřeby je projekt otevřen pro veřejnost. V tu chvíli může kdokoliv nahlédnout do jeho obsahu a za určitých podmínek se může připojit a aktivně spolupracovat na překladu.

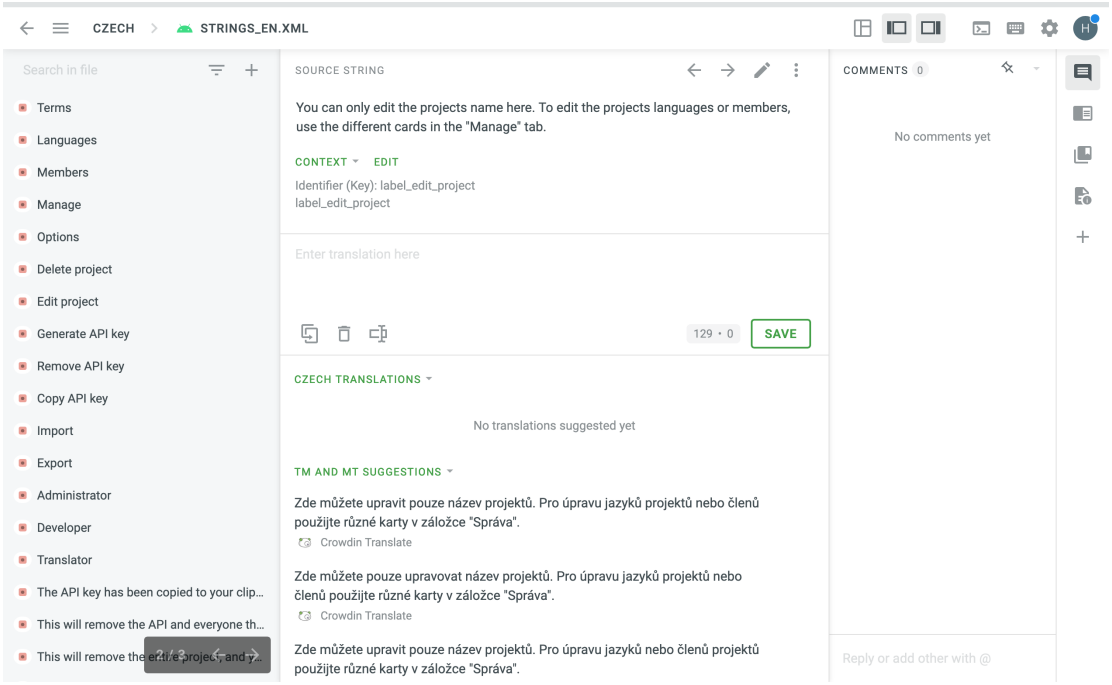
Poskytované integrace jsou zde opravdu bohaté, Crowdin umožňuje například propojení s GitHubem[1], GitLabem[2], ale i pro nás zajímavou možnost spojení s mobilní aplikací. Mimo klasický export do formátu `xml` pro Android či `strings` pro iOS lze využít i SDK pro oba systémy, umožňující aktualizaci textů bez nutnosti vydání nové verze celé aplikace. Vývojáři tak mohou doručit aktualizace textů kdykoliv a okamžitě [9].

Crowdin má různé cenové plány s různě omezenou funkcionalitou. Základní plán „Free“ je zdarma, nicméně poskytujeme místo pouze pro 60 000 uložených slov (tj. počet slov ve veškerém uloženém textu, odlišné od počtu frází – počet klíčů určených k překladu), jen jednu integraci a explicitně zakazuje použití zmíněného SDK. Již první placený plán „Pro“ (49 €) přidává možnost propojení s SDK. S dražšími plány

[1] Vývojářská platforma pro správu zdrojových kódu: `https://github.com/`
[2] Obdoba GitHubu: `https://about.gitlab.com/`

„Team“ (146 €), „Team+“ (437 €) a „Bussines“ (cena neuvedena) jsou odemčeny nové možnosti a funkce [10].

Jako hlavní nevýhodu tohoto řešení vidím malou kapacitu slov pro bezplatný plán a při tom nemožnost nastavit členům jinou roli než „Manager“ (platí jen pro soukromé projekty). Ačkoli působí design aplikace moderně, není dle mého vhodný pro uživatele, kteří nejsou technicky zdatní.



Obrázek 3.1. Překlad frází pomocí platformy Crowdin.

3.2 Phrase Strings

Phrase, dříve Memsource, je další zástupce z nejpoužívanějších řešení pro lokalizaci softwaru. Já se v tomto porovnání zaměřím pouze na jednu z mnoha služeb poskytovaných pod jeho záštitou, konkrétně na Phrase Strings. To proto, že je svou funkcionalitou nejvíce podobná ostatním porovnávaným řešením. Za zmínku ale stojí její výhoda propojenosti s ekosystémem Phrase [11].

Stejně jako Crowdin i zde se uživatel nejprve musí registrovat a následně vytvořit projekt. Postup je podobný, zvolí si název projektu, primární zaměření a vybere podporované jazyky. Následně je uživatel vyzván k nahrání souboru s existujícími texty, jenž budou určeny k překladu do dalších jazyků. Přidávání nových klíčů je podmíněno výběrem souboru, ke kterému patří, obdobně jako v předchozí platformě.

Jedním z klíčových nástrojů je funkce „Jobs“. Jde o zadání práce na skupině frází, které je potřeba přeložit. Při vytváření zadání se přiřadí k požadovaným jazykům jeden či více členů projektu a lze stanovit i termín dokončení práce. Kromě toho je možné překládat i klasickým způsobem bez využití „Jobs“ přímo ve webovém rozhraní.

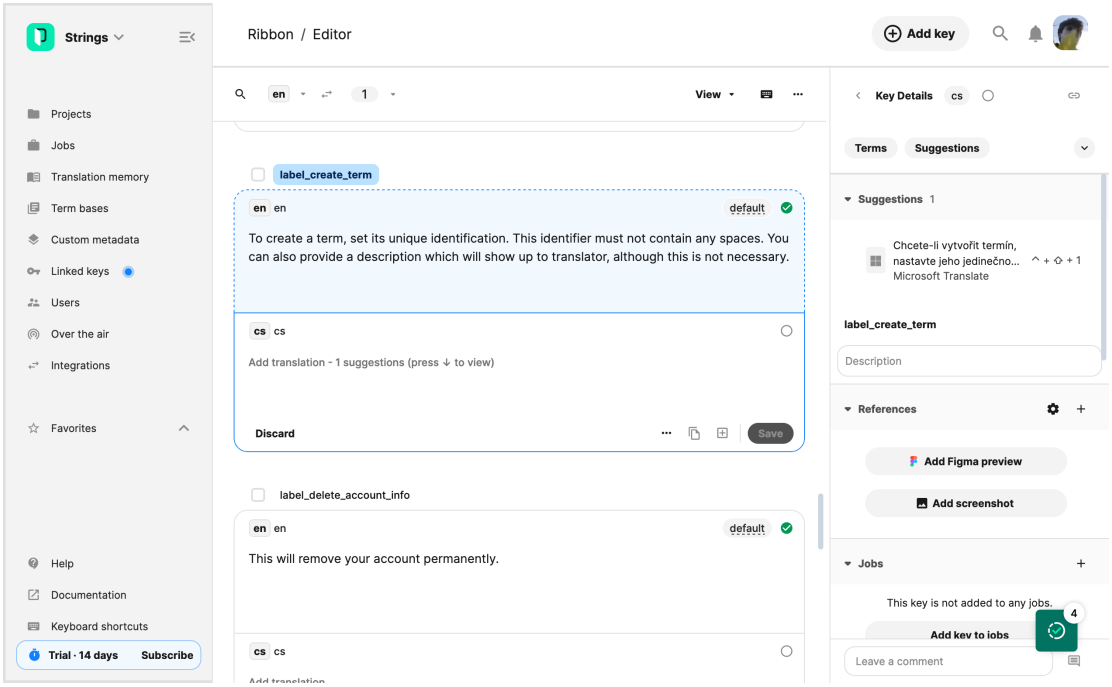
Projekt nelze nastavit jako veřejný a každý člen je tak pozván jeho vlastníkem či administrátorem. Mimo tyto dvě role může člen mít roli projektový manažer, vývojář, designér, překladatel či host (seřazeno podle množství oprávnění).

Integrací nabízí Phrase opravdu mnoho, nechybí mezi nimi propojení s GitHub a GitLab repozitářem, rozhraní pro příkazový řádek (CLI) nebo mobilní SDK pro An-

droid i iOS. SDK umožňuje stejně jako na platformě Crowdin vzdálenou aktualizaci textů.

Jako jediná zde porovávaná platforma nemá Phrase bezplatnou variantu, ale je možné si ji na 14 dní vyzkoušet bez nutnosti platby v režimu „Trial“. Po uplynutí zkušební doby si uživatel musí zvolit jednu z osmi placených nabídek. Nabídky jsou rozděleny na dvě kategorie; pro společnosti využívající vlastní překladatele a pro živnostníky či poskytovatele překladatelů. Nejlevnější plán obsahující službu Strings „Starter“ stojí 125 € měsíčně a limituje počet uložených slov na 200 000 [12]. Nutno dodat, že cena je uvedena za celou platformu nejen za službu „Strings“. Tu nelze zakoupit samostatně.

Nevýhodou je neexistence bezplatné varianty a vysoká cena nejlevnějšího dostupného tarifu. Další úskalí vidím v nepřehlednosti uživatelského rozhraní, které na první pohled působí chaoticky.



Obrázek 3.2. Překlad frází pomocí platformy Phrase Strings.

3.3 Localazy

Platforma je ve většině ohledech velice podobná již porovnaným, neliší se nutností registrace, účastí v projektech či tříděním frází do souborů.

Projekt v Localazy lze nastavit veřejným, nebo ho nechat soukromým a přidávat do něj členy ručně. Člen má hierarchicky uspořádané role „Translator“, „Trusted translator“, jehož překlady jsou automaticky označeny za ověřené, „Reviewer“, „Manager“ a „Owner“ [13]. Roli reviewer si lze představit obdobně jako roli vývojáře, poskytuje mu oprávnění vytvářet nové fráze a přidávat integrace.

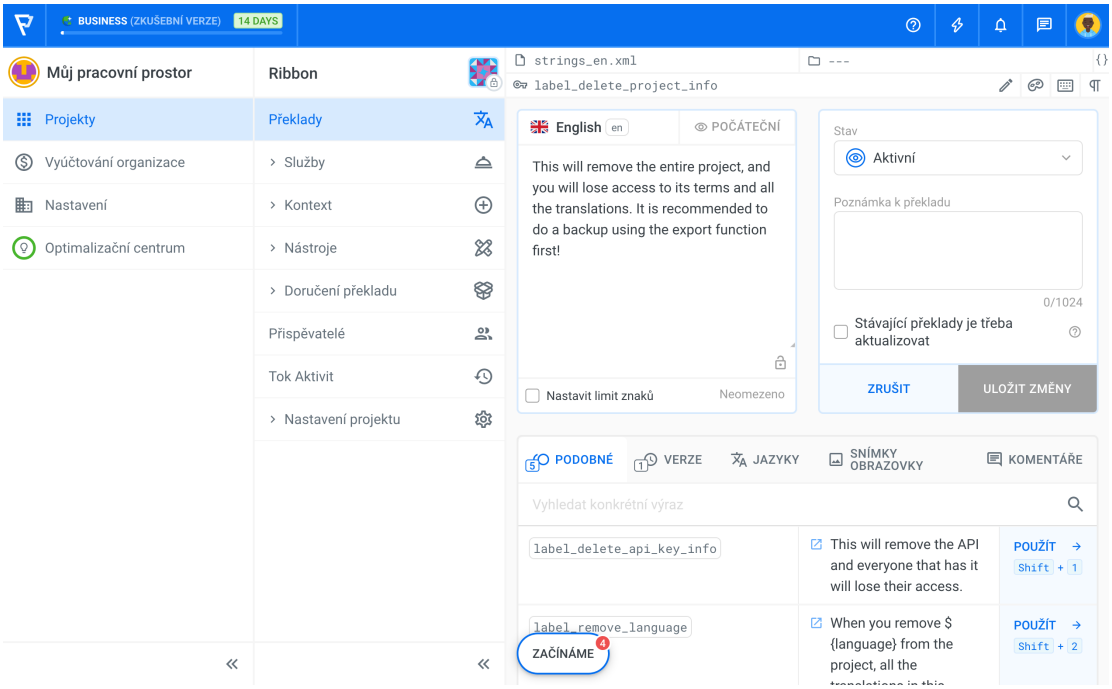
Naváži tedy rovnou možnostmi propojení. Nabízí stejně jako předchozí platformy mobilní SDK pro vzdálené doručení nových překladů, přístup přes API či CLI. Klasicky je možné i exportovat a importovat soubor s překlady v jednom z početného seznamu formátů.

Tarify, které tato platforma nabízí, jsou rozděleny do pěti kategorií, kde cenu lze dále upravit pomocí nastavení maximálního počtu frází. Jedna z nabídek je bezplatná, avšak

její využitelnost je mizivá. Projekt může obsahovat pouze 200 frází, na druhou stranu není omezeno použití SDK. Bez konkrétní ceny je uveden plán „Enterprise“, ostatní tarify jsou nastaveny následujícím způsobem[3] [14]:

- Professional – 31 €, maximálně 1 000 frází až 54 €/2 500 frází,
- Autopilot – 69 €/3 500 frází až 124 €/7 500 frází,
- Business – 155 €/10 000 frází až 389 €/100 000 frází.

Platformě se dle mého velice vyvedl design, který je přehledný a intuitivní. Její cena ale rychle roste s velikostí projektu, což může být pro některé překážka.



Obrázek 3.3. Překlad frází pomocí platformy Localazy.

3.4 PoEditor

PoEditor je ve spoustě ohledech odlišný od ostatních platforem, jeho uživatelské rozhraní je jednodušší, fráze nejsou děleny do souborů, ale všechny patří do jednoho seznamu a neobtěžuje uživatele s překlady pomocí umělé inteligence. Aplikace je rozdělena do dvou hlavních sekcí: překlady a fráze. Překladatelé budou pracovat v oddílu věnovaném překladům, kde si vyberou konkrétní jazyk. Pro vývojáře je určena druhá sekce. Tato sekce umožňuje přidávání a odebírání frází, úpravu jejich atributů, ale i změnu jejich překladů ve všech jazycích najednou.

Další rozdíl můžeme pozorovat v tom, jak v projektech fungují role. Překladatel je totiž přiřazován konkrétnímu jazyku, k jiným nemá přístup. Neexistuje zde role vývojář, pouze „Admin“. Ten má většinu oprávnění, jen nemůže smazat projekt a přidat nové administrátory. Poslední rolí je vlastník, ten je v projektu jen jeden a má všechny práva, včetně například nastavení viditelnosti projektu na veřejný.

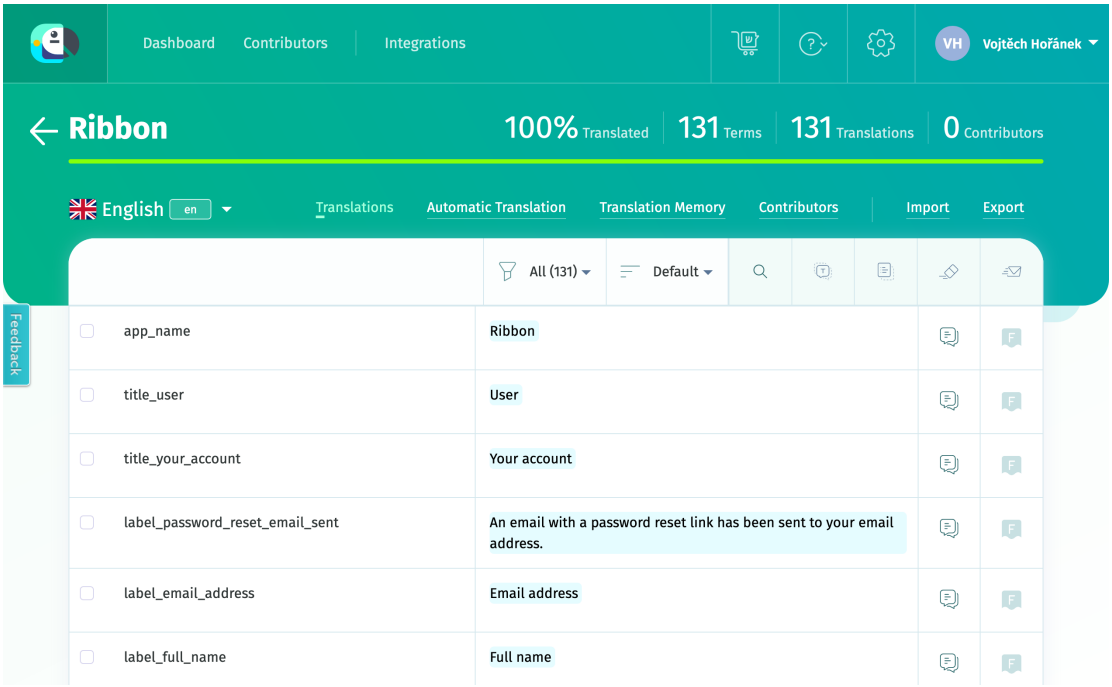
Integrace jsou v PoEditoru omezenější, nenabízí SDK pro mobilní aplikace ani CLI. Je ale možné využít API, a to v jakémkoliv placeném i neplaceném tarifu.

[3] Ceny jsem pro konzistenci převedl z dolarů na eura s kurzem 1 $ = 0,93 € aktuálnímu k 25. 4. 2024.

Co se tarifů týče, PoEditor nabízí využitelný bezplatný tarif, kde omezuje maximální počet frází na 1000 a počet členů projektu na pět. Další plány začínají na přívětivé ceně. Postupně odemykají funkce a navyšují již zmíněný maximální počet frází[4] [15]:

- Start – 12 €, 3 000 frází, již poskytuje možnost přidat neomezeně členů
- Plus – 36 €, 10 000 frází
- Premium – 95 €, 30 000 frází
- Enterprise – 160 €, 100 000 frází

Až na možnosti integrace je dle mého PoEditor jedna z nejlépe povedených služeb pro lokalizaci. Její design je velice intuitivní a jeho použití je tak snadné i pro netechnicky orientované uživatele.



Obrázek 3.4. Překlad frází pomocí platformy PoEditor.

3.5 Lokalise

Poslední porovnávaná služba Lokalise má mnoho společného s ostatními, fráze jsou tu opět roztříděny do souborů, lze nahrát existující texty nebo nastavit projekt jako veřejný. Jako v Phrase zde existuje koncept zadávání práce pro překladatele. Navíc se otevírá možnost pro vytvoření zadání s použitím velice pokročilé filtrů, které určí jaké fráze je třeba přeložit.

Platforma mě zaujala rozložením uživatelského rozhraní. V projektu totiž existuje pro správu frází jen jedna záložka „Editor“, která je určena nejen pro překladatele, ale i pro vývojáře. Lze v ní nastavit překlad do jednoho nebo více jazyků zároveň pomocí přepínače „Bilingual/Multilingual“, filtrovat nepřeložené fráze či neověřené překlady.

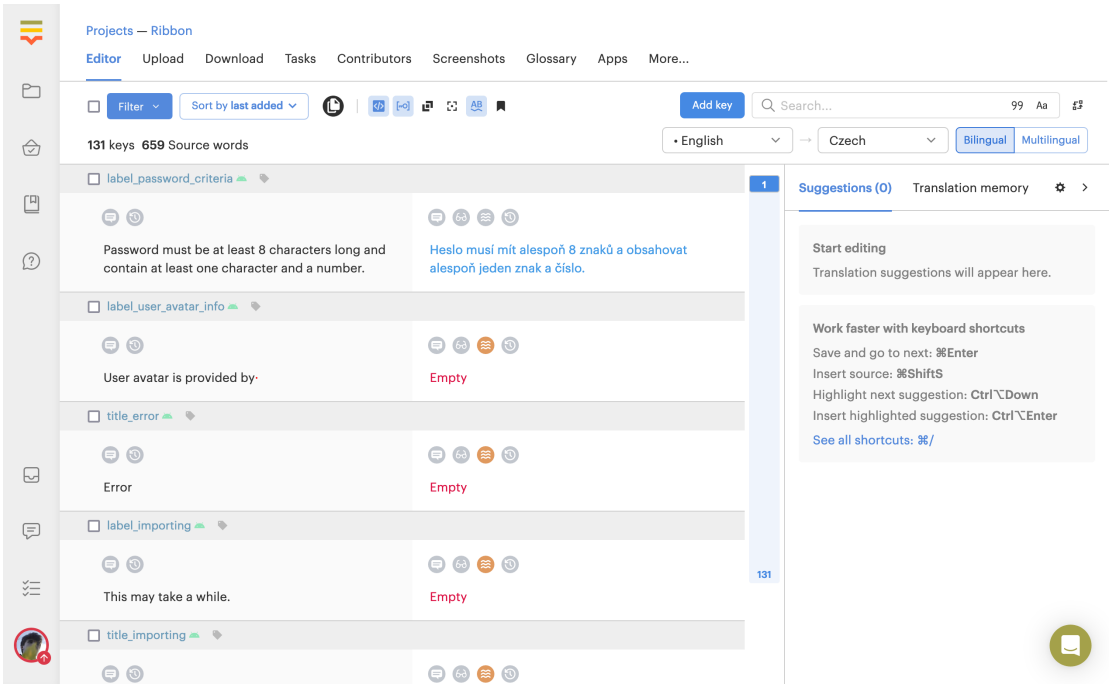
Při přidávání nového člena do projektu volí administrátor pouze ze dvou rolí: překladatel a administrátor. U prvního z nich se musí ještě nastavit k jakému jazyku bude mít přístup pro čtení a k jakému pro zápis. Pozvánky jsou odeslány na email i uživateli, který ještě nemá na platformě vytvořený účet.

[4] Ceny opět převedeny z dolarů na eura se stejným kurzem.

Lokalise nabízí nespočet možných nástrojů, vlastní API, CLI, ale i obdobné SDK pro mobilní aplikace [16].

Ačkoliv je možné použít bezplatný tarif, jeho využití je z pohledu budoucího použití spíše na zkoušku. Limituje počet frází na 500, v projektu lze mít jen dva členy a nelze při něm využít zmíněné SDK. Placené tarify začínají na ceně 112 € za 5 000 uložených frází, 215 € za 10 000 frází a 772 € za 30 000 frází. Nabízí i „Enterprise“ plán u kterého není uvedena cena [17].

Vyjma vysoké ceny mi přijde uživatelské rozhraní služby přehlcené a nepřehledné, libí se mi ale pokročilé možnosti filtrace klíčů a množství integrací, které jsou podporovány.



Obrázek 3.5. Překlad frází pomocí platformy Lokalise.

3.6 Srovnání

V tabulce 3.1 je uvedený souhrnný přehled porovaných řešení. Ve sloupci integrace není uveden kompletní výčet všech možností, které platforma nabízí, ale pouze ty, které jsou pro tuto práci relevantní.

Platform	Zdarma	Přívětivý design	Integrace	Přispěvatelé
Crowdin	Ano	Částečně	API, CLI, SDK	Zdarma pro veřejné projekty, jinak všichni v roli Manager
Phrase Strings	Ne	Částečně	API, SDK	150 uživatelů na projekt
Localazy	Ano, spíše na zkoušku	Ano	API, CLI, SDK	Neomezeno
PoEditor	Ano	Ano	API	Zdarma 5 přispěvatelů
Lokalise	Ano, spíše na zkoušku	Ne	API, CLI, SDK	Zdarma 2 uživatelé

Tabulka 3.1. Souhrnný přehled funkcionalit porovnaných řešení.

Kapitola 4
Analýza požadavků

Po rešerši aktuálně dostupných řešení jsem identifikoval jejich společné klíčové vlastnosti. Podle nich stanovil požadavky, které bude tvořená platforma splňovat. Vznikl model případů užití obsahující aktéry a jednotlivé případu užití. Na závěr jsem ověřil jejich pokrytí.

Analýza požadavků je vhodná zejména k pochopení potřeb budoucích uživatelů a upřesněním cílů, které jsem si v této práci stanovil. Ty budou implementovány v prototypu platformy.

4.1 Procesy

V této části se zaměřím na textový proces procesů, které jsou běžné pro uživatele služby. Popis slouží k prohloubení pochopení toho, co by implementované řešení mělo obsahovat a pro prioritizaci funkcionalit.

4.1.1 Registrace a přihlášení

Uživatel si před použitím služby bude muset zřídit uživatelský účet. Ten si vytvoří pomocí dvojice emailu a hesla. Pro snažší identifikaci ostatními uživateli vyplní i své celé jméno. Zadanou emailovou adresu bude nutné potvrdit pomocí kliknutí na odkaz zaslaný do emailové schránky. Pokud si takový účet nevytvoří, či nebude mít emailovou adresu potvrzenou, nebude vpuštěn dále do rozhraní.

Po potvrzení emailu bude uživateli dovoleno použít svůj účet k přihlášení a užívání celé platformy.

4.1.2 Správa projektů

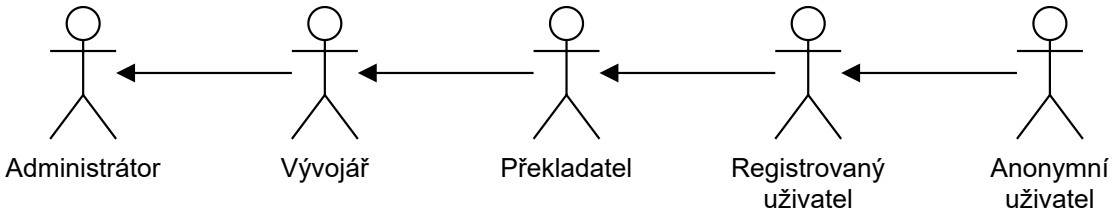
Uživatel je schopný vytvořit nový projekt, který ve službě de facto reprezentuje překládaný softwarový produkt. V něm se děje veškerá práce na překladech, vytváření nových frází, přidávání podporovaných jazyků či zvaní členů. Pokud je uživatel role administrátor, bude mít možnost projekt smazat, změnit jeho jméno či spravovat klíč pro jeho vzdálený přistup (více v sekci 4.2).

4.1.3 Překlad frází

V projektu má jeho člen možnost přeložit jakoukoli frází do zvoleného jazyka. Uživatel si bude moci nastavit výchozí jazyk pro překlad, podle kterého se má řídit. Klíčovou funkcí bude možnost filtrování nepřeložených frází, která usnadní práci překladatelům. Kromě filtrace bude možné zvolit řazení frází a vyhledávat pomocí jejich jména.

4.1.4 Vzdálený přístup

Přístup k exportu a importu frází bude umožněn přes API pod podmínkou autentizace pomocí klíče. Klíč je k disposici vývojářům a administrátorům z webového rozhraní služby a nejdříve je nutné ho vygenerovat. Po jeho získání je možné integrovat do softwarového projektu automatické stahování nových překladů s využitím HTTP požadavku na API.



Obrázek 4.1. Aktéři.

4.2 Aktéři

Uvnitř modelu existuje pět typů aktérů, které rozlišuji. Jsou to entity, které interagují se systémem. Kromě prvních dvou účastníků jsou ostatní role vázány na konkrétní projekt, na kterém se podílejí.

Anonymní uživatel Jde o uživatele, který ještě nemá vytvořený uživatelský účet. Je mu dovolenou pouze se přihlásit, či si účet vytvořit.

Registrovaný uživatel Jde o uživatele, který si vytvoří uživatelský účet v systému. Systém ukládá informace o jeho emailu a jménu i seznam projektů, kterých je členem.

Překladatel Překladatel je role vázaná na účastníka projektu, která mu umožňuje procházet fráze a tvořit, modifikovat a mazat jejich překlady. Zároveň si je může filtrovat a hledat v nich.

Vývojář Tato role obsahuje všechna práva jako překladatel, navíc může vytvářet fráze zcela nové, mazat a přejmenovávat je. Má možnost zobrazit si přístupový klíč k API, importovat a exportovat fráze a jejich překlady.

Administrátor Role administrátora dává uživateli největší kontrolu nad projektem. Mimo oprávnění vývojáře může v projektu spravovat podporované jazyky, zvát nové členy, měnit členům role. Může projekt přejmenovat či ho smazat. Dále také vytvořit či smazat API klíč.

4.3 Funkční požadavky

Funkční požadavky slouží k definici činností a chování, které bude systém nabízet. Jsou klíčové k zajištění očekávané funkcionality.

F1 – Autentizace Před použitím služby se musí uživatel přihlásit do existujícího účtu, nebo vytvořit účet nový. Přihlašování bude probíhat pomocí emailu a hesla. Ověření emailu bude vyžadováno a provedeno pomocí zaslání odkazu do emailové schránky.

F2 – Správa uživatelského účtu Po registraci bude moci uživatel změnit své zobrazované jméno, obnovit si heslo nebo celý účet smazat.

F3 – Tvorba a správa projektu Uživatel si bude moci vytvořit libovolný počet projektů, kterým dá vlastní název. Projekt bude moci později přejmenovat, či smazat.

F4 – Nastavení jazyků v projektu Do projektu si lze přidat jakýkoliv podporovaný jazyk a jeho regionální variantu, posléze ho lze i odstranit.

F5 – Překlad frází Uživatelům je k dispozici funkce pro překládání frází v projektu. Mohou si vybrat referenční jazyk a filtrovat fráze, které čekají na překlad.

F6 – Správa frází Vývojáři a administrátoři budou moci vytvářet nové fráze pro překlad, měnit jejich název a popis. Uvidí, kdo fráze vytvořil, v jakých jazycích a kým byly přeloženy. Případně je mohou i mazat. V seznamu frází budou moci filtrovat přeložené a nepřeložené.

F7 – Správa členů Administrátor bude moci přizvat nové členy do svého projektu. Na výběr má jednu ze tří rolí: administrátor, vývojář nebo překladatel. Tato pozvánka je identifikována emailem pozvaného uživatele a ten ji po přihlášení může přijmout. V případě potřeby má administrátor možnost členovi změnit jeho roli i mu členství odebrat.

F8 – Import a export Vývojář a administrátor můžou do projektu importovat existující fráze ve formátu Android XML a Apple Strings. Ve stejných formátech budou moci fráze ve zvoleném jazyku exportovat.

F9 – Přístup k API Administrátor může k projektu vytvořit API klíč, který umožňuje přístup k exportu a importu bez nutnosti použití webového rozhraní. Vývojáři mají k tomuto klíči také přistup, nesní ho ale vytvořit či smazat.

4.4 Nefunkční požadavky

Nefunkční požadavky hrají klíčovou roli v definování vlastností a chování softwarového systému. Jsou doplňkem funkčních požadavků, které specifikují, co má systém dělat, a určují, jakým způsobem má systém fungovat.

N1 – Dostupnost webového rozhraní Webové rozhraní platformy bude volně dostupné a kompatibilní s všemi běžnými prohlížeči. Díky responzivnímu designu se rozhraní automaticky přizpůsobí jak stolním počítačům, tak i mobilním zařízením, čímž zajistí pohodlné používání na jakékoliv obrazovce.

N2 – Lokalizace Základní jazyk platformy bude angličtina. Použité technologie zaručí snadné rozšíření o překlad do dalších jazyků

N3 – Vzhled Aplikace bude mít jednoduchý a přehledný vzhled, který bude následovat principy Material Design 3 [18].

4.5 Případy užití

Následující případy užití jsou všechny prováděny ve webovém rozhraní platformy, pokud není zmíněno jinak. Rozdělil jsem je podle funkcionalit, ke kterým se vážou.

4.5.1 Případy užití vztahující se k uživateli

UC1 – Registrace Uživatel si chce vytvořit nový účet. Vyplní formulář určený k registraci. Tímto se stane registrovaným uživatelem.

UC2 – Přihlášení Uživatel již má svůj účet. Pro přihlášení využije přihlašovacího formuláře a tím se autentizuje do aplikace.

UC3 – Změna zobrazovaného jména Uživatel chce změnit jméno, které zadal při registraci. Přejde na stránku svého účtu, upraví vstup se jménem a zvolí volbu uložit.

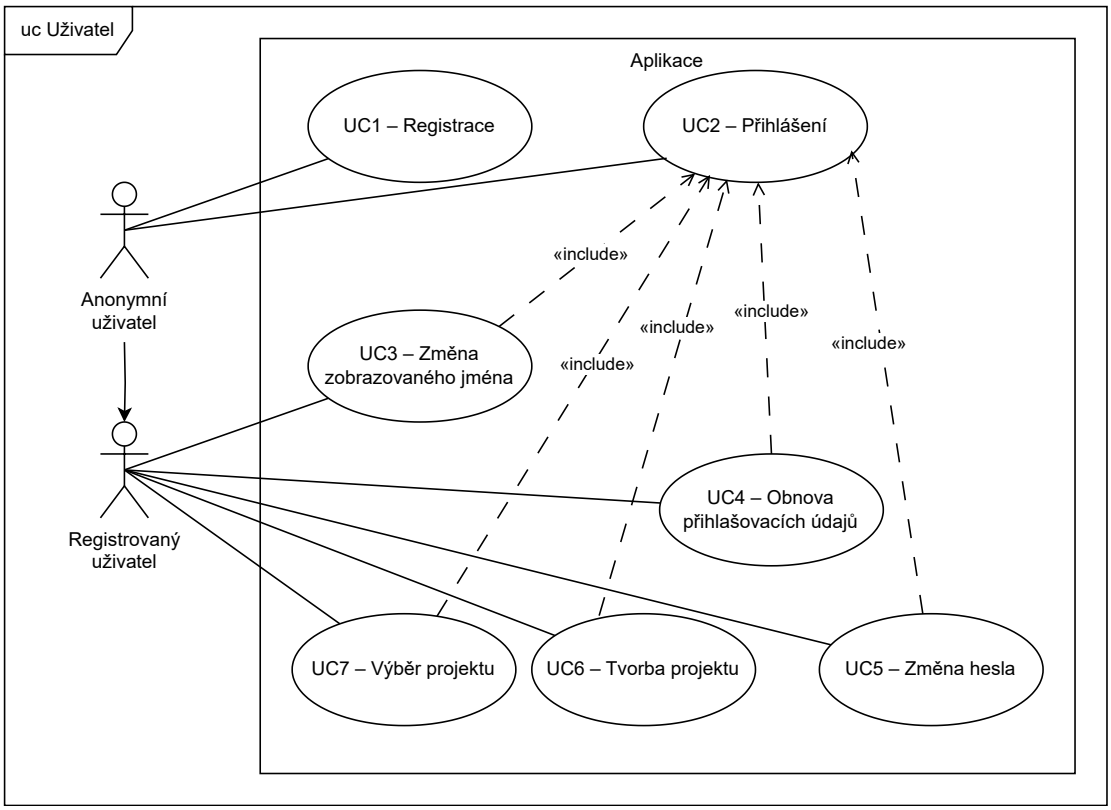
UC4 – Obnova přihlašovacích údajů Uživatel zapomněl heslo a chtěl by si jej obnovit. Na přihlašovací obrazovce zvolí možnost zapomenutého hesla, zadá email pod

kterým se registroval. Po potvrzení mu přijde email s instrukcemi a odkazem pro obnovení hesla.

UC5 – Změna hesla Uživatel chce změnit své heslo a je momentálně přihlášen. Přejde do sekce svého účtu a zvolí možnost obnovit heslo. Obdobně jako v UC4 je mu odeslán email s instrukcemi a odkazem pro nastavení.

UC6 – Tvorba projektu Uživatel chce vytvořit nový projekt. Přejde na domovskou stránku, stiskne tlačítko „vytvořit projekt“, zvolí jeho název a akci potvrdí. Automaticky se tak stane i administrátorem projektu.

UC7 – Výběr projektu Uživatel si může vybrat projekt jehož je členem a dále s ním pracovat. Podle jeho role jsou mu zobrazeny akce, které na projektu může provádět.



Obrázek 4.2. Diagram případů užití relevantní k uživateli.

4.5.2 Případy užití vztahující se k správě projektu

UC8 – Přidání jazyka Vyvíjený software, k němuž náleží projekt v naší aplikaci, je třeba rozšířit do nové země. Administrátor do projektu přidá nový jazyk, aby tento fakt reflektoval.

UC9 – Odebrání jazyka Software již nechce podporovat jeden z jazyků. Administrátor ho odebere z projektu a tím smaže i překlady frází v tomto jazyce.

UC10 – Vytvoření API klíče V projektu je třeba zprovoznit export textů přes API. Administrátor kvůli tomu vygeneruje přístupový klíč, kterým se lze do API autorizovat.

UC11 – Smazání API klíče Exporty API už nejsou nutné, nebo není vhodné, aby bývalý vývojář měl stále přístup k této funkci. Administrátor tedy zvolí smazání klíče pomocí webového rozhraní.

UC12 – Přečtení API klíče Vývojář či administrátor integruje do svého softwaru automatické stahování překladů přes API a potřebuje k autentizaci přístupový klíč. Využije možnosti zkopírovat existující klíč ve webovém rozhraní projektu.

UC13 – Přizvání uživatele Do projektu je třeba přidat nového člena. Administrátor tak učiní pomocí emailové adresy zvaného uživatele.

UC14 – Změna role člena Překladatel nyní potřebuje přidávat nové fráze, na což nemá práva. Administrátor změní jeho roli v projektu na vývojáře.

UC15 – Odstranění člena Jeden z členů přestal na projektu pracovat a je nežádoucí, aby k němu měl nadále přístup. Administrátor ho tedy z projektu odstraní.

UC16 – Změna názvu projektu Projekt potřebuje změnit název, administrátor tak může učinit v jeho nastavení.

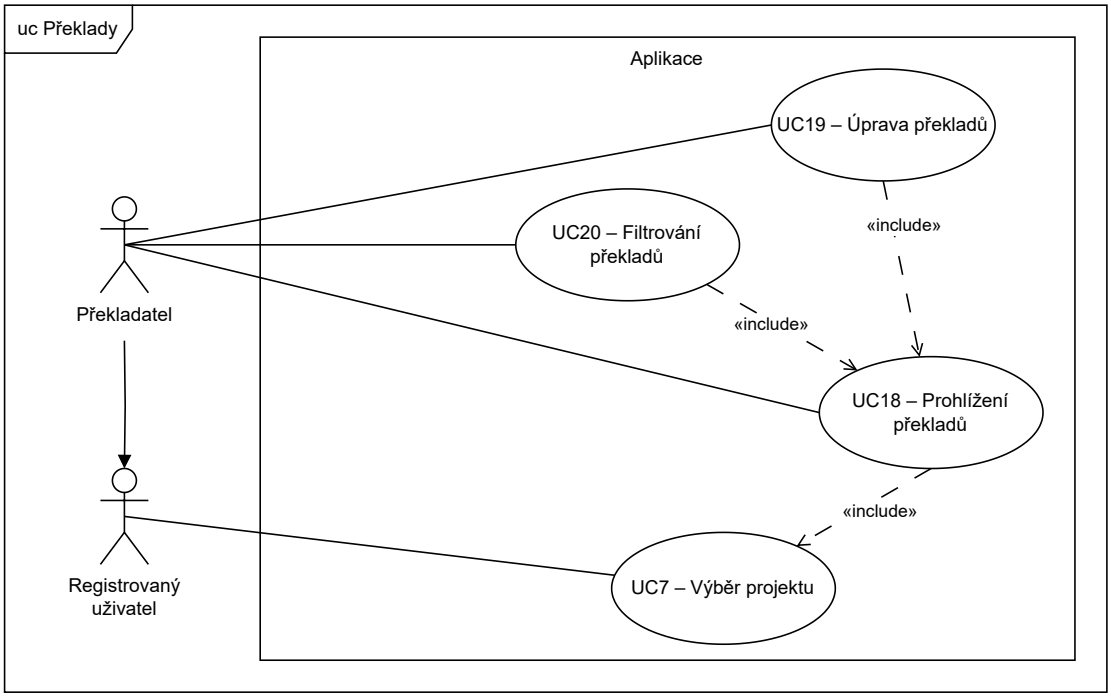
UC17 – Odstranění projektu Jelikož vývoj na projektu byl ukončen, je žádoucí ho smazat ze sytému. Administrátor tak může učinit v nastavení.

4.5.3 Případy užití vztahující se k správě překladů

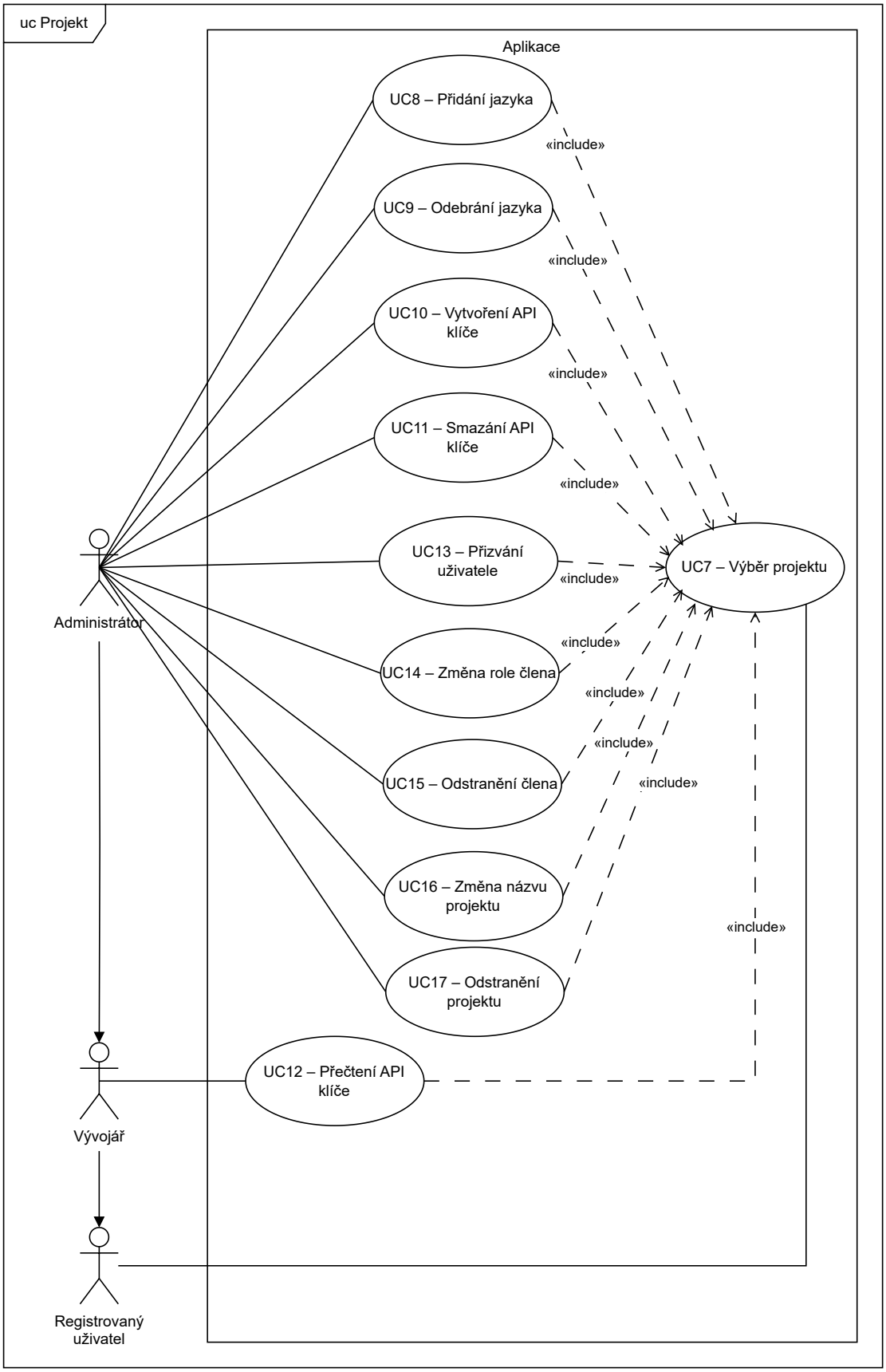
UC18 – Prohlížení překladů Člen projektu prochází fráze a jejich překlady ve vybraném jazyce. Uvidí u také poznámku od jejího autora, pokud existuje.

UC19 – Úprava překladů Člen projektu může při prochází frází tvořit, upravovat či mazat překlady v jazyce, který má momentálně vybrán.

UC20 – Filtrování překladů Člen projektu si při procházení potřebuje zobrazit pouze překlady splňující nastavené filtry, například pouze již existující. Bude moci i vyhledávat pomocí názvů fráze.



Obrázek 4.4. Diagram případů užití vztahující se k správě překladů.



Obrázek 4.3. Diagram případů užití vztahující se k správě projektu.

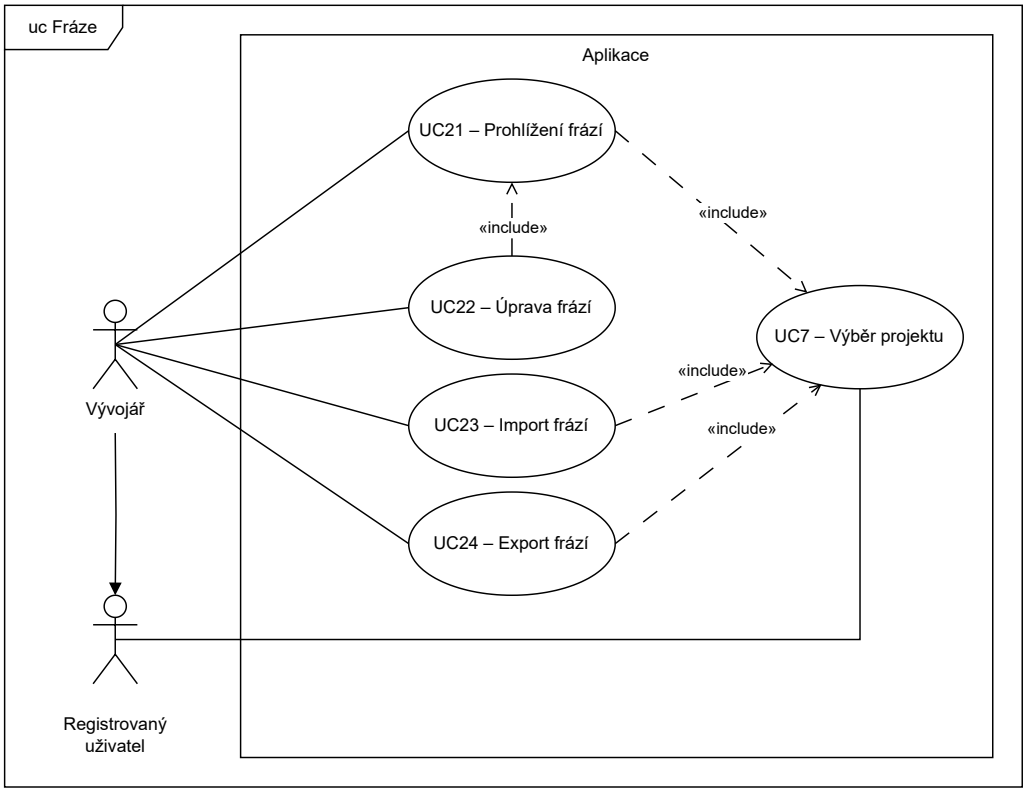
4.5.4 Případy užití vztahující se k správě frází

UC21 – Prohlížení frází Vývojář či administrátor projektu prochází fráze a u každé se mu zobrazují statistiky o přeloženosti, poznámka a informace o autorovi. Překlady si může po kliknutí i zobrazit.

UC22 – Úprava frází Vývojář či administrátor projektu může při procházení frází vytvářet fráze nové, upravovat název či popis existujících, či je mazat.

UC23 – Import frází Vývojář či administrátor projektu potřebuje nahrát soubor s existujícími frázemi a jejich překlady v daném formátu do projektu. Může zvolit přepsání existujících překladů, pokud by nastala kolize.

UC24 – Export frází Vývojář či administrátor projektu potřebuje získat překlady pro použití v nové verzi vyvíjeného softwaru. Pomocí webového rozhraní si zvolí požadovaný formát a jazyk. Aplikace mu následně poskytne soubor s exportem.

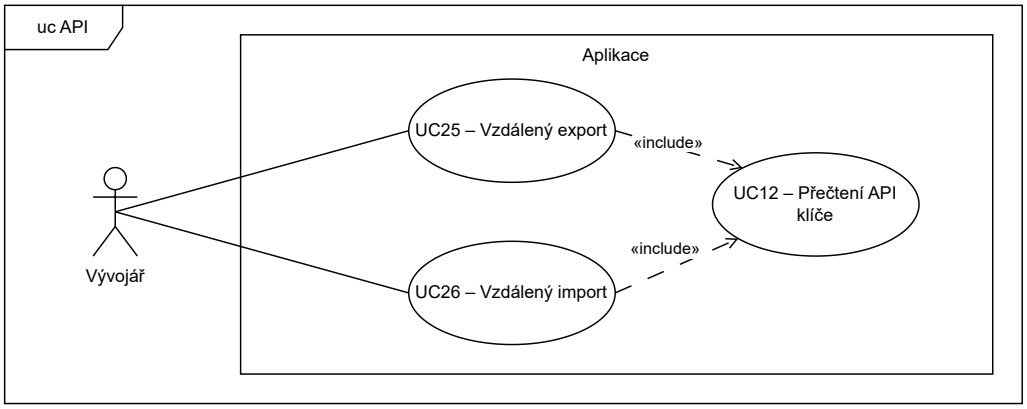


Obrázek 4.5. Diagram případů užití vztahující se k správě frází.

4.5.5 Případy užití vztahující se k vzdálenému přístupu

UC25 – Vzdálený export Vývojář či administrátor s přístupovým klíčem zavolá pomocí API endpoint určený pro export. Server mu poskytne uložené texty ve formátu který uživatel požaduje. Obdoba UC24, která se ale neprovádí ve webovém rozhraní.

UC26 – Vzdálený import Pomocí API endpointu určeného na import vývojář či administrátor s přístupovým klíčem nahraje existující fráze a jejich překlady. Obdoba UC23, která se ale neprovádí ve webovém rozhraní.



Obrázek 4.6. Diagram případů užití vztahující se k vzdálenému přístupu.

4.6 Pokrytí požadavků případy užití

V tabulce 4.1 je znázorněno, že pro každý funkční požadavek existuje případ užití. Všechny případy užití navíc pokrývají alespoň jeden funkční požadavek.

	F1	F2	F3	F4	F5	F6	F7	F8	F9
UC1	✓								
UC2	✓								
UC3		✓							
UC4		✓							
UC5		✓							
UC6			✓						
UC7			✓						
UC8				✓					
UC9				✓					
UC10			✓						✓
UC11			✓						✓
UC12			✓						✓
UC13							✓		
UC14							✓		
UC15							✓		
UC16			✓						
UC17			✓						
UC18					✓				
UC19					✓				
UC20					✓	✓			
UC21						✓			
UC22						✓			
UC23						✓		✓	
UC24						✓		✓	
UC25						✓		✓	✓
UC26						✓		✓	✓

Tabulka 4.1. Pokrytí požadavků případy užití.

Kapitola 5
Analýza technologií

V poslední analytické kapitole se budu věnovat rešerši technologií, které využiji při implementaci. Hlavním cílem je nalezení nástrojů, umožňující pohodlný vývoj s výhledem pro budoucí rozšíření. Je také důležité zjistit, jak jsou mezi sebou technologie kompatibilní a vyhnout se tak zbytečné práci navíc.

5.1 Backend

Jelikož celá platforma bude architektury klient-server, zaměřím se teď na výběr technologie, která bude schopná obstarat serverovou část platformy. Alternativa k použití vlastního serveru by byla vyvíjet pouze klienta, přímo napojeného na externí služby, které by sloužili například jako úložiště dat. Nevýhodou tohoto řešení je ale poměrně vysoká míra uzamčení s vybranou externí službou a případná budoucí migrace by vyžadovala přepsání podstatné části klienta. Navíc, eventuální vývoj klientů pro jiné platformy by znamenal implementaci celé byznys logiku nanovo.

Výběr správné technologie pro vývoj serverové aplikace tak umožní abstrahovat klienta od specifických rozhodnutí pro využité externí služby.

5.1.1 Java

Java je objektově orientovaný jazyk z dlouhou historií, který je pro vývojáře serverových aplikací velice známý. Software v něm napsaný běží na virtuálním stroji JVM, který umožňuje přenositelnost mezi operačními systémy a architekturami procesorů. Při kompilaci zdrojového kódu není vytvořen kód strojový (spustitelný procesorem), nýbrž speciální mezivrstva zvaná *Bajtkód*. JVM se pak stará o její interpretaci na konkrétním systému, kde aplikace běží.

5.1.2 Kotlin

I přes svou relativní mladost se Kotlin, jazyk s rozsáhlou podporou pro vývoj různých typů aplikací, již těší velké oblibě. Byl navržen s cílem dosáhnout plné interoperability s Javou. Syntaxe i filosofie obou jazyků se sice v mnohém liší, to ale vývojářům nezabraňuje využít Kotlin v existujícím Java projektu. Díky tomu, že Kotlin umožňuje běh na virtuálním stroji JVM, mohou mezi sebou jazyky oboustranně komunikovat. Lze tak například využívat existující frameworky a knihovny, které byly původně vytvořené pouze pro Javu.

Mimo běh na JVM má ale Kotlin i jiné způsoby kompilace. Díky technologii *Kotlin Multiplatform* je možné jeden a ten samý kód zkompilovat do strojového kódu spustileného bez JVM, do WebAssembly[1] nebo do JavaScriptu. Lze tak sdílet kód mezi podporovanými platformami. Je navíc dosažitelné použití nativních funkcí dané platformy pomocí `expect` a `actual` modifikátorů. Dopodrobna se této problematice budu věnovat v kapitole 7.

[1] binární formát spustitelný na webových stránkách

5.1.3 Spring Framework

Spring Framework je platforma určená primárně pro tvorbu serverových enterprise aplikací naprogramovaných v Javě. Jádro Springu je založeno na principu *Inversion of Control*, primárně pomocí vkládání závislostí. Spravuje životní cyklus objektů, jejich tvorbu a konfiguraci. Vývojář nevytváří instance tříd ručně, pouze řekne Springu jak je vytvořit a kde je následně bude potřebovat. To zjednoduší údržbu a testování.

Nad Springem je postaveno mnoho zásuvných modulů. Jeden z nich „Spring Boot“ usnadňuje tvorbu a konfiguraci projektu poskytnutím většiny nastavení. Není tak potřeba se zabývat prvotním rozběhnutím a rovnou se lze pustit do psaní byznys logiky aplikace. Mezi další často používané moduly patří například „Spring Security“ pro autentizaci a autorizaci uživatelů [19].

Díky interoperabilitě Kotlinu s Javou lze i v něm Spring využít. Z mého pohledu to ale není ideální přístup. Konfigurace se provádí způsobem, který je vhodný v Javě, a tak v Kotlinu působí jaksi těžkopádně. Například při zmíněném vkládání závislostí se často nelze obejít bez použití `lateinit` modifikátoru pro proměnné třídy, který ale může způsobit neočekávané chyby v případě její neinicializace. Stejně tak se lze při použití frameworků určených přímo pro Kotlin vyhnout anotacím.

5.1.4 Ktor

Kotlinu nativní framework Ktor umožňuje vyvíjet serverové (ale i klientské) aplikace s využitím všech funkcí, které Kotlin nabízí. Je z velké části postaven na `coroutines` (česky koprogramy), které jsou integrální částí Kotlinu. Umožňují asynchronní volání funkcí s možností pozastavení běhu. Oproti klasickému přístupu s využitím vláken nemusí operační systém přepínat kontext. V důsledku tak běží rychleji s menší režií.

Oproti Springu není Ktor komplexní platforma poskytující například vkládání závislostí. Je tedy na vývojáři zvolit pro tyto potřeby vhodné nástroje. Sám o sobě ale umožňuje integraci externích pluginů, přičemž některé z nich jsou vyvíjeny pod jeho záštitou. Jde například o pluginy pro autentizace, serializaci dat nebo směrování. Ukázka nejjednoduššího serveru, který odpovídá na požadavek kořenového adresáře textem `Hello, world!`, je uvedena v následujícím kódu převzatém z dokumentace frameworku Ktor [20].

```
fun main() {
   embeddedServer(Netty, port = 8000) {
      routing {
         get ("/") {
            call.respondText("Hello, world!")
         }
      }
   }.start(wait = true)
}
```

Výpis kódu 5.1. Ukázka použití Ktor serveru.

5.1.5 Koin

Jak jsem již zmínil, Ktor neposkytuje mechanismus pro vkládání závislostí. Je tak nasnadě použít knihovnu, která tuto funkcionalitu implementuje. Jednou z takových je

Koin. Ten byl vyvinut přímo pro Kotlin a obdobně tak využívá jeho bohatou syntaxi. Oproti Springu není nutné používat `lateinit` modifikátory. Vkládání je totiž prováděno přímo do konstruktorů objektů[2]. Další výhodou je existence modulu pro Ktor a jeho integrace je tak bezproblémová a snadná [21].

Definice objektů, které potřebují vkládat se dělí do *modulů*. Ten může vypadat zhruba následovně:

```
val applicationModule = module {
   singleOf(::RemoteRepository) bind Repository::class
   factoryOf(::UseCaseImpl) bind UseCase::class
}
```

Výpis kódu 5.2. Ukázka modulu v Koinu.

Tento modul definuje, jakým způsobem konstruovat čtyři objekty a zároveň jim určuje jejich životní cyklus. První řádek modulu, začínající `singleOf` říká, že pokud bude někdo potřebovat instanci `RemoteRepository` či `Repository` (v tomto případě je první zmíněné třída, druhé rozhraní), Koin vytvoří novou, nebo poskytne již existující tak, že po celý běh aplikace bude existovat pouze jedna instance. V druhém případě `factoryOf` je při každé žádosti vytvořena instance nová.

Pokud by pak například třída `UseCaseImpl` potřebovala instanci `Repository`, stačí přidat její závislost jako parametr konstruktoru:

```
class UseCaseImpl(
   private val repository: Repository,
)
```

Výpis kódu 5.3. Konstruktorové vkládání závislostí.

Jak je vidět, není potřeba žádných anotací a objekty navíc neví, kdo a jakým způsobem jim závislosti poskytne.

5.1.6 Firebase

Firebase je platforma poskytovaná společností Google obsahující služby pro usnadnění vývoje softwaru. Pod svojí záštitou nabízí například databázové úložiště, hosting statických webových stránek, analytika, systém pro automatické ohlašování chyb nebo autentifikaci. Služby jsou mezi sebou provázány v rámci jednoho projektu [22].

Pro přístup k jednotlivým službám je poskytováno SDK pro Android, iOS, JavaScript, Javu a další.

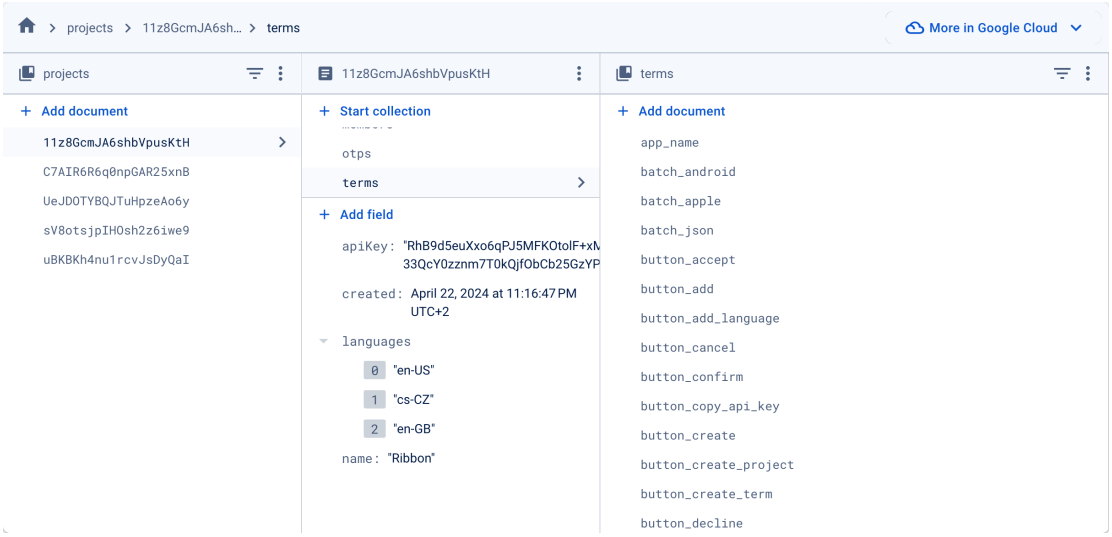
5.1.7 Firebase Firestore

Firestore slouží jako databázové úložiště na principu dokumentové NoSQL databáze. Data se tak neukládají do tabulek, nýbrž do kolekcí dokumentů. Ty v sobě obsahují seznam typu klíč-hodnota. Narozdíl od tradičních relačních databází nemá dokument pevně stanovené schéma a napříč kolekcí se tak jeho datový typ[3] může naprosto lišit.

Velkou výhodou této služby je její dostupnost a snadnost použití. Jelikož běží „v cloudu“, nemusí se vývojář starat o její konfiguraci, zálohování dat či škálování [23].

[2] Jsou poskytnuty i další možnosti vkládání, využívající například `lazy` delegát.

[3] Ve skutečnosti žádný datový typ dokument nemá, obrat využívám pouze k přiblížení struktury dat.

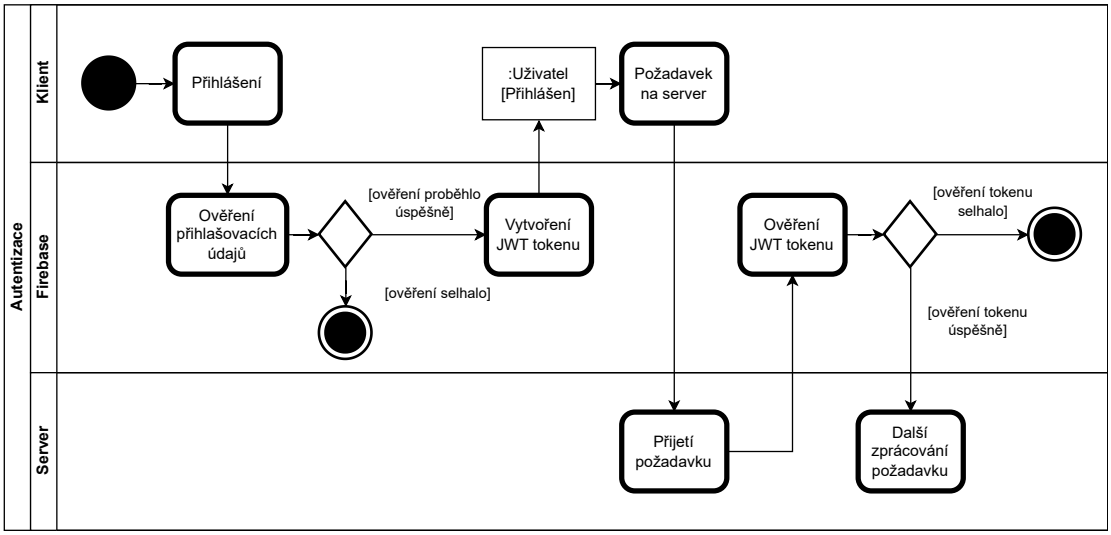


Obrázek 5.1. Zobrazení databázového úložiště Firestore ve webovém rozhraní.

5.1.8 Firebase Authentication

Další službou patřící pod Firebase je Authentication. Jak již název napovídá, jde o kompletní řešení autentizace uživatelů. Její nasazení je opravdu snadné, vývojář nemusí řešit způsob zabezpečení údajů, hesel a přístupových klíčů. To vše si totiž spravuje Firebase sám a aplikacím poskytuje SDK pro integraci. Navíc je možné pro přihlášení a registraci využít externích poskytovatelů jako je například Google či Facebook [24].

Autentizace funguje na základě JWT tokenů. Ten klient (frontend) při přihlášení vygeneruje a při každém požadavku na prostředek vyžadující přihlášení pošle na server kde je ověřen. Generování a ověřování je obsluhováno zmíněným SDK, pro vývojáře je tedy velice jednoduché toto řešení implementovat, viz obrázek 5.2.



Obrázek 5.2. Diagram přihlášení autentizace pomocí Firebase Authentication.

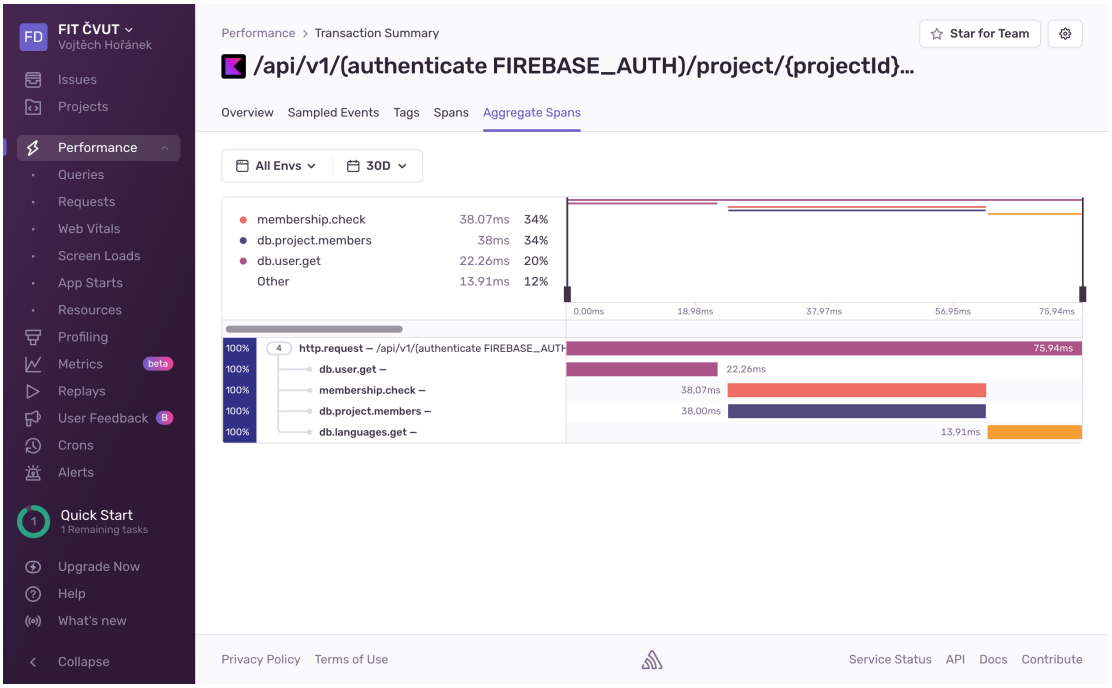
5.1.9 Sentry

Pro podporu běhu serveru a jeho monitoring bohužel nelze využít službu Firebase Crashlytics. Ta je totiž určená pouze pro mobilní aplikace. Jako alternativa se jeví služba

Sentry, která nabízí monitoring výkonu, chyb a mnoho dalšího. Všechny funkce jsou navíc dostupné z SDK, které je poskytované v několika variantách určených buď pro konkrétní serverové frameworky nebo jazyky [25].

Sentry by mělo pomoci při hledání případných pádů, které se můžou na serveru vyskytnout. Takovou událost totiž zaznamená se všemi dostupnými detaily a momentální konfigurací aplikace a odešle její záznam do databáze chyb. Ta je vývojářům dostupná přes webové rozhraní a můžou tak okamžitě reagovat a chybu opravit.

Nedílnou součástí provozu serverové aplikace je i monitoring výkonu. Ten je v Sentry také dostupný. V rámci každého požadavku je vytvořena transakce, která trvá až do konce jeho zpracování. Vývojář si mezitím může oddělit její části na takzvané „spans“ a pomocí nich sledovat chování jednotlivých funkcí. Jak je vidět na obrázku[4] 5.3, transakce `http.request` obsahuje tři *spany*. Jako první se volá `db.user.get`, tedy získání uživatele z databáze, dále se kontroluje jeho členství v projektu pomocí `membership.check` a `db.project.members` a na závěr se teprve získává seznam jazyků v projektu. Vývojáři je tedy hned patrné, že 88 % času celého požadavku zabírá ověřování práv uživatele, a jen v nepatrném zbytek se server věnuje akci, která je pro volajícího důležitá.



Obrázek 5.3. Měření výkony požadavky na server.

5.2 Frontend

Frontend platformy bude uživatelům poskytován jako interaktivní webová aplikace a tato sekce se tak zabývá vhodným výběrem frameworku, který je určen pro jeho tvorby. Měl by splňovat především jednoduchost implementace uživatelského rozhraní s vidinou budoucího vývoje a rozšíření o další platformy, například mobilní aplikace.

[4] Obrázek byl pořízen po implementaci a jedná se přímo o požadavek na jazyky v projektu

5.2.1 React

React je deklarativní knihovna pro vykreslování webových uživatelských rozhraní od společnosti Meta. Její použití je možné při tvorbě SPA, mobilních nebo serverově renderovaných aplikací napsaných v JavaScriptu a TypeScriptu. Neposkytuje ovšem vývojářům kompletní framework pro tvorbu takové aplikace, stará se jen o vykreslování rozhraní a jeho komponent. Je tak často využíván s dalšími knihovnami, které se mají za úkol routování a jiné žádané funkcionality.

Vykreslování rozhraní probíhá pomocí virtuální abstrakcí nad DOM. Při nutnosti aktualizace se tak využije virutálního DOMu a části stránky, ve kterých se nic nezměnilo React nepřekresluje. To oproti překreslování celého DOMu razantně zrychluje výkon. Vývojář se o hlídání změn nemusí start a z jeho pohledu to vypadá, jako kdyby se stránka vykreslila celá [26].

5.2.2 React Router

Jak jsem již zmínil, React neobsahuje funkcionalitu routování, tedy způsob navigace uživatele aplikací. Na to lze použít například knihovnu React Router. Tradiční webové aplikace (tedy nikoliv SPA) používají pro směrování uživatele mechanismus prohlížeče. Při načtení stránky stáhne požádá server o dokument, stáhne a vyhodnotí JavaScript a CSS a nakonec celou stránku vykreslí. Tento proces se děje při navštívení každé stránky, případně je využita mezipaměť prohlížeče pro rychlejší načtení.

Oproti tomu SPA aplikace využívají směrovaní přímo na klientovi, stažení dokumentu proběhne pouze při prvním načtení a dále se o vykreslování obsahu stránky stará kód na klientovi. Díky klientskému směrování je tak možné stránky načítat rychleji a uživatel ihned vidí potřebný obsah [27].

5.2.3 Kotlin JS

Další alternativou, spadající pod zmíněný Kotlin Multiplatform, je využití Kotlinu i pro vývoj webového frontendu. V dnešní době existuje opravdu mnoho knihoven pro multiplatformní vývoj a většina z nich je podporována i při kompilaci Kotlinu do JavaScriptu [28].

Využití Kotlinu by umožnilo sdílet části kódu mezi frontendem a backendem- Výrazně by tak usnadnilo vývoj. Navíc, pokud by se v budoucnu platforma rozšířila o mobilní aplikaci, její vývoj by přímo využil již napsaný kód.

5.2.4 Compose Multiplatform

Compose je deklarativní framework pro tvorbu uživatelského rozhraní v Kotlinu. Původně byl dostupný pouze na androidu. Později však byly vytvořeny verze pro iOS, Windows, macOS, Linux a web. Poslední zmíněná má navíc ještě dvě varianty [29]:

- Compose Web – Funguje na stejném principu jako Compose pro ostatní platformy. To znamená že existující uživatelské rozhraní lze bez nutnosti změny rovnou spustit ve webovém prohlížeči. Využívá Kotlin/WASM (kompliace Kotlinu do WebAssembly) a rozhraní vykresluje pomocí grafické knihovny SKIA[5] do plátna (*canvas*) bez využití DOMu.
- Compose HTML – Zcela odlišná knihovna, fungující pouze ve webovém prohlížeči. Využívá kompilaci Kotlinu do JavaScriptu a při vykreslování komponent upravuje DOM stránky. Jde tak o nativní přístup na webu. který se podobá zmíněnému Reactu.

[5] `https://skia.org/`

5.2.5 Kobweb

Kobweb je rozšířením nad knihovnou Compose HTML, které z ní dělá kompletní framework pro vývoj webové aplikace. Stará se o routing stránek, statický export a přidává spoustu užitečných komponent, které by si vývojář jinak musel vytvořit sám. Compose HTML se od ostatních variant Compose výrazně odlišuje v dostupném API a je nutno rozumět, jak kód stránky generuje. Kobweb se snaží toto chování sjednotit pomocí mezivrstvy *Silk*, která přidává layout funkce jako například `Row` a `Column` a zavádí z Compose známy systém *Modifierů*.

Mimo jiné Kobweb umožňuje *full-stack* vývoj a poskutuje tak i nástroje pro tvorbu serverového rozhraní. Není tak nutné použít jiný framework, například Spring.

5.2.6 Firebase Analytics

Ve webovém rozhraní neočekávám časté pády. Ani měření výkonu by vzhledem k povaze webových aplikací nedávalo veliký smysl. Možným kandidátem pro podporu běhu klientské části platformy je služba Firebase Analytics. Ta umožňuje shromažďovat analytické data o chování uživatelů. Navíc ji lze využít i pro sledování neobvyklého chování a detekcí případných chyb.

Sběr informací probíha v událostech zvaných *event*. Ty mají název a můžou obsahovat parametry, například ID navštívené položky.

Použití Firebase Analytics ve webové aplikaci navíc přináší výhodu v podobě automatického sledování navštívených stránek. Naslouchá akcí uživatele a sbírá údaje například o posunu stránky, stahování soubor a vyplnění formuláře. S každou událostí jsou připojeny informace o URL adrese, jazyku a velikosti prohlížeče [30].

Jak jsem již zmínil, službu je možné využít i pro vyhodnocení běhu aplikace. Při detekování chování, které by nemělo nastat, lze vývojáře informovat odesláním nové události. Mezi detekované chyby by mohlo například patřit neúspěšně odeslání požadavku na server a selhání při nedovoleném přístupu ke chráněným zdrojům.

Vzhledem k úrovni sledovaných informací na backendu platformy a nízké předpokládané míře chybovosti webového rozhraní jsem se rozhodl tuto službu z prototypu vynechat. Testování frontendu bude navíc probíhat nejen při jeho vývoji, ale hlavně závěrem metodou uživatelského testování, viz 8.3. To zaručí uspokojivou jistotu pro ostrý provoz. Budoucí vývoj by nicméně měl počítat s možností využití této služby.

5.3 Shrnutí

Po analýze zmíněných technologií jsem se rozhodl jít cestou, která umožní sdílení kódu mezi frontendem a backendem a zároveň poskytne flexibilitu pro budoucí vývoj.

Server bude napsán v Kotlin s použitím Ktor frameworku pro obsluhu HTTP požadavků a pro vkládání závislostí bude využit Koin. Jako datové úložiště poslouží Firebase Firestore, které zaručí žádanou flexibilitu. Pomocí Firebase Authentication bude spravována autorizace do platformy. Sentry pak poskytne solidní monitorování běhu aplikace a zaručí snadnou opravu případných chyb.

Jelikož použití Kotlin Multiplatform umožní mezi backendem a frontendem platformy sdílet kód, nabízí se jako velice vhodné řešení pro vývoj klientské webové aplikace. Kobweb spolu s Compose HTML tak bude zajišťovat vykreslování uživatelského rozhraní, mezitím co ostatní vrstvy aplikace budou používat čistě multiplatformní knihovny. To zahrnuje Ktor ve funkci klienta pro odesílání požadavků na sever a Koin, stejně jako na backendu, pro vkládaní závislostí. Díky tomu bude moci případný vývoj

klienta na novou platformu použít existující kód a jen přidat novou vrstvu s nativním uživatelského rozhraní.

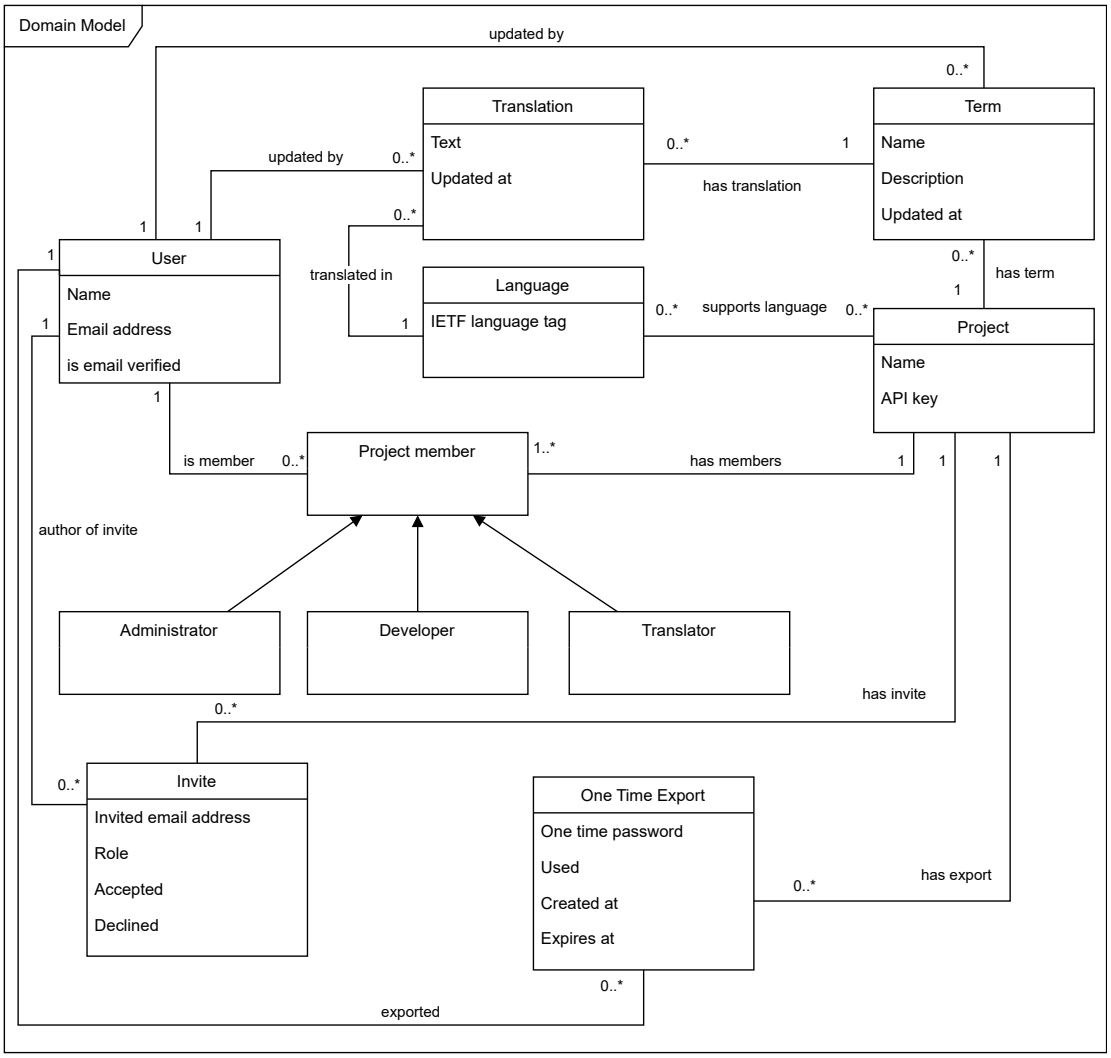
Zdůrazním že knihovnu Compose Web používající Kotlin/WASM jsem nezvolil proto, že při vykreslování obsahu stránky do plátna nepůsobí chování aplikace nativně a momentálně není knihovnou podporován prohlížeč Safari. Oproti tomu uživatelský zážitek při použití Compose HTML nebude rozdílný od jakékoliv jiné webové stránky.

Více se architektuře platformy budu věnovat v sekci 6.3.

Kapitola 6
Návrh

V následující kapitole se budu zabývat návrhem architektury, doménových a databázových modelů a uživatelského rozhraní aplikace. Pro vyhotovení byly využity funkční požadavky vytyčené v kapitole 4. Aby bylo možné si jednotlivé objekty rovnou propojit s kódem, jsou jejich názvy uvedeny v angličtině.

6.1 Doménový model



Obrázek 6.1. Diagram doménového modelu.

User Účet uživatele platformy. Je jednoznačně identifikován svou emailovou adresou, kterou musí před přihlášením do aplikace potvrdit. Dále je uloženo uživatelovo celé jméno, které se zobrazuje ostatním uživatelům.

Project Reprezentuje základní organizační jednotku, ve které uživatelé aplikace pracují. Při jeho vytvořením je zvoleno jméno a později může administrátor vygenerovat i API klíč, určený ke vzdálenému přístupu.

Project member Představuje člena projektu, kde na jedné straně je uživatel a na druhé projekt, jehož je členem. Projekt může mít neomezeně členů a uživatel může být členem, jakéhokoliv počtu projektů. Každé členství má určenou roli, která uživateli poskytuje různá oprávnění. Služba rozpoznává tři role, které jsou hierarchicky seřazené:

- Translator – Role s nejméně oprávněními, umožňuje uživateli překládat fráze. Je vhodná pro externí účastníky projektu, kteří mají za úkol pouze pracovat na překladech frází.
- Developer – Role vhodná pro vývojáře, kteří na projektu pracují. Dává jim stejná práva jako překladatelům, navíc ale můžou spravovat fráze, číst API klíč (pokud existuje), importovat do projektu existující fráze a exportovat existující.
- Administrator – Tato role poskytuje uživateli veškerá práva k projektu, např. jeho přejmenování, smazání či správu členů a podporovaných jazyků.

Language Jedná se o třídu zastupující jazyk podporovaný platformou. Pro jeho jednoznačné určení je použit IETF BCP 47 identifikátor, který je standardem používaným pro vyjádření lidských jazyků na internetu [31, 32]. Pro představu, vyjádření češtiny v BCP 47 by vypadalo takto: `cs-CZ`, zatímco britská varianta angličtiny má vyjádření `en-GB`. Projekt pak může podporovat libovolné množství těchto jazyků.

Invite Pozvánka určená pro uživatele registrovaným pod konkrétní emailovou adresou. Pokud účet s touto adresou nebyl vytvořen, pozvánka je přesto platná a čeká na vytvoření takového účtu. Při vytváření pozvánky je zadána i role, kterou bude uživatel v projektu zastávat. Dále jsou uloženy informace o přijmutí či odmítnutí pozvánky.

One Time Export Jedná se o přístup k exportu provedeného z uživatelského rozhraní. Každý takový export má náhodně vygenerované přístupové heslo a je platný po dobu určenou atributem „Expires at“. Po jeho použití, tedy stažení souboru s exportem, je rovněž zneplatněn.

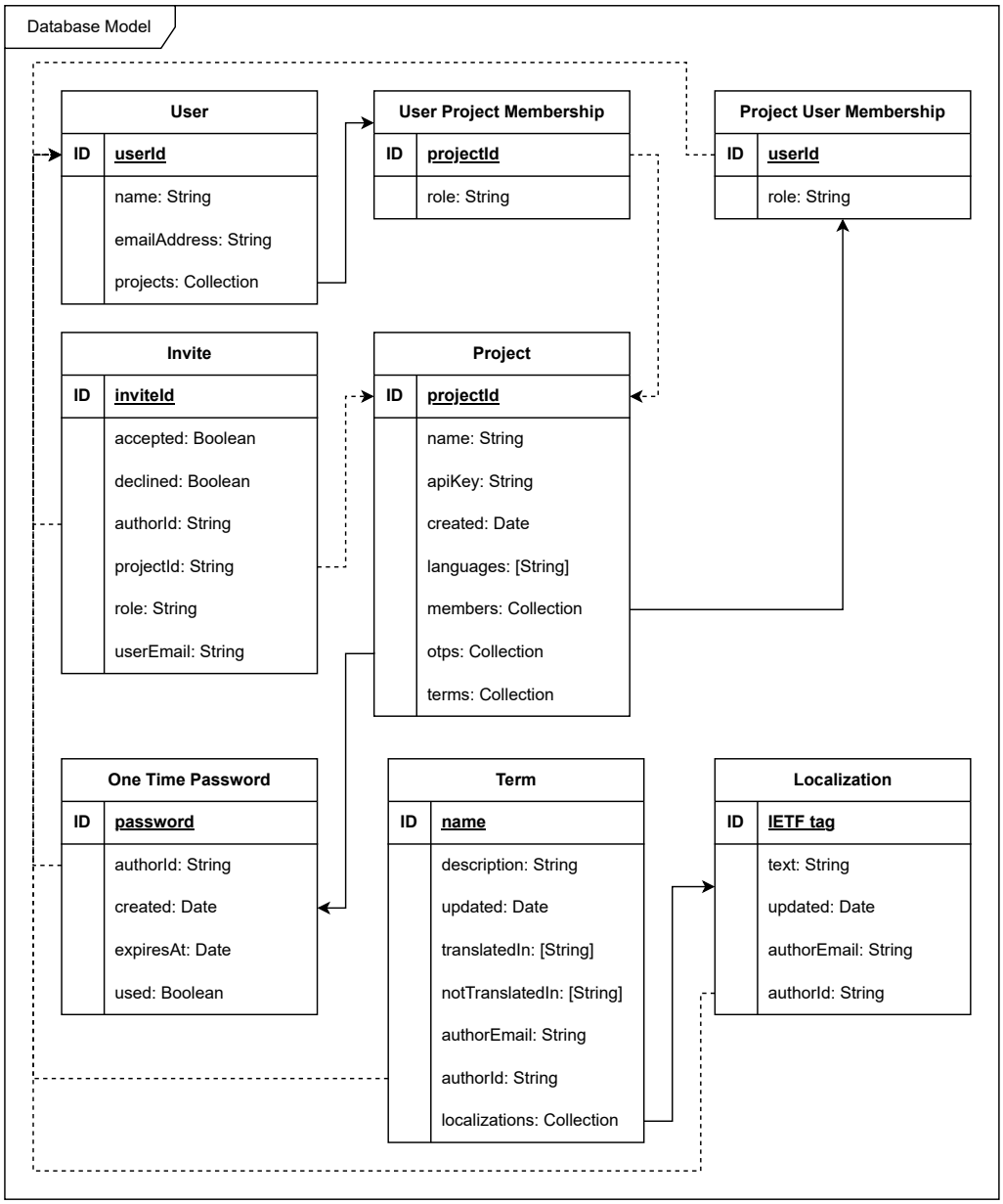
Term Každy projekt má k sobě přiřazený seznam frází, které jsou určeny k přeložení. Fráze je tvořena názvem (většinou systémový identifikátor), možným popisem a informací o času její aktualizace. Rovněž jsou ukládány informace o uživateli, který frázi vytvořil, nebo změnil.

Translation Jednotlivé překlady frází jsou pomocí této třídy přiřazeny k jazyku a uživateli, který je vytvořil, nebo změnil. Udržuje se i informaci o momentu poslední aktualizace.

6.2 Databázový model

Ačkoliv NoSQL databáze nemají standardizovaný způsob modelování a dokumenty nemají pevně stanovené schéma, pro potřeby vizualizace uložení dat a korektního návrhu

vniklo schéma vyobrazené na obrázku 6.2. Obdélník označuje schéma kolekce dokumentů, které z návrhu dodržuje, i když to databáze nevyžaduje. Nepřerušované čáry značí podkolekce se šipkou ve směru od rodiče k potomkovi. Přerušované pak označují referenci na identifikátor jiného dokumentu. Ve smyslu relačního modelu by se jednalo o cizí klíč.



Obrázek 6.2. Diagram databázového modelu.

Za zmínku stojí duplikace (nebo také denormalizace) dat. Lze si všimnout, že uživatel vlastní kolekci členství a zároveň v projektu nalezneme kolekci členů. Obě tyto kolekce pouze určují role uživatelů v projektu, neukládají celý projekt nebo uživatele. Denormalizace dat je v NoSQL databázích běžná praxe, za účelem zvýšení výkonu [33]. Ačkoliv je pro konzistenci dat vyžadováno upravovat dvě kolekce, má denormalizace

přínos ve značném zrychlení čtení. Zamezí se *joinům* a provádění složitých dotazů. Režie správy dat je navíc v tomto případě vzhledem k četnosti dotazování se na členy projektu z pohledu uživatele, ale i projektu zanedbatelná. Další duplikace dat proběhla v dokumentu *Term*, kde se ukládají informace o jazycích, ve kterých je fráze přeložena. To slouží k urychlení vypočítání statistik. Čtení dokumentů z podkolekce *Localizations* by operaci značně zpomalilo.

6.3 Architektura

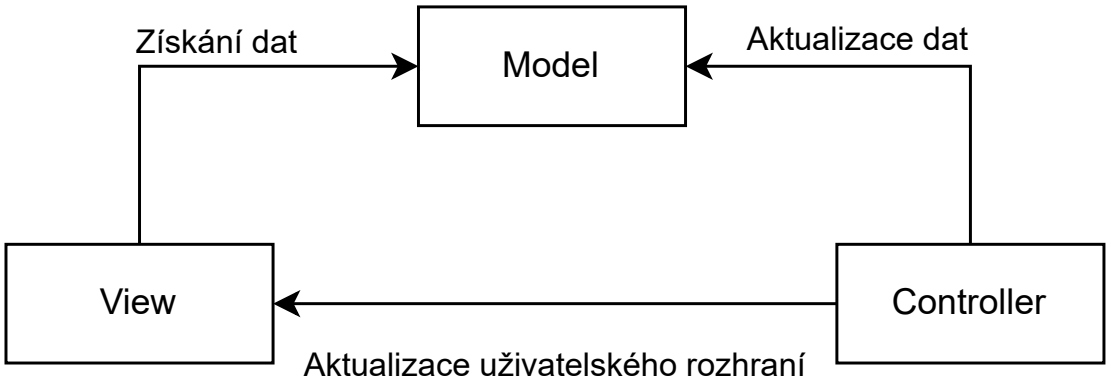
Sekce se bude zabývat návrhem vnitřní architektury platformy, propojení frontendu s backendem a sdílením kódu mezi nimi. Popíše vrstvy zvolené architektury a uvede motivaci pro její výběr.

Ani jedna ze zvolených technologií nevynucuje použití konkrétní architektury a je tak na mně zvolit vyhovující. Při volbě jsem se zaměřil hlavně na výběr takového architektonického vzoru, který jasně definuje vrstvám jejich zodpovědnost (prezentační/datová vrstva) a umožní co možná největší testovatelnost.

6.3.1 MVC

Jde o architektonický vzor, jehož historie sahá až do roku 1970. Používá se pro vývoj širokého spektra aplikací. Základní myšlenka spočívá v rozdělení aplikační logiky do tří vrstev a tím odděluje jejich zodpovědností [34, 35, 36].

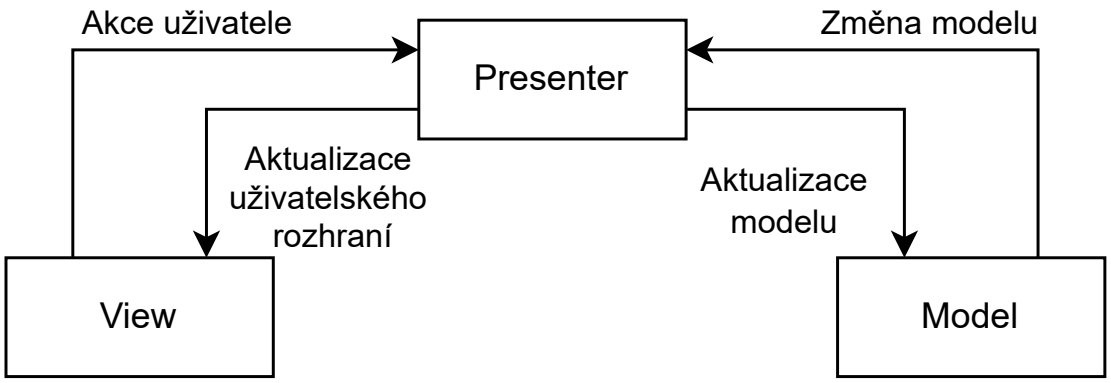
- Model *(model)* – Jde o vrstvu zajišťující byznys logiku aplikace. Definuje datové třídy a operace nad nimi.
- View *(pohled)* – Stará se o prezentování dat a zpracování uživatelských interakcí. Jde například o tlačítka nebo vstupy.
- Controller *(řadič)* – Propojuje vrstvy View a Model. Zaručuje správné zobrazení aktuálních dat, naslouchá uživatelským vstupům a stará se o následnou změnu modelu.



Obrázek 6.3. Model-View-Controller.

6.3.2 MVP

Návrhový vzor Model-View-Presenter vychází z MVC a poprvé se objevil v devadesátých letech minulého století. Snaží se vylepšit neduhy MVC větším oddělením vrstev a zlepšení testovatelnosti. Rozdílem, jak je vidět na obrázku 6.4, je nezávislost *View* na *Modelu*. Jako mezivrstva slouží *Presenter*, který se stará o veškerou prezentační logiku a a komunikuje s *View* přes drženou referenci. Další změnou je sloučení *Controlleru* z MVC do *View*, to se nyní stará i o uživatelské interakce [34, 35, 36].



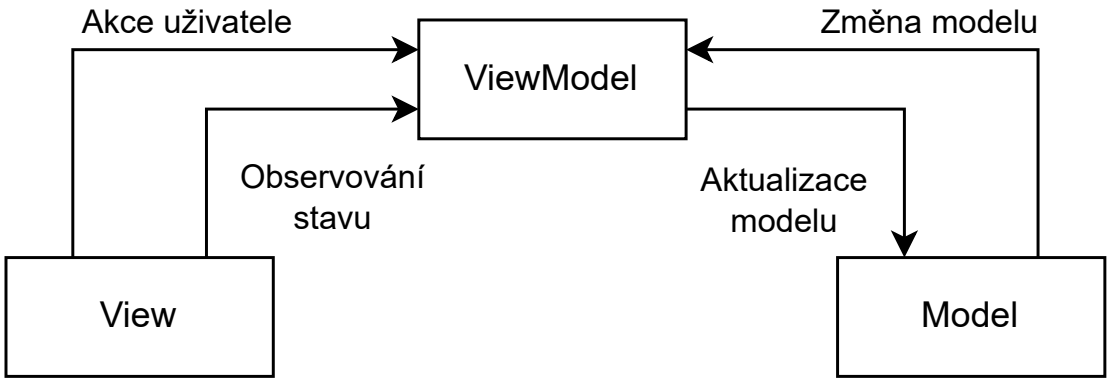
Obrázek 6.4. Model-View-Presenter.

6.3.3 MVVM

Model-View-ViewModel posouvá míru oddělení jednotlivých vrstev na ještě vyšší úroveň. Základní myšlenkou je oddělení byznys logiky od prezentace dat. *View* observuje (česky sleduje) změny stavu, který *ViewModelu* vystavuje. Tímto způsobem je zaručena aktualizace zobrazovaných dat bez nutnosti dalších žádostí o aktualizaci *View*, jak tomu bylo u MVP.

Uživatelské interakce zpracovává *View* s následným zavoláním funkcí poskytovaných *ViewModelem*. Ten se stará o aktualizaci modelu a svého stavu.

ViewModel si navíc nedrží referenci na *View* (díky vystavování stavu to není potřeba) a je naprosto nezávislý na způsobu zobrazení dat. Je ho tak možné znovu použít pro různé typy rozhraní či obrazovky uvnitř aplikace [34, 35, 36].



Obrázek 6.5. Model-View-ViewModel.

6.3.4 Clean Architecture

Česky volně přeloženo jako „čistá architektura“ je návrhový vzor, zaměřující se na rozdělení softwaru do vrstev, které jsou od sebe silně izolované. Od zmíněných architektur se výrazně odlišuje zaměřením se na byznys logiku aplikace. Definuje tak vrstvu *Modelu* ze výše zmíněných architektur, jejíž organizace byla dosud v roli vývojáře. Autor Robert C. Martin (na internetu známý jako Uncle Bob) ji zhotovil na základě analýzy jiných architektur, které si kladli společné cíle [37]:

- Nezávislost na frameworku – Architektura není závislá na existenci konkrétního frameworku. Ten je použit jako nástroj a nelimituje návrh softwaru.

- Testovatelnost – Byznys logika softwaru je testovatelná bez externích závislostí jako například UI, databáze, webový server atd.
- Nezávislost na UI – Uživatelské rozhraní lze snadno vyměnit bez zásahu do zbytku systému (byznys logiky).
- Nezávislost na databázi – Výměna databázového úložiště za jinou technologii neovlivní byznys vrstvu systému.
- Nezávislost na externím systému – Byznys logika je naprosto izolovaná od okolního světa.

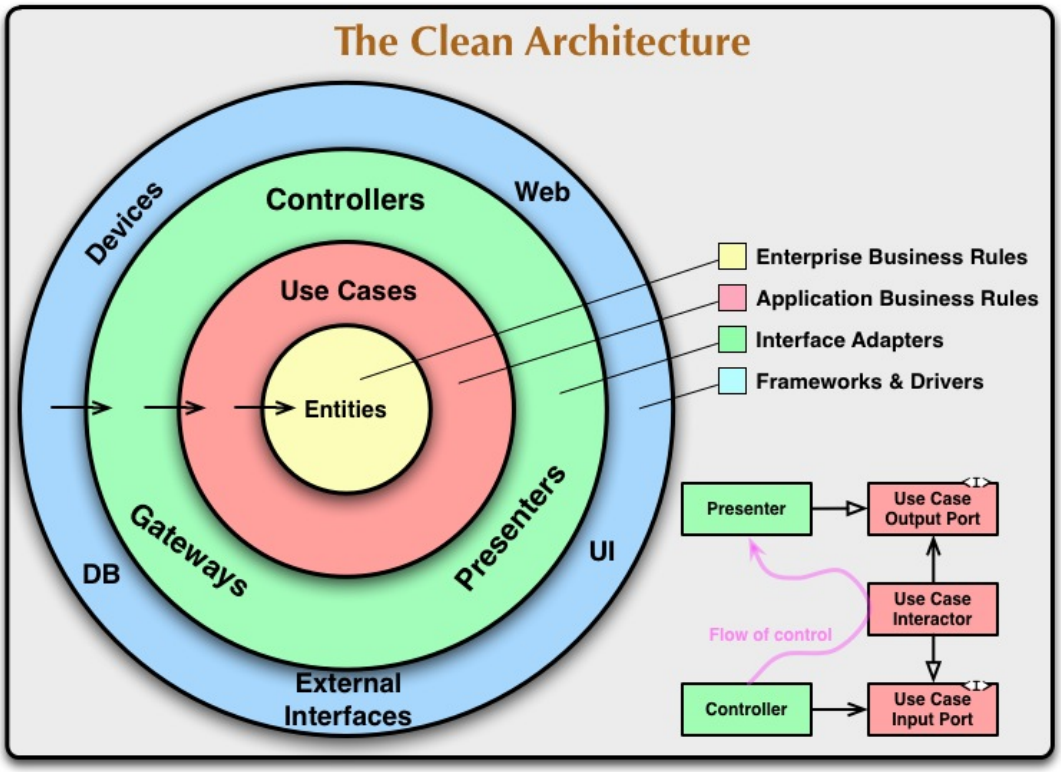
Tyto pravidla jsou pak shrnuta na obrázku 6.6. Kolečka reprezentují vrstvy aplikace, ty nejvzdálenější od středu slouží jako nástroje pro vývoj (tedy například framworky). V samotném středu je pak byznys logika systému. Důležité pravidlo architektury, naznačené šipkou jdoucí k entitám, je takzvané pravidlo závislostí. To říká, že závislosti jednotlivých vrstev jsou jednosměrné a to v dostředném směru. Nelze se tedy například z vrstvy *Entity* nebo *UseCases* odkazovat na kód ve vrstvě s *Controllery*.

Entity reprezentují byznys objekty celého podniku (případně pouze jediné aplikace) a jsou nejobecnějším pohledem na fungování celého systému. Nepředpokládá se jejich častá změna. *Entity* si lze představit jako datové objekty se základními operacemi.

Případy užití (neboli *UseCases*) obsahují byznys logiku aplikace. Definují chování, pravidla a orchestrují datový tok mezi entitami.

Další vrstvy (jejichž počet není podle Martina omezený) se starají o propojení byznys logiky s externími službami (databáze, framework atd.).

Architektura zaručuje vysokou testovatelnost softwaru a možnost velice snadné náhrady externího systému.

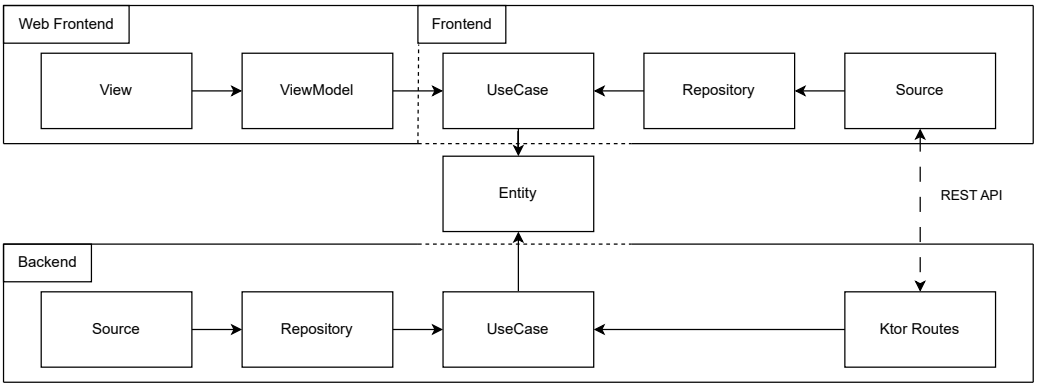


Obrázek 6.6. Clean Architecture, převzato z [37].

6.3.5 Shrnutí

Díky přednostem Clean architektury je vhodným adeptem pro využití při implementaci. Zvláště se pak, díky Kotlin Multiplatform, nabízí sdílení *Entit* mezi frontendem a backendem. Na obrázku 6.7 je vidět konkrétní vrstvení aplikace. Obě části implementace mezi sebou sdílí entity, další vrstvy jsou pak oddělené. Za zmínku stojí rozdělení frontendu na web a obecnou část. To je příprava pro případný budoucí vývoj klientské části pro jinou platformu (například Android aplikace, terminálové rozhraní atd.), kdy stačí naprogramovat vrstvy *View* a *ViewModel*. Ty jsou na zvoleném rozhraní závislé. Zbytek frontendu je od těchto vrstev oddělený.

Komunikace mezi oběma částmi platformy probíha pomocí REST API, jehož návrhu se budu věnovat v další části.



Obrázek 6.7. Architektura platformy.

6.4 REST API

REST API je způsob bezestavové komunikace mezi klientem a serverem. Poskytuje vývojářům velkou flexibilitu, škálovatelnost a pro svojí jednoduchost je často volen před protokolem SOAP. Komunikace probíhá pomocí HTTP požadavků, provádějící standardní operace jako čtení, vytváření, aktualizaci nebo mazání. Tyto operace jsou známé také pod označením CRUD a pro jejich rozlišení se v požadavku na prostředek používají HTTP metody GET, POST, DELETE a jiné.

Formát, v jakém jsou data poskytována, není REST protokolem určen a dokonce se lze setkat s API, které podporuje více formátů. Často je využíváno serializace dat do formátu XML nebo JSON.

Návrh takového API by měl splňovat následujících šest zásad [38]:

- Jednotné rozhraní – Prostředky vyžadované klientem jsou poskytovány jednotným a předvídatelným způsobem.
- Oddělení klienta a serveru – Obě strany jsou na sobě nezávislé. Klient pouze zná URI požadovaného prostředku, server jen poskytuje data pomocí HTTP protokolu a nijak nemodifikuje klienta.
- Bezestavovost – Všechny požadavky na server musí obsahovat veškeré informace potřebné k jeho zpracování. Serveru není povoleno si o požadavky ukládat jakékoliv informace, není za potřebí vytvářet *sessions*.
- Možnost ukládání do mezipaměti – Pokud je to možné, je žádoucí na obou stranách komunikace používat mezipaměť. Server musí informovat klienta, zda je tento mechanismus povolen. Cílem je zrychlení klienta a zlepšení škálovatelnosti serveru.

- Vícevrstvý systém – Nepředpokládá se přímá komunikace mezi klientem a serverem. Požadavek může projít mezilehlými systémy a ani jedna strana by to neměla poznat.
- Kód na vyžádání (nepovinně) – Většinou jsou přes REST poskytované statické data, je ale možné na vyžádaní posílat i spustitelný kód.

6.4.1 Koncové body

Endpointy, česky koncové body, jsou kombinací jednoznačného URI identifikátoru a HTTP metody. Určují funkcionalitu poskytovanou serverem klientovi.

Na základě doménového modelu jsem proto navrhnul koncové body, které bude backend platformy poskytovat. Pro umožnění změny ve schématu dat bez rozbití zpětné kompatibility mají všechny identifikátory předponu `api/v1/`. Úprava dat by tak znamenala zvýšení verze v předponě a tvorbu nových koncových bodů.

Kvůli obsáhlosti všech endpointů jsem jejich popis zde zjednodušil, plná dokumentace je k dispozici ve formátu OpenAPI v příloze práce[1].

`/auth` – Autentizace uživatele. Registrace nového účtu a získání informací o přihlášeném uživateli.

`/batch` – Vzdálený export a import frází. Přístup je podmíněn vygenerovaným API klíčem. Slouží pro integraci při vývoji softwaru.

`/invite` – Správa pozvánek. Uživatel pomocí tohoto endpointu získá seznam svých pozvánek, může je přijmou nebo odmítnout.

`/language` – Seznam podporovaných jazyků. Identifikátory jazyků jsou zvoleny podle IETF BCP 47.

`/user` – Správa uživatelského účtu. Umožňuje číst cizí účty za podmínky, že s ním má přihlášený uživatel společný projekt.

`/project` – Tvorba a správa projektu.

`/project/{id}/apiKey` – Správa přístupového klíče určeného pro autentizaci v koncových bodech `/batch`.

`/project/{id}/invite` – Vytvoření nové pozvánky pro účast na projektu.

`/project/{id}/language` – Správa jazyků v projektu.

`/project/{id}/member` – Správa členů projektu.

`/project/{id}/oneTimeExport` – Určeno pro tvorbu exportů ve webovém uživatelském rozhraní. Viz entita *One Time Export* z doménového modelu.

`/project/{id}/term` – Správa frází v projektu. Požadavky na tento koncový bod používají kurzorové stránkování. Fráze jsou tak vraceny po skupinách o velikosti danou parametrem `count`.

`/project/{id}/term/{id}/translation` – Správa překladů fráze.

[1] Je dostupná na server pod adresou `/swagger` a lze jí nalézt i ve složce winder/app/src/main/resources/openapi/documentation.yaml

6.5 Drátěný model

Již z nefunkčního požadavku *N3* jsem si stanovil hlavní cíl při návrhu uživatelského rozhraní; přehlednost. Před implementací frontendu jsem vytvořil drátěný model, který měl zobrazovat jeho základní rysy, oproštěné od barev či jiných designových prvků.

Pro vytvoření modelu jsem použil nástroj *Figma* [39], který umožňuje navržené obrazovky udělat klikatelné a otestovat tak, jak by probíhala uživatelova interakce.

Celé rozhraní je navrženo z pohledu vývojáře či administrátora projektu. Jediná změna zobrazení pro překladatele je neviditelnost přepínačů záložek na navigační liště.

Ukázka z drátěných modelů je k dispozici v příloze B, celý ho lze nalézt ve složce `wireframe` v přílohách práce. Následuje popis jednotlivých obrazovek.

6.5.1 Domovská obrazovka

Začal jsem návrhem společných komponent jako navigační lišty, tlačítek a základním rozložením rozhraní. Již po této fázi jsem přistoupil ke tvorbě domovské obrazovky, na které si uživatel zvolí projekt se kterým chce pracovat. Má i možnost vytvořit si nový. Vyhotovený návrh je vidět na obrázku B.1.

6.5.2 Jazyky

Po vybrání projektu je uživatel přesměrován na seznam jazyků (viz obrázek B.2), které jsou jím podporovány. Koncepčně se jedná o záložku „překlady“, která je určená primárně překladatelům, využívat ji nicméně mohou i ostatní členové. Vybrání položky v tomto seznamu znamená zvolení si jazyka, do kterého chce uživatel překládat. Následně jsou mu tak ukázány dostupné fráze.

6.5.3 Překladání

Při překládání má uživatel možnost si zvolit výchozí jazyk, ze kterého bude čerpat a v seznamu jsou mu pak zobrazeny fráze a jejich překlady. V levém sloupci jsou vidět texty ve zvoleném výchozím jazyce, v pravém má možnost upravovat překlady. Na obrázku B.3 je vidět překlad z češtiny do angličtiny.

6.5.4 Fráze

Správa frází je určená pro vývojáře a administrátory. Rozhraní vypadá podobně jako v záložce s překlady, navíc lze vidět všechny překlady najednou. Fráze může uživatel přidávat a mazat. Úprava názvu pomocí probíhá po rozkliknutí jejich řádku. Návrh je vidět na obrázku B.4.

6.5.5 Nastavení projektu

Poslední hlavní záložkou je nastavení. To je rozděleno do tří sekcí pomocí karet. V jedné z karet lze vidět členy, kteří na projektu spolupracují. Po jejím rozkliknutí se uživatel dostane na stránku, kde je může spravovat. Další sekce zobrazuje podporované jazyky a stejně tak umožňuje po kliknutí přejít na jejich úpravu.

Nedílnou součástí je karta, kde lze provádět akce nad projektem. Podle role člena jsou zde zobrazeny různé možnosti. Například na obrázku B.5 se jedná o pohled administrátora. Ten má tak schopnost projekt přejmenovat, smazat či vygenerovat klíč pro vzdálený přístup přes API. Tlačítka import a export se zobrazují i vývojářům.

6.5.6 Testování drátěného modelu

Po návrhu drátěného modelu se uskutečnilo jeho otestování na dvou osobách. Vyhotovil jsem dva testovací scénáře, pokrývající aktéry ze sekce 4.2 a to pro překladatele a administrátora. Scénář pro aktéra vývojář jsem netvořil z důvodu, že administrátor může vykonávat stejné případy užití a jejich role se v reálném projektu často překrývají. Pro zbylé aktéry také není třeba scénáře tvořit, jsou pokryty výše zmíněnými, nebo jsou primitivní.

6.5.7 Scénář pro překladatele

- Vyberte projekt „Foobar“. Nadále v něm budete pracovat.
- Zvolte překládání do anglického jazyka.
- Zkuste najít způsob, jak si nastavit referenční jazyk.
- Přeložte frází, která v češtině obsahuje text „Souhlasíte s podmínkami?“.
- Zkuste najít způsob, jak mezi frázemi vyhledávat.
- Zkuste najít způsob, jak fráze filtrovat.
- Zkuste najít způsob přechodu na další stránku frází.
- Odhlaste se z aplikace.

6.5.8 Scénář pro administrátora

- Vytvořte projekt „Foobar“. Nadále v něm budete pracovat.
- Zobrazte si fráze v projektu.
- U fráze `button_continue` zobrazte její překlady.
- Frázi `error_network` přejmenujte.
- Smažte frázi `label_consent`.
- Nalezněte, jakým způsobem byste přidali novou frázi.
- Nalezněte způsob, jak lze vygenerovat API klíč.
- Jakým způsobem byste do projektu přizvali nového člena?
- Nalezněte způsob, jak lze do projektu přidat nové jazyky.
- Importujte do projektu existující fráze.
- Nalezněte možnost exportování frází.
- Přejmenujte projekt.
- Smažte projekt.

6.5.9 Výsledky testování

Z dvou proběhlých testů byla získána zpětná vazby a návrhy, které v implementaci znamenali následující odlišnosti od drátěného modelu:

- Sjednocení tlačítka pro přidání fráze a vytvoření projektu. Je vhodnější plovoucí forma tlačítka v pravém dolním rohu.
- Zobrazení do jakého jazyka probíha překlad. V drátěném modelu se zobrazuje pouze referenční jazyk a v obrazovce s překlady není nikde napsáno, do jakého jazyka má překladatel překládat.
- Přidání nadpisu ke kartě s možnostmi. Sjednotit jí tak s ostatními kartami na stránce s nastavením.
- Přidání tlačítka pro zobrazení překladů na stránce s frázemi. Nebylo jasné, že kliknutí na řádek vyvolá jejich zobrazení.
- Přidání tlačítka pro úpravu názvu fráze. Kliknutí na její název pro úpravu se ukázalo jako nedostačující.

Kapitola 7
Implementace

Kapitola se zabývá realizací prototypu platformy na základě předchozích návrhů a s vybranými technologiemi. Implementace je rozdělena na dvě části; serverovou aplikaci zahrnující především vývoj REST API pomocí frameworky Ktor a klientské webové rozhraní využívající knihovnu Compose HTML s nadstavbou Kobweb.

Název platformy jsem zvolil na základě anglického slova „string“, které v překladu znamená řetězec textu, ale může se také chápat jako provázek nebo šňůra. Protože jsem chtěl zdůraznit, že platforma má zlepšit správu překladů textů, přišlo mi vhodné ji nazvat „Ribbon“, tedy stuha. V jistém smyslu název reflektuje nejen účel celé platformy, ale i jeden z jejích cíl – být pro uživatele intuitivní či vzhledná.

Cílem je převedení teoretických konceptů do praxe a ověření tak správnosti návrhu. Jsou rozebrány hlavní výzvy, kterým jsem při realizaci čelil.

7.1 IDE

Před začátkem vývoje aplikace bylo nutné vybrat vývojové prostředí – IDE. Pro vývoj v Kotlinu se jako nejlepší jeví nástroje od samotných autorů jazyka, firmy JetBrains. V dnešní době nabízejí dvě možnosti:

- IntelliJ IDEA – Pokročilé integrované vývojové prostředí s dlouholetou tradicí a nepřebernými možnostmi. Obsahuje plnou podporu pro Kotlin a díky indexaci kódu nabízí pokročilou navigaci v projektu [40].
- JetBrains Fleet – Nový editor, který je momentálně poskytován v *preview* verzi. Měl by být jednodušší a rychlejší, ale zároveň podporovat plnohodnotnou navigaci jako v IntelliJ. Pokud vývojář zapne „Smart Mode“, využívá i stejný sytému indexace [41].

I když vývoj pomocí nového nástroje Fleet by mohl poskytnout zajímavou zkušenost a navíc možnost zpětné vazby vývojářům při nalezení nedostatků, rozhodl jsem se pro ověřené a stabilní vývojové prostředí IntelliJ IDEA.

7.2 Verzování

Při vývoji je důležité uchovávat historii zdrojového kódu. Pro tuto potřebu jsem využil nástoje *git* s napojením na vzdálený repozitář ve službě GitHub[1]. V budoucnu je možné do projektu přizvat další vývojáře a pomocí této služby na něm spolupracovat. Všechen kód je verzovaný v jednom repozitáři (*monorepo*), což zjednodušuje jeho správu. Alternativou by mohlo být vytvoření samostatných repozitářů pro frontend, backend a sdílené entity. To by ale bylo vzhledem k velikosti projektu kontraproduktivní.

Navíc, pomocí služby GitHub, lze platformu automaticky otestovat a nasadit, o tom více v sekci 9.3.

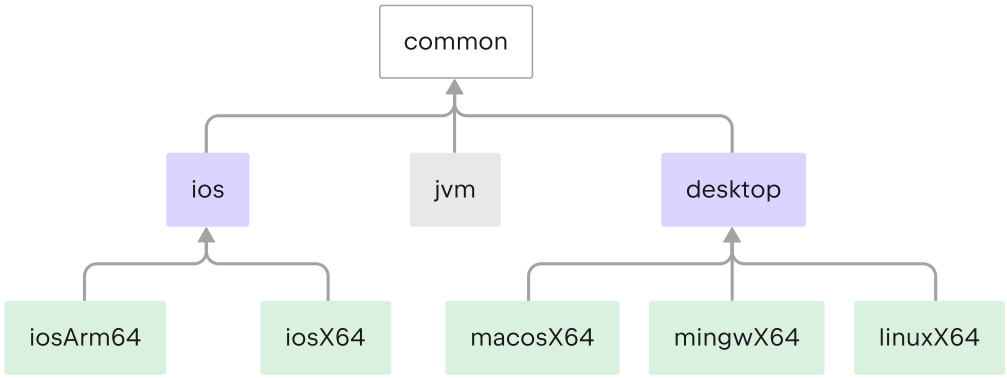
[1] `https://github.com/`

7.3 Struktura projektu

Zvolení správné struktury projektu lze vynutit architektonická pravidla a umožnit rychlejší kompilaci projektu. Pro sestavení projektu je použit standardní sestavovací nástroj *Gradle* [42], často používaný v Kotlin projektech[2].

V Gradlu probíhá sestavování projektu po takzvaných *subprojects*. To jsou samostatné jednotky kódu, které mají v souboru `build.gradle` určené závislosti a nastavenou konfiguraci. Podprojekty lze dělit na ještě menší jednotky zvané *source sets* s vlastními závislostmi a konfigurací. Podprojekt aplikuje *pluginy* (zásuvné moduly), které rozšiřují a automatizují jeho sestavení.

Jeden z takových pluginů poskytuje například Kotlin Multiplatform. Pomocí něj se nastavují cílové platformy kompilace projektu *targets*, každá mající vlastní source set. Společný source set *commonMain* pak obsahuje kód, který může být sdílen s ostatními.



Obrázek 7.1. Struktura KMP projektu, převzato z [43].

Projekt jsem tedy na základě těchto informací strukturoval co nejvíce modulárním způsobem s věnování pozornosti rychlému sestavení, zvolené architektuře a sdílení kódu.

Kořenová složka projektu obsahuje tři hlavní podsložky:

- `shared` – Jedná se o plnohodnotný Gradle modul, obsahující sdílený kód mezi frontendem a backendem. Zde lze nalézt entitní vrstvu aplikace.
- `winder` – Složka backendu platformy. Moduly v ní obsahují target pouze pro JVM.
- `unwinder` – Složka frontendu platformy. Moduly v ní obsahují target pouze pro JavaScript.

Podrobnou strukturu zobrazuje následující výpis, kde složky bez podsložek reprezentují gradle podprojekty (moduly). Oddělením implementace datových zdrojů do samostatného podprojektu jsem znemožnil použití Firebase v jiné než datové vrstvě a vynutil tak aplikování principů Clean Architecture (viz 6.3.4). Moduly funkcí jsou mezi sebou nezávislé (vyjma `shared`), což zrychlí sestavení projektu a zjednodušuje jeho provázanost.

[2] Při použití Kotlinu Multiplatform se dokonce jedná o jedinou možnost: `https://discuss.kotlinlang.org/t/fullstack-kotlin-with-maven/16008/2`

```
shared ..... sdílený modul entit
winder ..... složka obsahující backend
  app ..... vstupní bod backendu
  feature ..... moduly jednotlivých funkcí
    shared ..... sdílený modul backendu
    batch
    language
    project
    term
    translation
    user
  firebase ..... implementace datových zdrojů pomocí Firebase
unwinder ..... složka obsahující frontend
  web ..... vstupní bod webového frontendu
  feature ..... moduly jednotlivých funkcí
    shared ..... sdílený modul frontendu
    auth
    batch
    language
    project
    term
    translation
    user
```

7.4 Sdílené třídy

Realizaci platformy jsem začal tvorbou sdílených entit a základních tříd pro *UseCases*. Pro představu uvádím dvě hlavní entity, které jsou využívány při překládání.

7.4.1 Entity

Entita `Term` zastupuju frázi patřící pod projekt. Obsahuje identifikátor, který si volí uživatel při jejím vytvoření, volitelně popis, seznam překladů, metadata o autorovi a času změny a statistiky o její přeloženosti. Překlady jsou v entitě označené jako volitelný atribut. Backend je poskytuje na vyžádání a ve výchozím stavu se jejich seznam v entitě nevyskytuje.

```
@Serializable
data class Term(
    val id: String,
    val description: String?,
    val translations: List<Translation>?,
    val updated: Instant,
    val authorEmail: String?,
    val authorId: String?,
    val statistics: TermStatistics?,
)
```

Výpis kódu 7.1. Entita fráze.

Další entitou je samotný překlad `Translation`. Překlad je vždy veden pod nějakým jazykem, jehož identifikace probíhá na základě již zmíněného formátu IETF BCP 47. Dále obsahuje samotný text překladu a metadata o autorovi a času změny.

```
@Serializable
data class Translation(
    val languageId: String,
    val text: String,
    val updated: Instant,
    val authorEmail: String?,
    val authorId: String?,
)
```

Výpis kódu 7.2. Entita překladu.

7.4.2 UseCase

Další komponentou byznys vrstvy aplikace jsou případy užití. V nich nalezneme aplikační logiku. Pro sjednocení rozhraní jsem vytvořil základní třídu `UseCase`, která poskytuje operaci pro jeho spuštění. Je využito přetížení operátoru `invoke` (pomocí extension funkce) umožňující jeho spuštění obdobným způsobem, jako volání metody. Implementace tohoto operátoru je v backendu a frontendu rozdílná (viz výpis 7.4). Zatímco backend pouze zavolá metodu `execute`, na frontendu je výsledek obalen třídou `Result` a případné výjimky jsou odchyceny a převedeny na `Result.Failure`.

Rozdíl pramení ze způsobu vypořádávání se s chybami. Pro serverové aplikace je běžně využívání výjimek (*exceptions*), framework pro obsluhu požadavků je zachytí a klientovi vrátí chybnou hlášku. Nedojde tak k násilnému ukončení aplikace. Narozdíl od toho na frontendu jsou výjimky nežádoucí, nemá je kdo automaticky odchytit a pokud by na to vývojář zapomněl, mohlo by dojít k selhání celé aplikace. Používání obalovacích tříd typy `Result` je běžná praxe, která vede k nutnosti ověřovat úspěšnost požadované operace.

```
// Varianta se vstupním parametrem
interface UseCase<in Params, out Result> {
    suspend fun execute(params: Params): Result
}

// Varianta bez vstupních parametrů
interface UseCaseNoParams<out Result> {
    suspend fun execute(): Result
}
```

Výpis kódu 7.3. Základní třídy UseCase.

7.5 Backend

Následovala implementace serverové části aplikace. Jeho modul jsem nazaval winder, což odkazuje na anglické slovo ve smyslu „motat“. Vzhledem k názvu platformy to má naznačovat jeho roli zpracování dat z externích služeb a jejich poskytování v jednotném formátu.

```
// Implementace operátoru na backendu
suspend operator fun <Params, Result> UseCase<Params, Result>.invoke(
    params: Params,
) = execute(params)

// Implementace operátoru na frontendu
suspend operator fun <Params, Res> UseCase<Params, Res>.invoke(
    params: Params
): Result<Res> = runCatching { execute(params) }
```

Výpis kódu 7.4. Operátor invoke pro UseCase.

7.5.1 Ktor

Obsluha HTTP požadavků od klienta je vyřešena frameworkem Ktor. Při implementaci jsem použil velké množství jeho pluginů:

- ContentNegotiation – Slouží pro serializaci a deserializaci posílaných dat. Využívá knihovny kotlinx.serialization [44].
- CORS – Nastavení Cross-Origin Resource Sharing. Jde o zabezpečení API, které slouží k identifikaci domény, ze které je webovému prohlížeči dovoleno s ním komunikovat.
- Authentication – Autentizace uživatelů. Podporuje *Basic*, *Digest* a *Bearer* schéma, ale lze jej rozšířit o vlastního poskytovatele autentizace.
- StatusPages – Umožňuje nastavení reakce na výjimky v aplikaci v podobě adekvátní odpovědi s chybovým stavem.
- Resources (Type-safe routing) – Rozšíření klasického směrování o takzvané *resources*, třídy s definicí URI , parametrů a jejich datových typů. Viz následující blok kódu.
- SwaggerUI – Poskytování webového rozhraní swaggeru[3] na základě vygenerované OpenAPI dokumentace. Dostupné na adrese `/swagger`.
- Koin – Vkládání závislostí v těle požadavků. Více o něm v následující sekci.

Routy (či endpointu) se vytváří pomocí funkcí s názvem zvolené HTTP metody. Při použití pluginu *Resources* je navíc předáván typový argument, který slouží pro specifikaci cesty a případných parametrů. Ve výpisu 7.5 jsem uvedl definici endpointu pro vytvoření nového projektu pomocí metody POST. Posledním parametrem funkce je lambda funkce, spuštěná při požadavku na tento endpoint.

7.5.2 Koin

Pro zachování principu *Inversion of control*, důležitém pro správnou implementaci Clean Architecture bylo využito mechanismu vkládání závislostí pomocí knihovny Koin. Definice objektů, spravovaných touto knihovnou, se dělí na moduly. V implementovaném řešení má každý gradle podprojekt veřejně viditelný Koin modul. V podprojektu `app` pak dochází k registraci všech modulů do Koinu. Je tak možné vkládat závislosti, jejichž implementace není ostatním podprojektům viditelná.

Ve výpisu 7.6, ukazující modul uživatele, jsou k nahlédnutí dvě hlavní metody vkládání závislostí. Pomocí `singleOf` říkám Koinu, že má vytvořit jedinou instanci pro běh celé aplikace, zatímco `factoryOf` nutí vytvořit novou instanci při každém vložení. Pro

[3] `https://swagger.io/`

```
@Resource("/project")
class Project

fun Route.projectRoutes() {
    val createProject by inject<CreateProjectUseCase>()

   /**
    * Create a new project
    */
   post<Project> {
       val body = call.receive<ProjectCreation>()
       val project =
           withSentry {
               createProject(
                   CreateProjectUseCase.Params(
                       projectCreation = body,
                       user = requireSessionUser(),
                   ),
               )
           }

       call.respond(HttpStatusCode.OK, project)
   }
}
```

Výpis kódu 7.5. Definice route v Ktoru.

```
val featureUserModule = module {
    singleOf(::UserRepositoryImpl) bind UserRepository::class
    factoryOf(::RegisterUserUseCaseImpl) bind
       RegisterUserUseCase::class
    factoryOf(::GetUserUseCaseImpl) bind GetUserUseCase::class
}
```

Výpis kódu 7.6. Koin modul podprojektu uživatele.

Ktor existuje ještě takzvaný *Request Scope* umožňující vytvoření objektů vázaných na životní cyklus požadavku.

7.5.3 Serializace

Serializace je proces převádění datové struktury či objektu do jednoduše přenositelného proudu bajtů. Obecně je třeba serializovat objekty pro přenos dat mezi klientem a serverem. V implementaci využívám knihovnu *kotlinx.serialization* [44] pro serializaci i deserializaci[4] objektů ve formátu JSON. Entity stačí anotovat anotací `@Serializable`, o jejich konverzi se stará zmíněný plugin `ContentNegotiation`.

[4] Opak serializace.

7.5.4 Autentizace

Ačkoliv Ktor podporuje autentizaci pomocí tří zmíněných schémat, bylo pro podporu Firebase Authentication žádoucí vytvořit vlastního poskytovatele.

Očekává se, že klient odešle v žádosti hlavičku `Authorization` s hodnotou `Bearer` následovanou tokenem z Firebase. Tento token se na backendu ověří a pokud je správný, nastaví se takzvaný *Principal*. To je objekt autentifikované osoby, se kterým lze pracovat při zpracování požadavku.

```
class FirebaseAuthProvider(
    private val config: Config,
) : AuthenticationProvider(config) {
    override suspend fun onAuthenticate(
        context: AuthenticationContext
    ) {
        // Získání obsahu HTTP hlavičky Authorization
        val authHeader = context.call.request
            .parseAuthorizationHeader()

        // Ověření tokenu
         val token = firebase
            .verifyIdTokenAsync(authHeader.blob).await()

        // Získání uživatele
        val user = firebase
            .getUserAsync(token.uid).await()

        // Nastavení Principal
        val firebasePrincipal = FirebasePrincipal(user)
        context.principal(firebasePrincipal)
    }
}
```

Výpis kódu 7.7. Autentizační poskytovatel využívající Firebase.

Mimo tohoto poskytovatele vznikli další dva; pro ověření API klíče při vzdáleném přístupu a pro ověření jednorázového hesla pro export z webového rozhraní (viz entita „One Time Export“).

Při požadavku na chráněný zdroj se ověřují práva uživatele vykonat danou operaci pomocí `CheckProjectMemberPermissionUseCase`. Pokud uživatel například není členem projektu, nebo nemá potřebnou roli, je vyhozena výjimka a klientovi vrácena odpověď s chybovým stavem `Forbidden`.

7.5.5 Databáze

Pro usnadnění práce s NoSQL databází Firebase Firestore jsem vytvořil DAO objekty na základě schématu ze sekce 6.2. Uvádím příklad objektu pro frázi. Anotace `@DocumentId` říká, že anotovaná vlastnost má být namapována na ID dokumentu.

Za povšimnutí stojí `companion` objekt, který v sobě má textové reprezentace vlastností z DAO. To je nutné z důvodu limitace při filtraci či jiných složitějších operací.

```
data class TermDao(
    @DocumentId
    var id: String = "",
    var description: String? = null,
    var updated: Date = Date(),
    var authorEmail: String? = null,
    var authorId: String? = null,
    var translatedIn: List<String> = emptyList(),
    var notTranslatedIn: List<String> = emptyList(),
) {
    companion object Fields {
        const val DESCRIPTION = "description"
        const val TRANSLATED_IN = "translatedIn"
        const val NOT_TRANSLATED_IN = "notTranslatedIn"
        const val UPDATED = "updated"
        const val AUTHOR_ID = "authorId"
        const val AUTHOR_EMAIL = "authorEmail"
    }
}
```

Výpis kódu 7.8. DAO fráze.

Firebase SDK při konstrukci takového dotazu vyžaduje textový název atributu a neumí ho přečíst z názvu vlastnosti DAO třídy.

Čtení dat z databáze probíhá způsobem zobrazeným ve výpisu 7.9. Nejdříve se funkcí `firestore.term` vytvoří objekt `DocumentReference`, sloužící pouze jako cesta ke konkrétnímu dokumentu. Následné až funkcí `readObjectOrNull`, s typovým parametrem označující čtené DAO, se celý dokument z databáze stáhne a převede na instanci `TermDao`. Funkce `toDomain` z něj udělá doménovou entitu `Term`.

Ještě doplním, že zmíněná `firestore.term` je vlastní extension funkce pro přístup k DocumentReference. Motivace pro tvorbu bylo sjednocení rozhraní poskytovaného knihovnou Firestore ve všech *DataSouce* implementacích. Její zjednodušenou definici lze také nalézt ve výpisu 7.9.

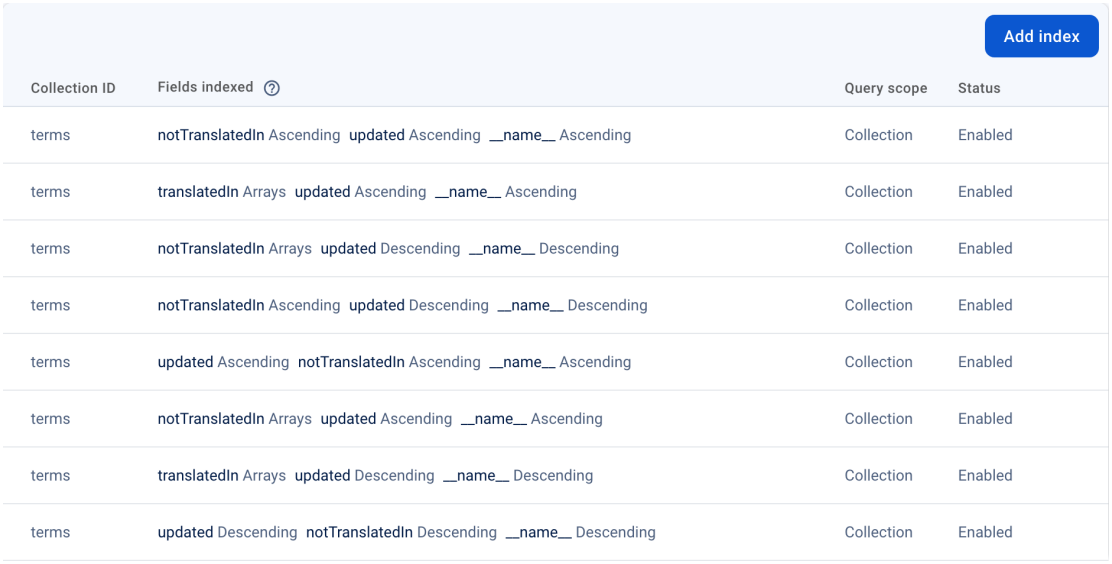
Firebase Firestore podporuje i transakce a dávkové zápisy, sloužící k optimalizaci dotazů. Oba mechanismy zaručují atomicitu, nemůže tedy dojít k provedení jen části operací. Narozdíl od dávkových zápisů můžou transakce provádět i čtecí operace, které se musí nacházet před zápisovými. Pokud by při vykonání transakce došlo ke změněn čtených dat, momentální stav se zahodí a transakce je spuštěna od začátku. Dávkové zápisy jsou tak často díky jednoduššímu plánování rychlejší a je vhodné je využívat, pokud není třeba zaručit konzistenci čtených dat. Při implementaci jsem v několika případech využil oba tyto mechanismy.

Indexace dat probíhá pro každý atribut dokumentu automaticky a jednoduché dotazy[5] lze provádět ihned po přidání dat. Ukázalo se, že pro složitější dotazy, jako hledání frází s aplikovaným filtrem a řazením, je nutné vytvořit složené indexy. Při složitém dotazu bez něj Firestore selže a vygeneruje odkaz, na kterém nabídne tvorbu těchto indexů. Nakonec je v databázi 8 složených indexů, starajících se o filtrování frází, viz obrázek 7.2.

[5] S podmínkou na jeden atribut.

```
internal fun Firestore.term(
    projectId: String,
    termId: String,
) = collection("projects")
    .document(projectId)
    .collection("terms")
    .document(termId)

override suspend fun getTerm(
    projectId: String,
    termId: String,
): Term? {
    return withSentrySpan("db.term.get") {
        firestore.term(projectId, termId)
            .readObjectOrNull<TermDao>()
            ?.toDomain(
                projectId = projectId,
                includeLanguageIds = emptyList(),
                includeAllLanguages = true,
            )
    }
}
```

Výpis kódu 7.9. Čtení frází pomocí Firebase Firestore.



Obrázek 7.2. Složený index v databázi Firebase Firestore.

7.5.6 Sentry

Monitoring backendu probíhá ve službě Sentry. Ta vývojářům bohužel neposkytuje SDK pro integraci do Ktoru. Bylo tedy nutné vytvořit vlastní plugin. Ktor je naštěstí pro vývoj pluginů dobře připraven a vystavuje různé metody pro napojení na zpracování požadavků. Výsledný plugin propojující Sentry s Koterm je vidět ve výpisu 7.10.

```
val SentryPlugin = createApplicationPlugin("SentryPlugin") {
    on(MonitoringEvent(Routing.RoutingCallStarted)) { call ->
        val transaction = Sentry.startTransaction(
            call.route.toString(),
            "http.request",
            TransactionOptions().apply {
                isBindToScope = true
            },
        )

        transaction.setTag("method", call.request.httpMethod.value)
        // Přidání transakce do atributů požadavky
        call.attributes.put(
          SentryDefaults.transactionKey,
            transaction
        )
    }

    on(ResponseBodyReadyForSend) { call, content ->
        val transaction = call.sentryTransaction ?: return@on
        if (transaction.status == null) {
            // Nastavení stavu podle HTTP status kódu
            transaction.status = content.status?.value
                ?.let(SpanStatus::fromHttpStatusCode)
        }
    }

    on(ResponseSent) { call ->
        val transaction = call.sentryTransaction ?: return@on
        call.response.status()?.let { status ->
            // Nastavení stavu podle HTTP status kódu
            transaction.status = SpanStatus
                .fromHttpStatusCode(status.value)
        }
        // Ukončení transakce
        transaction.finish()
    }

    on(CallFailed) { call, cause ->
        val transaction = call.sentryTransaction ?: return@on
        // Asociace výjimky s transakci
        transaction.throwable = cause
        transaction.status = SpanStatus.INTERNAL_ERROR
    }
}
```

Výpis kódu 7.10. Plugin pro monitorování požadavků pomocí Sentry.

Transakce je započata při události `Routing.RoutingCallStarted`, kterou rozesílá *Routing* plugin ihned po nalezení vhodného endpointu. Pro identifikaci transakcí se používá cesta a HTTP metoda endpointu.

Při volání UseCase v požadavku je pak transakce předávána v kontextu Kotlin coroutiny s využitím `AbstractCoroutineContextElement`. To umožňuje jakékoliv funkci transakci získat a vytvořit v ní nový *span*, jako například v již uvedeném výpisu 7.9.

Při ukončení požadavky se transakce uzavře, přidá se informace o jejím úspěchu ve formě HTTP status kódu a odešle se do databáze Sentry. Navíc, když dojde k vyhození neočekávané výjimky, zmíněný plugin `StatusPages` to zaznamená a do Sentry o tom zašle událost.

Po tomto napojení lze ve webovém rozhraní služby vidět čas trvání transakcí, jejich spanů, monitorovat výkon a sledovat případné chyby. Výsledek lze vidět na obrázku 5.3 v dřívější kapitole.

7.5.7 Export a import

Důležitou součástí backendu je schopnost provádět exportování a importování frází a jejich překladů. Díky zaměření platformy primárně na překlad textů v mobilních aplikací jsem ze věnoval podpoře dvou formátů:

- Android XML – Formát používaný při vývoji nativních android aplikací. Soubor typu XML obsahující jeden `resource` tag a v něm seznam objektů `string` s atributem `name` a tělem obsahující zobrazovaný text [45].
- Apple Strings – Formát určený pro překlad iOS, Mac OS a dalších aplikací pro systémy od Applu. Jednoduchý zápis ve stylu `"klíč" = "hodnota";` [46].

V obou formátech lze v textu použít formátovací řetězce pro nahrazení textu při běhu aplikace. Většina specifikátorů je mezi oběma formáty identická, inspirovaná klasickou `printf` funkcí z jazyka C [47]. Jediný rozdíl je ve specifikátoru pro nahrazení textu. Zatímco Android používá `%s`, Apple `%@`. Ačkoliv Apple má i specifikátor `%s`, je určen pro nahrazení klasickým polem znaků ukončeným nulovým bajtem. Oproti tomu `%@` slouží pro textovou reprezentaci Objective-C objektu [48]. V praxi se tak víceméně první varianta nepoužívá. Většinou je potřeba nahrazovat za objekt `String` jehož textová reprezentace je on samotný. Pro vzájemnou kompatibilitu mezi formáty jsem zvolil reprezentaci využívanou Applem. Nerozbijí se tak texty, které v Apple aplikacích používají `%s` (ačkoliv je jich minimum) a při exportu do Android formátu se specifikátory převedou na `%s`.

Export zajišťuje UseCase `ExportRibbonsUseCase`, který dále deleguje práci podle požadovaného formátu. Import funguje obdobně, UseCase `ImportRibbonsUseCase` použije pro formáty rozdílné parsery. Zde nastal drobný problém. Ačkoliv soubor ve formátu Android, lze deserializovat pomocí knihovny *kotlinx.serialization* a jejím rozšířením o XML [49], pro formát Apple žádný takový nástroj neexistuje. Přistoupil jsem tedy k implementaci vlastního parseru tohoto formátu, který funguje na principu konečného automatu. Jeho kód lze nalézt ve třídě `AppleStringsParserImpl`. Podporuje plnou paletu možností, které tento formát nabízí, včetně víceřádkových komentářů a escapování sekvencí. Při selhání umí detekovat, kde přesně chyba nastala.

Mimo zmíněné dva formáty pro vývoj mobilních aplikací jsem vytvořil vlastní jednoduchý JSON formát „Ribbon“, který je možné také používat.

7.6 Frontend

Po dokončení implementace backendu jsem přistoupil k vytvoření klienta v podobě webového rozhraní. Modul byl pojmenován „unwinder“ ve smyslu opačné role frontendu; „rozbalení“ a prezentace dat. Snímky obrazovky z hotové implementace lze vidět v příloze C.

7.6.1 Design

V nefunkčním požadavku *N3 – Vzhled* jsem si vytyčil použití designového systému Material Design 3 [18]. Bohužel, pro Compose HTML neexistuje jeho implementace a tak jsem musel komponenty uživatelského rozhraní tvořit sám podle specifikací. Vytvořené komponenty lze nalézt v balíčku `unwinder.web.components.material` a zahrnují například tlačítka, textové vstupy, typografii, zaškrtávací políčko a mnohem více. Při implementaci jsem dbal na nefunkční požadavek *N1 – Dostupnost webového rozhraní*. Komponenty jsou navrženy s ohledem na responzivitu.

Použitou paletu barev jsem vygeneroval pomocí nástroje *Material Theme Builder* [50]. Stačilo zvolit výchozí barvu a stáhnou vygenerované definice pro Compose.

7.6.2 Logo

Aby bylo platformu možné snadno identifikovat, vytvořil jsem pro ní logo. Základem mi posloužila ikona `ribbon` z kolekce *Font Awesome* [51], která je distribuována pod licencí *SIL OFL 1.1* [52]. Je ji možno použít bez jakýchkoliv omezení.

Následná úprava proběhla v aplikace *Figma* [39], kde jsem ikoně přidal pozadí a změnil barvy tak, aby respektovaly vygenerovanou paletu Material Design.

Obrázek 7.3. Logo platformy.

7.6.3 Lokalizace

Frontend by podle nefunkčního požadavku *N2 – Lokalizace* měl být snadno přeložitelný do jiného, než anglického jazyka. Pro jeho lokalizaci jsem tak použil knihovnu *Libres* [53], určenou pro správů textů a obrázků v Kotlin Multiplatform.

Texty jsou spravovány ve formátu `.xml`, velice podobným způsobem, jak je tomu při vývoji Android aplikací. Jediným rozdílem je způsob použití řetězcových šablon. Text, který je třeba nahradit, stačí označit `${pojmenování náhrady}`. Podobá se tak řetězcovým šablonám v jazyku Kotlin. Ukázka takového souboru je vidět ve výpisu 7.11.

Navíc, díky podobnosti formátu, jsem lokalizace spravoval ve tvořené platformě a rovnou tak testoval její funkčnost.

V kódu se k textu přistupuje pomocí třídy `Res.string`, která v sobě obsahuje proměnné s názvem frází. Třída je automaticky generovaná při kompilaci.

7.6.4 Komunikace s API

Pro přístup k REST API poskytovaného backendem jsem použil klientskou variantu knihovny Ktor. Ta, stejně jako na serveru, má plugin `ContentNegotiation` pro automatickou serializaci a deserializaci objektů. Jak je vidět ve výpisu 7.12, lze velice

```
<resources>
    <string name="app_name">Ribbon</string>
    <string name="title_hello">Hello ${name}</string>
    <string name="title_your_account">Your account</string>
</resources>
```

Výpis kódu 7.11. Ukázka lokalizačního souboru Libres.

jednoduše vytvořit požadavek na server. Extension funkce na klientovi `use` je mnou vytvořená a slouží pouze pro odchycení výjimek Ktoru a jejich převod do vlastních výjimek aplikace.

I na klientovi bylo potřeba vytvořit poskytovatele autentizace, využívající Firebase Authentication. Místo ověřování token generuje a odesílá v HTTP hlavičce `Authorization`.

```
return httpClient.use {
    get("api/v1/project/$projectId/term") {
        parameter("size", size)
        parameter("cursor", cursor)
        parameter("query", query)
        parameter("filter", filter)
        parameter("filterLanguage", filterLanguage)
        parameter("order", order)
        includeLanguageIds.forEach { language ->
            parameter("includeLanguage", language)
        }
    }.body<Paging<Term>>()
}
```

Výpis kódu 7.12. Tvorba požadavku pomocí Ktor client

7.6.5 Multiplatform Settings

Pro uložení hodnot do paměti prohlížeče `LocalStorage`, jsem použil knihovnu *Multiplatform Settings* [54]. Její velkou výhodou je funkčnost na všech platformách podporovaných KMP s využitím specifického úložiště platformy. Při případném vývoji dalšího klienta by tak nebylo třeba používat další řešení. Umožňuje navíc ukládání komplexních objektů pomocí serializace s využitím zmíněné `kotlinx.serialization`.

V implementaci se knihovna využívá na ukládání informací o přihlášeném uživateli a jeho projektech, pro zrychlení načítání často používaných dat.

7.6.6 Compose HTML

Tvorba uživatelského rozhraní, tedy *View* vrstvy, probíhala pomocí knihovny Compose HTML. Komponenty se tvoří pomocí standardních Kotlin funkcí s anotací `@Composable`. Proto se jim někdy říká composable funkce. Tyto funkce jsou speciální svým chováním, emitují (neboli vysílají, generují, vytvářejí) UI. Emitováním UI se rozumí tvorba uzlů ve virtuálním stromu reprezentující uživatelské rozhraní. Celý proces vykreslování, nazavný *Composition* postupně skládá komponenty (uzly) v jeden celek. V Compose HTML se modifikuje přímo DOM webové stránky, ostatní varianty

Compose používají pro vykreslení platformě závislé grafické plátno (zástupcem je například již zmíněná SKIA).

Všechny varianty Compose na pozadí používají takzvaný *Compose Runtime*, který se stará o rekompozice (překreslení komponenty nebo jejich částí), správu stavu a spouštění vedlejších efektů (*Side Effects*).

Composable funkce přijímá parametry, které určují zobrazovaná data, nebo modifikují její vzhled. Navíc lze využít takzvaného *Slot API*, neboli mechanismu předávání composable lambda funkcí, které už sami o sobě emitují UI.

Speciálním, a zpravidla volitelným, parametrem většiny composable funkcí je `Modifier`. Ten umožňuje, jak již z názvu vyplývá, modifikovat vzhled a chování komponenty. Mezi modifiery patří například `size` pro nastavení velikosti, `padding` nastavující odsazení vnitřního obsahu od okraje komponenty nebo `onClick` pro vyvolání akce při kliknutí.

Ve výpisu 7.13 lze nahlédnout na funkci `RadioButtonItemGroup`. Ta sice nevyužívá pouze funkce čistého Compose HTML, ale i rozšíření Kobweb. To s cílem poskytnout lepší ukázku frameworku Compose jako takového. Parametr `content` je sám o sobě composable funkcí a slouží tak jako obsah komponenty. Funkce nad ním vytváří pouze obal a pomocí `Column` jej skládá pod sebe. Pro úpravu ze strany volajícího je předáván `modifier` parametr.

```
@Composable
fun RadioButtonItemGroup(
    modifier: Modifier = Modifier,
    content: @Composable ColumnScope.() -> Unit,
) {
    Column(
        modifier = modifier
            .gap(SpaceSmall),
    ) {
        content()
    }
}
```

Výpis kódu 7.13. Composable funkce.

7.6.7 Kobweb

Pro propojení jednotlivých komponent v Compose HTML jsem využil knihovny Kobweb. Ta poskytuje rozhraní pro tvorbu stránek, do kterých lze uživatele směrovat. Dále se stará o export stránky pro nasazení a přidává komponenty a funkce z klasického Compose. Jak lze vidět ve výpisu 7.14, funkci, kterou považujeme za stránku, stačí anotovat anotací `@Page` s parametrem její cesty. Za cílem snažšího směrování mezi stránkami jsem si vytvořil extension funkce nad *routerem*.

Pro znepřístupnění stránek bez přihlášení, jsem zhotovil funkci s vedlejším efektem `RequireAuthenticationEffect`, která přesměruje nepřihlášené uživatele na stránku s přihlášením. Vedle této vznikly i varianty pro ověření role a oprávnění v projektu.

Funkce `rememberPageContext()` vrací jakýsi kontext stránky. V něm lze nejen přistoupit ke zmíněnému *routeru*, ale i k případným parametrům stránky. Ty jsou dvojího typu:

- V cestě (*path parameter*) – Nacházejí se v adrese stránky, například v detailu projektu jde o jeho ID, které se vyskytuje za lomítkem `project/11z8...`.
- V dotazu (*query parameters*) – Nacházejí se až za cestou a znakem otazníku. Například řazení a filtrace frází `/terms?order=updated_desc&filter=translated`.

Proměnná `style` v sobě obsahuje definici palety barev. Instance této palety se získává z aktuálního barevného módu pomocí `CompositionLocal`[6]. Aplikace momentálně podporuje pouze světlý režim. Díky tomuto je připravena i pro režim tmavý, vyžadoval by to budoucí vývoj.

Každá stránka má svůj *ViewModel*, který se stará o logiku zpracování vstupů a zobrazovaných dat. Jeho návrhu se budu věnovat v další sekci.

```
fun Router.routeToProjects() = tryRoutingTo("/projects")

@Page("/projects")
@Composable
fun ProjectsPage() {
    RequireAuthenticationEffect()
   val ctx = rememberPageContext()
    val style = ColorMode.current.toSitePalette()
    val viewModel = rememberViewModel<ProjectsViewModel>()

    LaunchedEffect(viewModel) {
        viewModel.onAppear()
    }

    ...
}
```

Výpis kódu 7.14. Definice stránky v Kobwebu.

7.6.8 ViewModel

Logika zpracování vstupů a stav dat zobrazovaných na stránce jsou umístěny ve *ViewModelu*. Nejprve jsem vytvořil jednotné rozhraní, které pak implementují *ViewModely* specifických stránek.

Jediným způsobem, kterým *ViewModel* poskytuje data *View*, je přes jeho stav, jiné veřejné proměnné by se v něm neměli vyskytovat. Navíc použití `mutableStateOf` z *Compose Runtime* umožní composable funkcím stav sledovat a při jeho změně se automaticky aktualizovat. Pro usnadnění změny stavu byla vytvořena funkce `update`, ta ho nastaví na vrácenou hodnotu svého lambda parametru.

Každý *ViewModel* si v sobě nese `CoroutineScope`, v němž spouští podprogramy (*coroutines*). Jeho instance k němu může přistupovat pomocí proměnné `coroutineScope` nebo funkcí `launch`. Ta spustí parametr `block` v novém podprogramu na zmíněném kontextu.

Abych konstrukci instance *ViewModelu* usnadnil, implementoval jsem funkci `rememberViewModel`. Ta vytvoří nový `CoroutineScope`, který spravuje Compose. Nemůže se tak stát, že by v něm spuštěný podprogram zůstal běžet i po schování

[6] Mechanismus pro předávání proměnných mezi composables.

```
abstract class ViewModel<State : Any>(initialState: State) {
    var state by mutableStateOf(initialState)
        private set

    lateinit var coroutineScope: CoroutineScope

    protected fun update(action: State.() -> State) {
        state = action(state)
    }

    protected fun launch(
        context: CoroutineContext = EmptyCoroutineContext,
        body: suspend () -> Unit,
    ): Job {
        return coroutineScope.launch(context) { body() }
    }
}
```

Výpis kódu 7.15. Základní třída ViewModelu.

stránky. Následně se pomocí Koinu, získá nová instance *ViewModelu* specifikovaného typu `T`. Z funkce je pak vrácena tato instance s nastaveným `coroutineScope`, obalená v `remember`. Jedná se o funkci která si mezi rekompozicemi pamatuje vrácenou hodnotu svého lambda parametru, dokud nedojde ke změně jejích ostatních parametrů, označovaných jako *keys*. Pokud dojde k jejich změně, lambda parametr se spustí znova a hodnota v vrácena z `remember` změní.

```
@Composable
inline fun <reified T : ViewModel<*>> rememberViewModel(
    vararg keys: Any?,
): T {
    val coroutineScope = rememberCoroutineScope()
    val vm = koinInject<T>()
    return remember(vm, coroutineScope, *keys) {
        vm.coroutineScope = coroutineScope
        vm
    }
}
```

Výpis kódu 7.16. Pomocná funkce pro vytvoření instance ViewModelu.

Kapitola 8
Testování

8.1 Jednotkové testování

Unit testing, česky jednotkové testování, je jedna ze základních metod automatického testování. Klade si za cíl otestovat co možná nejmenší části zdrojového kódu (zpravidla metody či třídy). Čím více takovýchto jednotek testy pokrývají, tím více si můžeme býti jistí správností a funkčností celku.

Návrh samotných testů by měl dbát na jejich co největší míru izolovanosti. To znamená, že test by měl být opakovatelný, spustitelný sám o sobě a bez vedlejších efektů na stav aplikace. Za tímto účelem jsou vytvářeny pomocné objekty, anglicky „mocks“, které simulují chování částí tříd používaných testovanou jednotkou. Doba trvání testu by měla být co nejkratší, protože je žádoucí je spouštět často.

Díky zvolené architektuře se přímo nabízí testovat případy užití (*Use Cases*), jelikož jde o jediné veřejné rozhraní byznys vrstvy. Navíc jejich vlastnost izolovanosti od implementačních detailů nižších vrstev zajišťuje aktuálnost testů i při budoucím vývoji.

Testy jsem pokryl pouze modul backendu, jelikož webové rozhraní neobsahuje téměř žádnou byznys logiku a v podstatě jen zobrazuje data jemu poskytována. Pokud ale v budoucnu dojde k jeho rozšíření o složitější operace, je žádané takové testy vytvořit.

Pro otestování backendu platformy jsem zvolil knihovnu *JUnit 5* [55] s využitím zásuvného modulu pro *Koin*. Pro jazyk Java existuje populární framework pro vytváření pomocných objektů *mockito* [56]. Ačkoliv ho lze využít i v Kotlinu, hledal jsem nativní knihovnu. Hlavním důvodem byla hezčí syntaxe využívající konstrukce jazyka Kotlin a možnost testování multiplatformních modulů, kde Java knihovny nelze použít. Nakonec jsem tedy zvolil knihovnu *MockK* [57], která tyto kritéria splňuje. V rámci implementace jsem vytvořil 42 jednotkových testů, které pokrývají základní funkcionalitu. Jde o testy zaměřené na kontrolu rolí uživatelů, správu projektu, export a import ve všech dostupných formátech. I když jejich objem by mohl být větší, hlavní případy užití jsou pokryty.

8.1.1 Příklad jednotkového testu

Jako ukázku testování jsem zvolil test který vyzkouší, že překladateli nebylo v případu užití `CheckProjectMemberPermissionUseCase` uděleno oprávnění k vytvoření nové fráze. Pomocí příkazu `declareMock` s typovým parametrem `ProjectSource` vytváříme pomocný objekt na kterém definuji chování metody `getProjectMembers`. a to konkrétně tak, že při každém volání s jakýmkoliv parametrem (v tomto případě `id` projektu) metoda vrátí seznam s jedinou položkou testovaného překladatele.

Pro spuštění případu užití se používá `suspend` metoda, a tak je nutné ho obalit pomocí `runBlocking`. Tato funkce spustí svůj parametr (tj. poskytnutou lambda funkci) a zablokuje přitom hlavní vláknu. V podstatě tak spustí případ užití bez koprogramů.

Pro ověření funkčnosti pak funkce `assertThrows` očekává vyhození výjimky specifikované typovým parametrem (`AuthException.Forbidden`). Pokud je taková výjimka vyhozena, test je prohlášen za úspěšný, v opačné případě selže.

```
@Test
fun `translator is not granted CREATE_TERM permission`() {
    declareMock<ProjectSource> {
        coEvery { getProjectMembers(any()) } returns
            listOf(
                ProjectMember(
                    user = Mocks.user,
                    role = MemberRole.TRANSLATOR,
                ),
            )
    }

    runBlocking {
        assertThrows<AuthException.Forbidden> {
            useCase(
                CheckProjectMemberPermissionUseCase.Params(
                    projectId = "",
                    user = Mocks.sessionUser,
                    permission = RolePermission.CREATE_TERM,
                ),
            )
        }
    }
}
```

Výpis kódu 8.1. Jednotkový test pro CheckProjectMemberPermissionUseCase.

8.2 Statická analýza

Statická analýza je automatická metoda testování, při níž se netestuje běh programu, pouze se vyhodnocuje kvalita kódu na základě jeho struktury a vlastností. Pomáhá s identifikací chyb a slabin v kódu, ještě před jeho spuštěním. Tím zlepšuje kvalitu a bezpečnost softwaru a snižuje náklady a časové nároky spojené s opravami a úpravami po nasazení.

Jeden z druhů statické analýzy je takzvaný „Lint“. Ten funguje na základě sady pravidel, které definují správnou syntaxi, styl psaní kódu, různé konvence a další aspekty, které mají vliv na kvalitu kódu. Při spuštění se prochází zdrojový kód aplikace a jakákoliv odchylka zahlásí chybu. Vývojáři je pak doporučeno kód opravit.

Jelikož použití Lint nástroje nevyžaduje veliký zásah do existujícího kódu a navíc vynucuje použití zaběhlých konvencí i při budoucím vývoji, využil jsem ho při implementaci platformy. Zvolil jsem knihovnu KtLint [58], určenou přímo pro Kotlin. Ta navíc umožňuje automatické úpravy kódu na základě zmíněných pravidel. Pravidla lze definovat pomocí souboru `.editorconfig` [59]. Ten je podporovaný většinou IDE a pravidla jsou tak reflektována i při psaní kódu. Většina nastavení byla zanechána ve výchozím stavu. Následují tak oficiální doporučení pro styl Kotlin kódu.

KtLint je poskytován jako samostatný nástroj pro příkazovou řádku a každý vývojář si ho tak musí nainstalovat na vlastním zařízení. Pro snažší integraci s projektem jsem zvolil zásuvný modul do používaného systému na sestavení „KtLint Gradle“ [60].

Ten pak stačí aktivovat v konfiguraci projektu a vývojáři můžou nástroj využít bez dodatečné instalace.

8.3 Uživatelské testování

Tento druh testování je důležitý pro vyhodnocení správného návrh rozhraní a pomůže identifikovat jeho slabiny. Probíhá pozorováním uživatelů při používání aplikace a následné diskuzi o nejasnostech či problémech, na které narazili. Výstup z něj je nejen zpětná vazba testerů, ale i odhalení případných nedostatků rozhraní, kterých si vývojář nevšiml.

Proto jsem vytvořil testovací scénáře, podle kterých se vybraní testeři řídili. Platforma již běžela v produkčním prostředí a ti k ní přistupovali ze svého zařízení.

Testovací scénáře jsem rozdělil do dvou kategorií podle aktérů. Zatímco první scénář simuluje chování externího překladatele přizvaného do projektu, druhý scénář by měl naznačovat chování administrátora při různých úkonech se správou projektu. Pokrývají tak většinu funkcí celé platformy, jak ukazují případy užití naznačené u jednotlivých kroků.

Testování se zúčastnilo pět testerů, kterým jsem náhodně přidělil jeden z již zmíněných testovacích scénářů:

8.3.1 Překladatel

1. Přihlaste se do aplikace s následujícími údaji: *(UC2)*
   - email: test3@example.com
   - heslo: Heslo123
2. Přijměte pozvánky do projektu „Ribbon“ *(UC13)*
3. Přejděte do projektu „Ribbon“ *(UC7)*
4. Zvolte si překlad do češtiny *(UC18)*
5. Jako svůj referenční jazyk nastavte angličtinu *(UC18)*
6. Zobrazte si pouze nepřeložené fráze *(UC20)*
7. Vyhledejte mezi nimi frázi `label_show_statistics` *(UC20)*
8. Přeložte tuto frázi do češtiny *(UC19)*
9. Odstraňte překlad fráze `title_modify_member`, najděte ji ručně mezi přeloženými frázemi *(UC19)*
10. Změňte své jméno na „Překladatel“ *(UC3)*
11. Odhlaste se z aplikace

8.3.2 Administrátor

1. Registrujte se do aplikace pomocí vámi zvolených údajů *(UC1)*
2. Vytvořte nový projekt s názvem „Testovací projekt“ *(UC6)*
3. Přidejte do projektu jazyk „English (United States)“ *(UC8)*
4. Vytvořte novou frázi s názvem „label_info“ *(UC22)*
5. Rovnou přidejte její překlad v angličtině (jakýkoliv) *(UC19)*
6. Přidejte druhou frázi `net_error` s popiskem „Network error, when a connection is interrupted“ *(UC22)*
7. Frázi nepřekládejte, přizvěte do projektu překladatele s emailem test2@example.com *(UC13)*
8. Vyčkejte než překladatel frázi přeloží
9. *Fráze byla přeložena překladatelem*

10. Povyšte překladatele na roli vývojáře *(UC14)*
11. Nakonec uživatele test2@example.com odstraňte *(UC15)*
12. Importujte do projektu fráze v angličtině ze souboru strings.xml. Jsou ve formátu „Android (.xml)“. Přepište přitom existující překlady. *(UC23)*
13. Přejmenujte název fráze `button_accept` na `btn_accept` *(UC22)*
14. Odstraňte frázi `button_cancel` *(UC22)*
15. Exportujte překlady v angličtině ve formátu „Apple strings (.strings)“ *(UC24)*
16. Otestujte správu API klíče, nakonec ho smažte. *(UC10, UC11, UC12)*
17. Přejmenujte projekt na „Ukončuji testování“ *(UC16)*
18. Nakonec projekt smažte *(UC17)*
19. Smažte svůj účet

8.3.3 Tester 1 - překladatel

Tester přistupoval k aplikaci pomocí svého notebooku s prohlížečem Safari. Uživatelské rozhraní mu přišlo intuitivní a neměl problém se v něm orientovat.

Pro vyhledání fráze v kroku sedm tester použil vyhledávací menu prohlížeče a mezitím se ručně přesouval mezi stránkami. Později ale objevil i vyhledávací pole aplikace, které vyhledání značně urychlilo.

Další zpětnou vazbu mi dal při ukládání překladů. Snažil se použít klávesu Enter, ta ale místo uložení překladu přidávala nový řádek. Proto navrhnul, že by bylo vhodné přidat tlačítko na uložení. Nakonec ale sám zjistil, že pro potvrzení stačí překliknout ze vstupního pole.

Na závěr, při změně jména svého účtu, skoro kliknul na tlačítko smazat, místo na tlačítko uložit.

8.3.4 Tester 2 - překladatel

Testování probíhalo na notebooku testera v prohlížeči Firefox. Při vyhledávání v kroku sedm si nevšiml vyhledávacího pole a frázi hledal v seznamu ručně. Po mém upozornění ale vyhledávací pole nakonec použil a frázi rychle našel. Dále při změně jména si všiml, že v horní liště zůstalo jméno staré, které pro správné zobrazení potřebovalo obnovení stránky.

Při závěrečném vyjádření uvedl, že ačkoliv používal podobnou platformu poprvé, přišla mu vcelku přehledná.

8.3.5 Tester 3 - administrátor

Tester použil svůj notebook a k aplikaci přistupoval pomocí prohlížeče Safari. Po registraci si nevšiml informativní hlášky o nutnosti potvrzení emailu a zkoušel rovnou přejít k přihlášení. Po mém upozornění si pomocí mobilního telefonu emailovou adresu ověřil a do aplikace se již přihlásil úspěšně.

V kroku pět se přesunul ze záložky frází na záložku překladů a frázi přeložil tam. To je sice také možné, nicméně fráze lze překládat i po jejich vytvoření bez nutnosti přepnutí záložky.

Nejvíce zmatený byl avšak při importování a exportování. V obou případech mu trvalo dlouhou chvíli než pochopil, jak zvolit jazyk.

Po dokončení scénáře ale poskytl pozitivní zpětnou vazbu a vysvětlil své zaváhání prvotním setkáním s rozhraním. Bylo podle něj nicméně přehledné.

8.3.6 Tester 4 - administrátor

Tester dostal za úkol použít svoje mobilní zařízení a tak aplikaci otevřel v prohlížeči Google Chrome na telefonu Pixel 7a s operačním systémem Android.

Po registraci si ihned všiml informace o odeslání emailu, ta byla avšak moc dlouhá a na obrazovce se neukázala celá. Její smysl ale pochopil a pokračoval do své emailové aplikace. Při kliknutí na odkaz čekal automatické přesměrování do přihlášení, to se ale nestalo.

V pátém kroku při překládání fráze poznamenal, že by čekal zobrazení ukládacího tlačítka. Díky předchozím zkušenostem se službou PoEditor[1], ale neměl s uložením problém. Při pozvání překladatele v kroku sedm byl zmaten zobrazováním horních záložek i ve vnitřním menu seznamu členů. Tester si během vykonávání kroku deset všiml chyby, že dialog s novou volbou role neukazuje ve výchozím stavu aktuální roli uživatele. Funkci importování potřebnou v kroku dvanáct hledal nejdříve v záložce s frázemi, ale rychle pochopil, že se nachází na kartě možností ve vedlejší záložce. Jako předchozí tester – administrátor měl problém se zvolením jazyka a zkoušel klikat na tlačítko „Import“ bez jeho vybrání. Poslední drobnou chybou, objevenou díky použití mobilního zařízení, bylo nezjevné zobrazení výsledku importu. Ten se totiž ukázal mimo zobrazovaný obsah a nebylo na něj nijak upozorněno.

Tester dostal prostor pro závěrečné vyjádření. Uvedl, že se mu s aplikací pracovalo dobře a její vzhled působil přívětivě.

8.3.7 Shrnutí

Uživatelské testování potvrdilo správný návrh uživatelského rozhraní. Na zúčastněné působilo intuitivně a bez větších problémů se v něm orientovali. Podařilo se otestovat funkčnost aplikace na množství odlišných prohlížečů a velikostí obrazovek, čímž se naplnil první nefunkční požadavek *dostupnost webového rozhraní*.

Posbírané podněty jsem následně zapracoval do aplikace a drobné chyby opravil. Jednalo se například o:

- Úprava vzhledu zvolení jazyků v obrazovce pro importování a exportování.
- Automatické posunutí na výsledek importu.
- Vylepšení zobrazení hlášky o nutnosti ověření emailové adresy.
- Zobrazení aktuální role člena v dialog sloužící pro její změnu.
- Oprava aktualizace jména v liště při jeho změně.

Automatické přesměrování do aplikace při ověřování emailové adresy, navržené čtvrtým testerem jsem nepřidával. Vylepšení tohoto procesu by vyžadovalo přidání dalšího endpointu na server a vlastní implementaci ověřování. Je to tak podnět pro budoucí vylepšení platformy, jelikož aktuální řešení je funkční.

Po úpravě výše zmíněných nedostatků jsem se rozhodl provést poslední test s novým uživatelem pro ověření zlepšení chování.

8.3.8 Tester 5 - administrátor

Tester použil svůj mobilní telefon iPhone 15 s webovým prohlížečem Safari. Po registraci si všiml informace o nutnosti ověření emailu a pokračoval ve své emailové aplikaci. Pokládám tedy opravu tohoto nedostatku za úspěšnou.

V kroku pět se opět jako předchozí testeři překlikl do záložky s překlady a nevyužil možnost editace překladů na stránce s frázemi. Ačkoliv se toto stalo i třetímu testerovi, nevnímám to jako veliký nedostatek a jde spíše o zaváhání při prvním setkání s rozhraním.

Za zmínku stojí zaváhání v kroku dvanáct kdy se funkcionalitu importu snažil hledat v záložce s frázemi i překlady.

[1] Jedno z analyzovaných konkurenčních řešení, viz sekce 3.4.

Nakonec jedinou výtkou bylo zobrazení názvu projektu. To není v žádné záložce vidět a bylo tak po ukončení testování přidáno do titulku stránky. Jinak testerovi připadal design rozhraní přívětivý a neměl problém se v něm orientovat. Považuji tak vytyčené chyby z minulých testů za opravené.

Kapitola 9
Nasazení do produkce

Po dokončení implementace jsem přistoupil k nasazení celé platformy do produkčního prostředí. Kapitola pojednává o nasazení frontendu i backendu s využitím jednoduchých, ale efektivních služeb.

9.1 Frontend

Webové rozhraní je poskytováno jako statická stránka, veškerý kód je spouštěn ve webovém prohlížeči na zařízení uživatele a není tak potřeba výkonný server pro zpracování požadavků. V současné době existují služby, které umožní nasazení statických stránek velice snadno. Navíc poskytují výhody jako například automatickou správu SSL certifikátů.

9.1.1 GitHub Pages

Jedna z těchto služeb poskytovaná přímo platformou GitHub umožňuje publikování webové stránky z repozitáře. Ten je nutné vytvořit nový, určený jen pro uložení obsahu webové stránky a byl by tak oddělený od repozitáře, kde je uchován kód platformy. Hlavní výhodu vidím v tom, že všechen obsah je verzován pomocí nástroje git a lze se tak jednoduše vrátit k předchozí verzi. Pro nasazení této platformy to ale není klíčové, jelikož obsah stránky je generován ze zdrojového kódu, který už je verzován v původním repozitáři.

Samozřejmostí je možnost přidat vlastní doménu a vynutit uživatele používat jen zabezpečený provoz přes protokol HTTPS.

9.1.2 Firebase Hosting

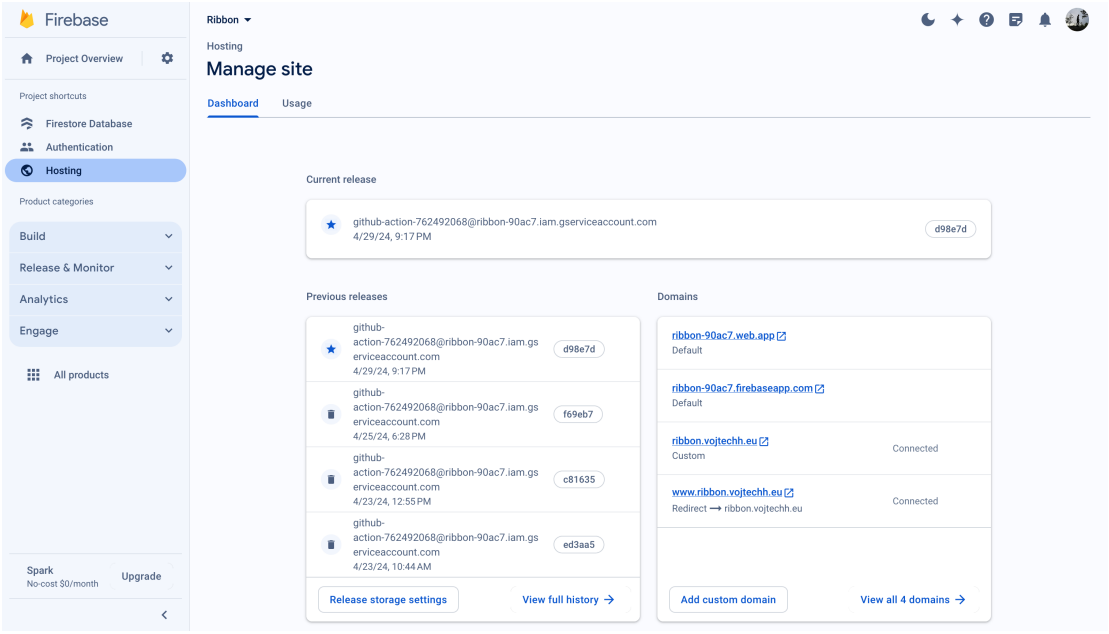
Další ze služeb je integrována do platformy Firebase, její využití si o to tedy říká. Nahrání obsahu je jednoduché, stačí použít nástroj „Firebase CLI“, kterému nejdříve nastavíme jaký Firebase projekt má použít a pomocí příkazu `firebase deploy` obsah stránky nahrajeme a zveřejníme.

Ve webovém rozhraní projektu pak máme možnost přidat vlastní doménu a dokonce vidět předchozí verze webu, jež byly nahrány dříve. Nastavení zabezpečené komunikace pomocí HTTPS není třeba, služba nešifrovanou komunikaci přes HTTP ani nepodporuje a není tak nutné starat se o jakékoliv nastavení.

Nakonec jsem zvolil tuto službu, díky její velice snadné použitelnosti a faktu, že je pak možno další část platformy spravovat přímo ve Firebase. Po nasazení je frontend dostupný na adrese `https://ribbon.vojtechh.eu`.

9.2 Backend

Jelikož backend je ve svém jádru pouze aplikace běžící na JVM, její nasazení nebude tak snadné. Je potřeba výkonný server, schopný neustále obsluhovat dotazy od klientů.



Obrázek 9.1. Frontend aplikace nasazený ve službě Firebase Hosting.

9.2.1 Heroku

Jedna ze služeb, která prostor pro nasazení poskytuje je Heroku. Pomocí kontejnerů *Dynos* lze spustit instanci libovolné aplikace. Je možné přímo nahrát spustitelný soubor, nebo zvolit propojení s GitHub repozitářem a nastavit automatické nasazení po nahrání nových změn.

Bohužel, Heroku neposkytuje bezplatný režim ani na zkušební dobu a při nejlevnějším tarifu za 5 € měsíčně se kontejnery uspávají po 30 minutách nečinnosti.

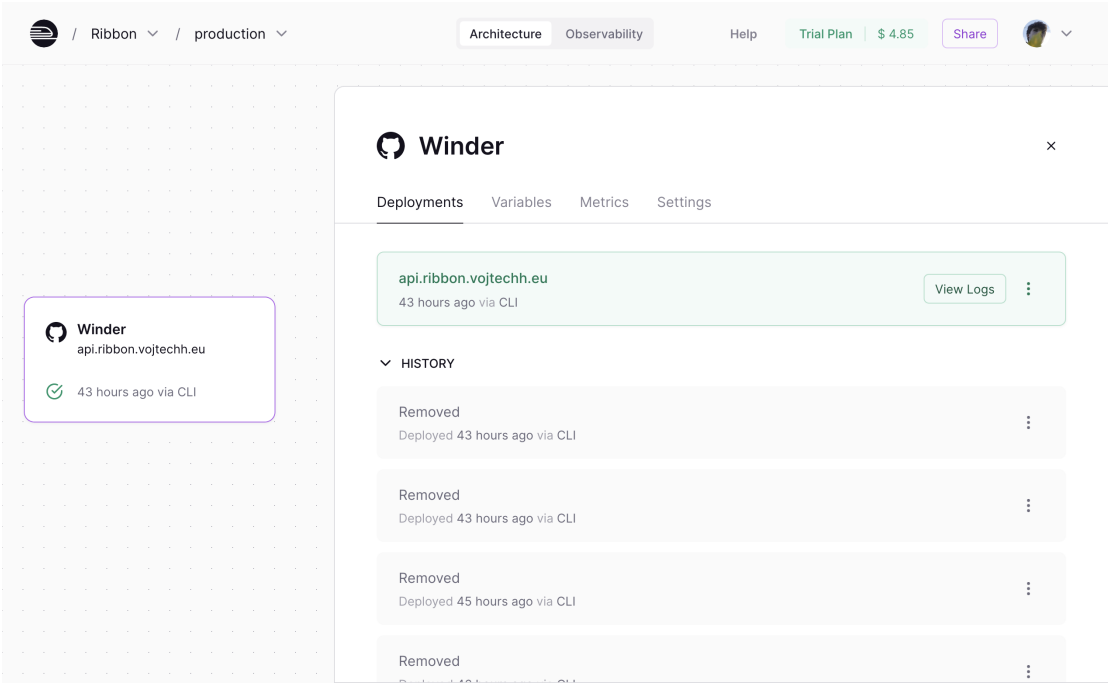
9.2.2 Railway

Obdobnou službu Railway lze vyzkoušet bezplatně, po registraci obdrží uživatel 5 $ pro volné využití a je tak vhodným adeptem pro nasazení vyvinuté platformy. Po vyčerpání kreditů jsou běžící instance nasazených aplikací pozastaveny a je nutné zvolit jeden z placených tarifů. Nejlevější z nich stojí 5 $ měsíčně s dalšími poplatky za použité zdroje (operační paměť, výkon).

Instance aplikací běží v Docker kontejnerech a při prvotním nastavení je třeba poskytnou příkaz na sestavení a spuštění aplikace. Až Railway kontejner nastaví a spustí, je aplikaci běžící vně poskytnuta proměnná `$PORT`. Hodnota této proměnné určuje port, na kterém se očekává, že bude probíhat komunikace vně kontejneru. Ve výchozím nastavení je pak ven z kontejneru vystavený pouze port `443`, komunikace tedy probíhá pomocí protokolu HTTPS.

Také Railway lze propojit z GitHub repozitářem a po nahrání nových změn se začne automaticky nasazovat nová verze aplikace.

Pro svojí jednoduchost a možnost bezplatného vyzkoušení jsem tak zvolil právě službu Railway a backend pomocí ní nasadil. Využil jsem i vlastní domény, kdy API je dostupné na adrese `https://api.ribbon.vojtechh.eu`.



Obrázek 9.2. Backend aplikace nasazený ve službě Railway.

9.3 GitHub Actions

Pro další usnadnění celého procesu nasazení jsem použil funkci nabízenou platformou GitHub „Actions“. Jedná se nástroj pro CI/CD, který umožňuje sestavit, otestovat a nasadit aplikaci přímo z repozitáře. Pro definici akcí, které chceme provést se používají tzv. pracovní postupy („workflows“). Ty se skládají z pracovních kroků („jobs“), které chceme vykonat a jsou popsány souborem ve formátu YAML.

Takové postupy jsem vytvořil tři, dva pro nasazení celé aplikace (backend a frontend) a jeden pro automatickou statickou analýzu celého projektu (viz sekce 8.2). Postupy jsou nastaveny na automatické spuštění po nahrání změn do větve `master`, nebo je lze spustit ručně přes webové rozhraní GitHubu. Před nasazením backendu se automaticky spustí jednotkové testy a jejich selhání celý proces nasazení zruší.

Definice pracovního postupu pro vydání backendu vypadá zhruba následovně:

Jak lze z uvedeného postupu vidět, přistupuje se k tajným proměnným v repozitáři, pomocí direktivu `secrets`. To je místo, kam je vhodné ukládat hesla či přístupové klíče potřebné v pracovních postupech. Nezkrácenou verzi a ostatní workflows lze najít v adresáři `.github/workflows/` v příloze se zdrojovým kódem.

9.4 Podpora běhu platformy

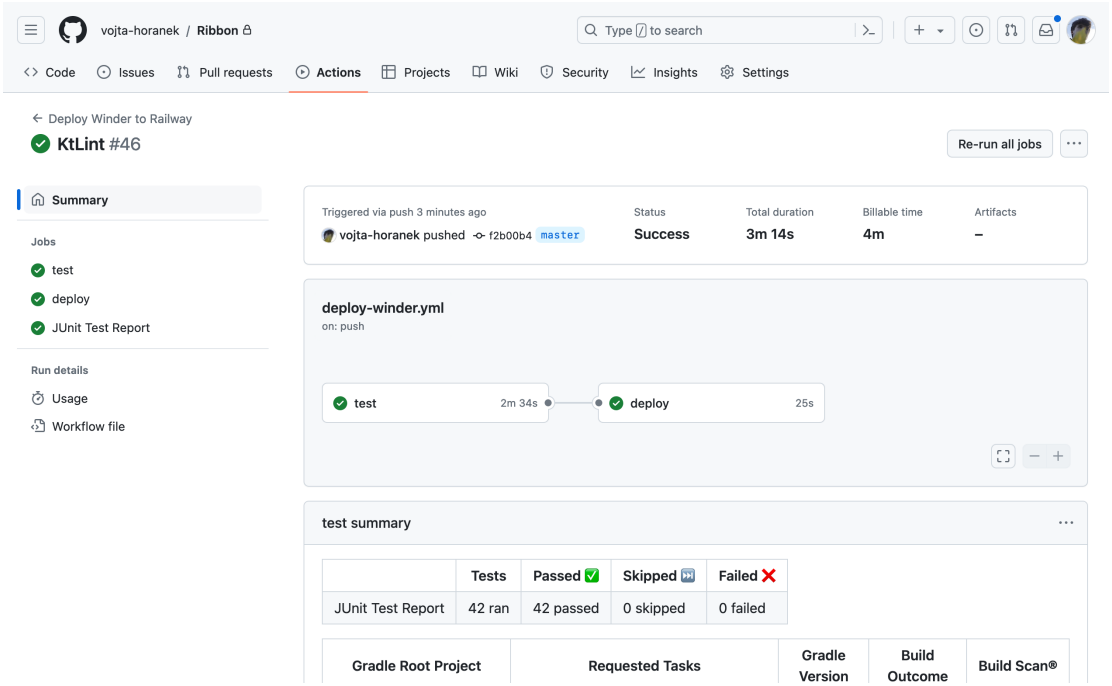
Jak bylo již zmíněno v sekci 7.5.6, pro podporu běhu platformy, konkrétně pro backend, jsem použil služby Sentry. Pracovní postup GitHub Actions po vydání vytvoří v Sentry nové novou verzi. Ta pomáhá při identifikaci chyb.

Sentry podporuje i oznámení chyb vývojářům pomocí emailu. Umožňuje tak velice rychlou reakci. Při vyřešení případné chyby je vhodné monitorovat, zda se objeví znovu, s novým označením verze. Pokud se tak stane, služba odešle emaily s varováním.

Při implementaci frontendu jsem nevyužil žádnou službu na sledování jeho stavu. V budoucnu je ale vhodné implementovat služby navržené v sekci 5.2.6.

```yaml
# Název pracovního postupu
name: Deploy Winder to Railway

on:
  push:
    branches: [ master ]    # Spuštění při změnách ve větvi master
  workflow_dispatch:        # Možnost manuálního spuštění

jobs:
  test:
    runs-on: ubuntu-latest  # Stroj, na němž postup poběží
    steps:
      # ...zjednodušeno...

      # Definice kroku pro spuštění jednotkových testů
      - name: Run tests
        run: >
        ./gradlew test
        --tests 'cz.cvut.fit.horanvoj.ribbon.winder*'

  deploy:
    needs:
      - test                # Job potřebuje úspěšně provedené testy
    runs-on: ubuntu-latest

    steps:
       # ...zjednodušeno...

      # Vytvoření nové verze v Sentry
      - name: Create Sentry release
        uses: getsentry/action-release@v1.7.0
        env:
          SENTRY_AUTH_TOKEN: ${{ secrets.SENTRY_AUTH_TOKEN }}
          SENTRY_ORG: ${{ secrets.SENTRY_ORG }}
          SENTRY_PROJECT: ${{ secrets.SENTRY_PROJECT }}
        with:
          environment: production

      # Oznámení službě Railway, že má spustit vydání nové verze
      - name: Deploy
        run: railway up --service Winder
        env:
          RAILWAY_TOKEN: ${{ secrets.RAILWAY_TOKEN }}
```

Výpis kódu 9.1. Pracovní postup nasazení backendu.



Obrázek 9.3. Zobrazení proběhlého workflow na platformě GitHub.

Kapitola 10
Závěr

Cílem bakalářské práce bylo navrhnout a implementovat prototyp platformy, která usnadní překlad při vývoji softwaru a speciálně mobilních aplikací. Hlavním přínosem měla být snadnost integrace do existujícího projektu.

V začátku práce jsem provedl rešerši existujících řešení, která ukázala na jejich společné funkcionality, ale zároveň identifikovala jejich nedostatky či omezení. Jednalo se primárně o mizivou využitelnost bezplatné varianty a jinak vysoké ceny, které rychle rostou s velikostí projektu. Dále také o design, jenž ve většině případů působil neintuitivně a mohl ztížit práci uživatelům s menší technickou zdatností, mezi něž často patří překladatelé.

Dále jsem se věnoval formování funkčních a nefunkčních požadavků nového řešení, definicí případů užití a aktérů. Tyto znalosti ujasnily postup pro následný návrh a implementaci. Po provedení analýzy dostupných technologií pro vývoj platformy byla zvolena technologie Kotlin Multiplatform. Tato volba umožnila sdílení doménových tříd mezi frontendem a backendem aplikace, čímž se zjednodušil vývoj a zkrátila doba potřebná pro implementaci. Pro tvorbu webového rozhraní byla zvolena knihovna Compose HTML s nadstavbou Kobweb a serverová část je vyvinuta ve frameworku Ktor.

Došlo k vypracování klikatelného drátěného modelu, zobrazující návrh webového rozhraní a zvolení vnitřní architekturu platformy. Architektura je založená na principu Clean Architecture a zdokumentovaná textově i pomocí diagramů.

Návrh jsem převedl do praxe a implementoval prototyp pokrývající vytyčené požadavky. Konkrétně od kompletní správy projektů, přes překlad jeho frází, až po exportování textů pomocí API. Ukázalo se, že analýza a návrh určili správný postup. Platforma je připravena pro snadné budoucí rozšíření.

Následně jsem popsal proces nasazení služby do produkčního prostředí. V serverové části jsem zprovoznil monitorování pádu a výkonu. Díky tomu je aplikace připravena pro použití v reálném projektu.

V závěru byla platforma otestována uživatelskými a jednotkovými testy, které prokázaly funkčnost a intuitivnost vytvořeného řešení.

Jako možnosti pro budoucí rozšíření platformy uvádím následující:

- Gradle/SPM Plugin – Vytvoření zásuvného modulu do nástroje řídící sestavení projektu mobilní aplikace. Umožňoval by stahovat nové překlady bez nutnosti přímého použití API.
- Vývoj nového frontendu – Díky technologii Kotlin Multiplatform by případný vývoj na další platformu znamenal pouze implementaci prezentační vrstvy a je tak nabíledni ho v budoucnu vyhotovit.
- Analytika – Vhodným rozšířením podpory chodu by bylo přidání služby Firebase Analytics do webového rozhraní.
- Historie překladů – Přínosem by mohlo být přidání historie překladů, pokud by se uživatel potřeboval vracet k předchozím verzím.

- Více informací k frázi – Vylepšením s přínosem pro snažší překládání může být možnost přidání komentářů a snímků obrazovky k frázím. K ukládání příloh lze navíc použít další službu Firebase pojmenovanou *Storage*.
- Více formátů – Pro rozšíření mezi více vývojářů by bylo vhodné přidat podporu pro další formáty exportu a importu, které jsou využívány například při vývoji webové aplikace či jiného softwaru.
- Možnosti přihlášení – Implementace přihlašování pomocí účtů Google a Facebook by rozšířila okruh potenciálních uživatelů. Použitá služba Firebase Authentication tyto metody podporuje a jde tak o snadné rozšíření.

Literatura

[1] MoldStud. *The importance of localization in software development* [online]. [vid. 2024-04-19]. Dostupné na https://moldstud.com/articles/p-the-importance-of-localization-in-software-development.

[2] Collins, Rosann Webb. Software Localization: Issues and Methods. *ECIS 2001 Proceedings* [online]. s. 9 [vid. 2024-04-19]. Dostupné na https://aisel.aisnet.org/ecis2001/4.

[3] TechTarget. *What is localization?* [online]. [vid. 2024-04-19]. Dostupné na https://www.techtarget.com/searchcio/definition/localization.

[4] Slashdot. *Best Localization Software of 2024* [online]. [vid. 2024-04-19]. Dostupné na https://slashdot.org/software/localization/.

[5] G2. *Top Free Software Localization Tools* [online]. [vid. 2024-04-19]. Dostupné na https://www.g2.com/categories/software-localization-tools/.

[6] crozdesk. *Localization Software* [online]. [vid. 2024-04-19]. Dostupné na https://crozdesk.com/it/localization-software.

[7] Crowdin. *Company Description* [online]. [vid. 2024-04-20]. Dostupné na https://support.crowdin.com/company-description/.

[8] G2. *Crowdin Reviews & Product Details* [online]. [vid. 2024-04-20]. Dostupné na https://www.g2.com/products/crowdin/reviews.

[9] Crowdin. *600+ apps and integrations to make your whole company multilingual* [online]. [vid. 2024-04-20]. Dostupné na https://store.crowdin.com/.

[10] Crowdin. *Plans for everyone* [online]. [vid. 2024-04-20]. Dostupné na https://crowdin.com/pricing.

[11] Phrase. *The Phrase Localization Platform* [online]. [vid. 2024-04-20]. Dostupné na https://phrase.com/platform/.

[12] Phrase. *Introducing the Phrase Localization Platform* [online]. [vid. 2024-04-20]. Dostupné na https://phrase.com/pricing/.

[13] Localazy. *Project roles* [online]. [vid. 2024-04-23]. Dostupné na https://localazy.com/docs/general/defining-user-roles.

[14] Localazy. *Localization for teams of any size* [online]. [vid. 2024-04-25]. Dostupné na https://localazy.com/pricing.

[15] PoEditor. *Pricing plans that scale with your business* [online]. [vid. 2024-04-25]. Dostupné na https://poeditor.com/pricing/.

[16] Lokalise. *Lokalise apps* [online]. [vid. 2024-04-25]. Dostupné na https://lokalise.com/product/apps.

[17] Lokalise. *Choose the plan that's right for your team* [online]. [vid. 2024-04-25]. Dostupné na https://lokalise.com/pricing.

[18] Google. *Material Design* [online]. [vid. 2024-04-23]. Dostupné na https://m3.material.io/.

[19] Spring. *Projects* [online]. [vid. 2024-05-11]. Dostupné na https://spring.io/projects.

[20] JetBrains. *Ktor: Build Asynchronous Servers and Clients in Kotlin* [online]. [vid. 2024-05-11]. Dostupné na https://ktor.io/.

[21] Koin. *The pragmatic Kotlin & Kotlin Multiplatform Dependency Injection framework* [online]. [vid. 2024-05-14]. Dostupné na https://insert-koin.io/.

[22] Google. *Firebase* [online]. [vid. 2024-05-14]. Dostupné na https://firebase.google.com/.

[23] Google. *Cloud Firestore* [online]. [vid. 2024-05-14]. Dostupné na https://firebase.google.com/docs/firestore.

[24] Google. *Firebase Authentication* [online]. [vid. 2024-05-14]. Dostupné na https://firebase.google.com/docs/auth.

[25] Sentry. *Welcome to Sentry Docs* [online]. [vid. 2024-05-14]. Dostupné na https://docs.sentry.io/.

[26] Meta. *React* [online]. [vid. 2024-05-14]. Dostupné na https://react.dev/.

[27] React Router. *Feature Overview* [online]. [vid. 2024-05-05]. Dostupné na https://reactrouter.com/en/main/start/overview.

[28] Aratani, Akira. *Kotlin Multiplatform Libraries* [online]. [vid. 2024-05-05]. Dostupné na https://github.com/AAkira/Kotlin-Multiplatform-Libraries.

[29] JetBrains. *Compose Multiplatform* [online]. [vid. 2024-05-05]. Dostupné na https://github.com/JetBrains/compose-multiplatform.

[30] Analytics, Google. *Automatically collected events* [online]. [vid. 2024-05-14]. Dostupné na https://support.google.com/analytics/answer/9234069.

[31] Phillips, Addison a Mark Davis. *Matching of Language Tags* [RFC 4647]. [vid. 2024-05-09]. Dostupné na DOI 10.17487/RFC4647. Dostupné na https://www.rfc-editor.org/info/rfc4647.

[32] Phillips, Addison a Mark Davis. *Tags for Identifying Languages* [RFC 5646]. [vid. 2024-05-09]. Dostupné na DOI 10.17487/RFC5646. Dostupné na https://www.rfc-editor.org/info/rfc5646.

[33] influxdata. *Database Denormalization* [online]. [vid. 2024-05-09]. Dostupné na https://www.influxdata.com/glossary/database-denormalization/.

[34] Shaji, Pooja. *Choosing Android Architectures: MVC, MVP, MVVM, Clean Architecture, and MVI* [online]. [vid. 2024-05-09]. Dostupné na https://medium.com/@KodeFlap/choosing-android-architectures-mvc-mvp-mvvm-clean-architecture-and-mvi-8ad2a43f7f9b.

[35] wanghaonanlpc. *MVC vs MVP vs MVVM* [online]. [vid. 2024-05-09]. Dostupné na https://medium.com/@wanghaonanlpc/mvc-vs-mvp-vs-mvvm-bfcf7568aac0.

[36] Dashwave. *Android Architecture Patterns — MVC, MVP, MVVM, MVI, Clean Architecture* [online]. [vid. 2024-05-09]. Dostupné na https://medium.com/droidblogs/android-architecture-patterns-mvc-mvp-mvvm-mvi-clean-architecture-cde8029b8f37.

[37] Martin, Robert C. *The Clean Architecture* [online]. [vid. 2024-05-09]. Dostupné na https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[38] IBM. *What is a REST API?* [online]. [vid. 2024-05-11]. Dostupné na https://www.ibm.com/topics/rest-apis.

[39] Figma. *Figma: The Collaborative Interface Design Tool* [online]. [vid. 2024-05-11]. Dostupné na https://www.figma.com/.

[40] JetBrains. *IntelliJ IDEA* [online]. [vid. 2024-05-12]. Dostupné na https://www.jetbrains.com/idea/.

[41] JetBrains. *JetBrains Fleet* [online]. [vid. 2024-05-12]. Dostupné na https://www.jetbrains.com/fleet/.

[42] Gradle. *Gradle Build Tool* [online]. [vid. 2024-05-12]. Dostupné na https://gradle.org/.

[43] Kotlin. *Code sharing between platforms* [online]. [vid. 2024-05-12]. Dostupné na https://kotlinlang.org/docs/multiplatform.html#code-sharing-between-platforms.

[44] JetBrains. *Kotlin multiplatform / multi-format reflectionless serialization* [online]. [vid. 2024-05-12]. Dostupné na https://github.com/Kotlin/kotlinx.serialization.

[45] Developers, Android. *String resources* [online]. [vid. 2024-05-14]. Dostupné na https://developer.android.com/guide/topics/resources/string-resource.

[46] Apple. *String Resources* [online]. [vid. 2024-05-14]. Dostupné na https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/LoadingResources/Strings/Strings.html.

[47] IEEE. *IEEE Std 1003.1-2017: POSIX.1a* [online]. [vid. 2024-05-14]. Dostupné na https://pubs.opengroup.org/onlinepubs/9699919799/.

[48] Apple. *String Format Specifiers* [online]. [vid. 2024-05-14]. Dostupné na https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Strings/Articles/formatSpecifiers.html.

[49] Vrieze, Paul de. *XmlUtil* [online]. [vid. 2024-05-14]. Dostupné na https://github.com/pdvrieze/xmlutil.

[50] Foundation, Material. *Material Theme Builder* [online]. [vid. 2024-05-14]. Dostupné na https://material-foundation.github.io/material-theme-builder/.

[51] Awesome, Font. *ribbon* [online]. [vid. 2024-05-14]. Dostupné na https://fontawesome.com/icons/ribbon?f=classic&s=solid.

[52] Awesome, Font. *License* [online]. [vid. 2024-05-14]. Dostupné na https://fontawesome.com/v4/license/.

[53] Yudov, Danil. *Libres – Resources generation in Kotlin Multiplatform* [online]. [vid. 2024-05-14]. Dostupné na https://github.com/Skeptick/libres/tree/master.

[54] Wolf, Russell. *Multiplatform Settings* [online]. [vid. 2024-05-14]. Dostupné na https://github.com/russhwolf/multiplatform-settings.

[55] JUnit 5. *The 5th major version of the programmer-friendly testing framework for Java and the JVM* [online]. [vid. 2024-05-03]. Dostupné na https://junit.org/junit5/.

[56] mockito. *Tasty mocking framework for unit tests in Java* [online]. [vid. 2024-05-11]. Dostupné na https://site.mockito.org/.

[57] MockK. *mocking library for Kotlin* [online]. [vid. 2024-05-11]. Dostupné na `https://mockk.io/`.

[58] Pinterest. *KtLint* [online]. [vid. 2024-05-01]. Dostupné na `https://github.com/pinterest/ktlint`.

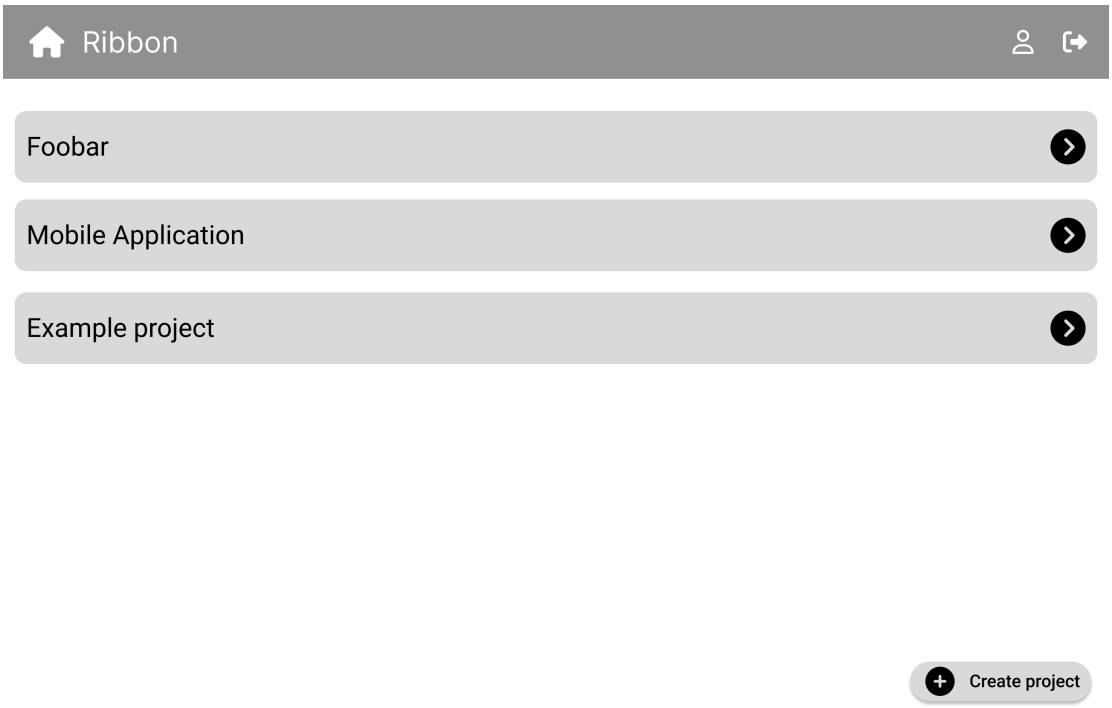
[59] EditorConfig. *What is EditorConfig?* [online]. [vid. 2024-05-01]. Dostupné na `https://editorconfig.org/`.

[60] JLLeitschuh. *Ktlint Gradle* [online]. [vid. 2024-05-01]. Dostupné na `https://github.com/JLLeitschuh/ktlint-gradle`.

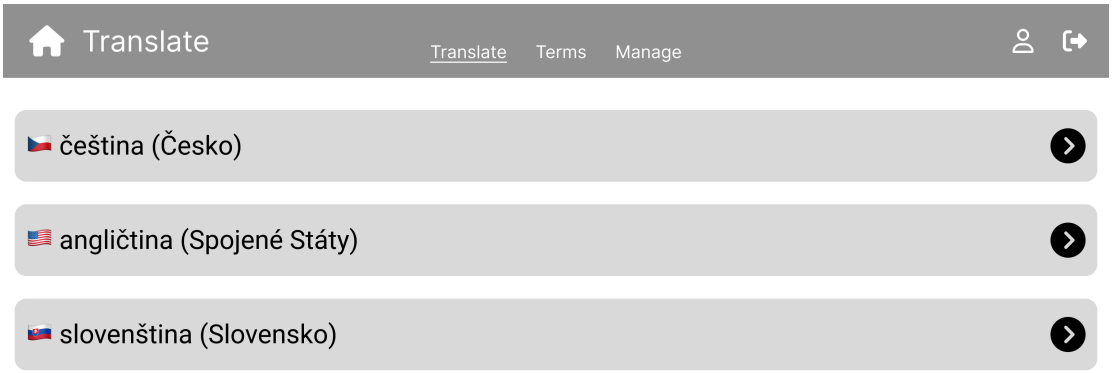
Příloha A
Seznam použitých zkratek

API	Application programming interface
CD	Continuous delivery
CI	Continuous integration
CLI	Command Line Interface
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DAO	Data access object
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated development environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
JWT	JSON Web Token
KMP	Kotlin Multiplatform
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
REST	Representational state transfer
SDK	Software development kit
SOAP	Simple Object Access Protocol
SPA	Single-page application
SPM	Swift Package Manager
SSL	Secure Sockets Layer
UI	User interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

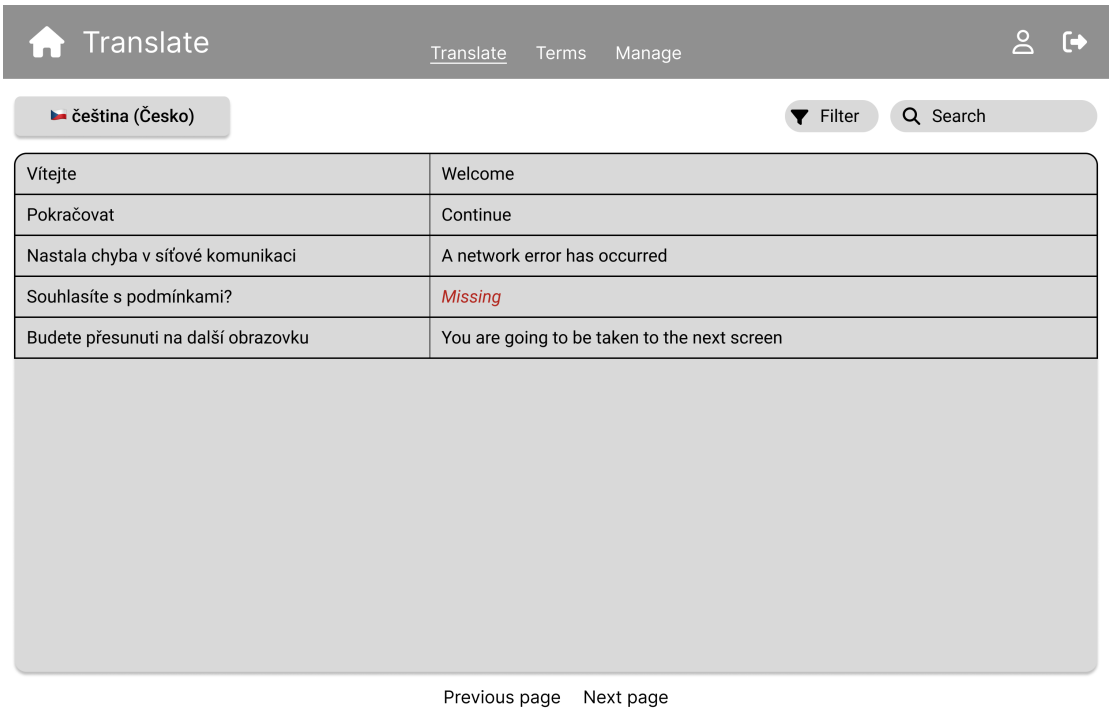
Příloha B
Drátěný model aplikace



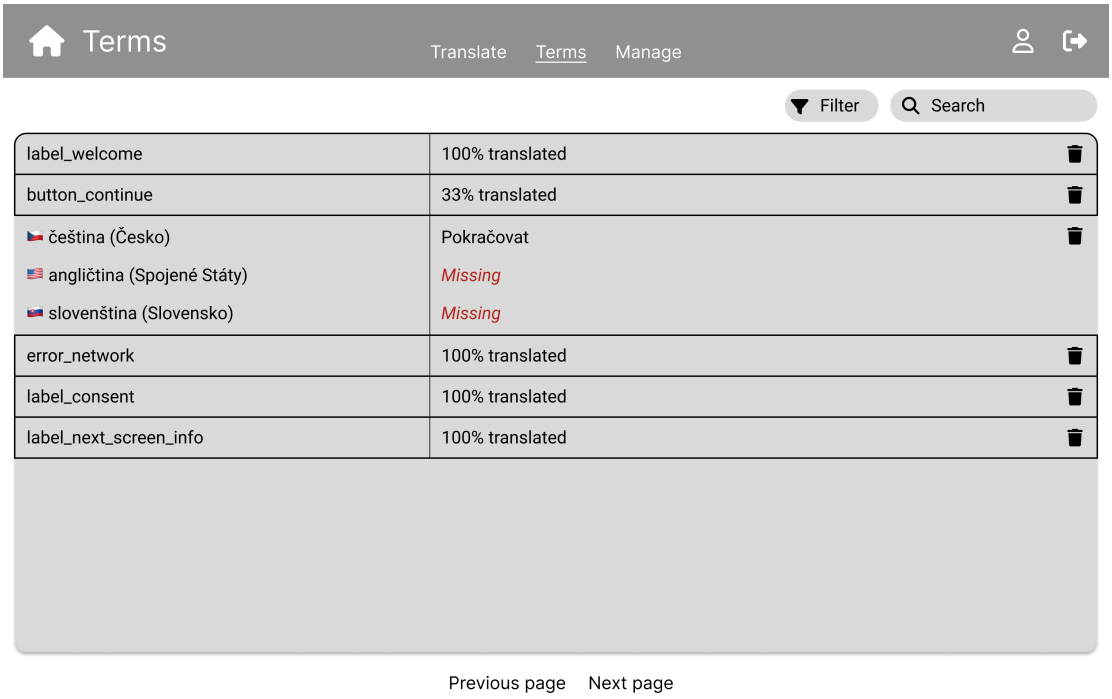
Obrázek B.1. Drátěný model stránky s projekty.



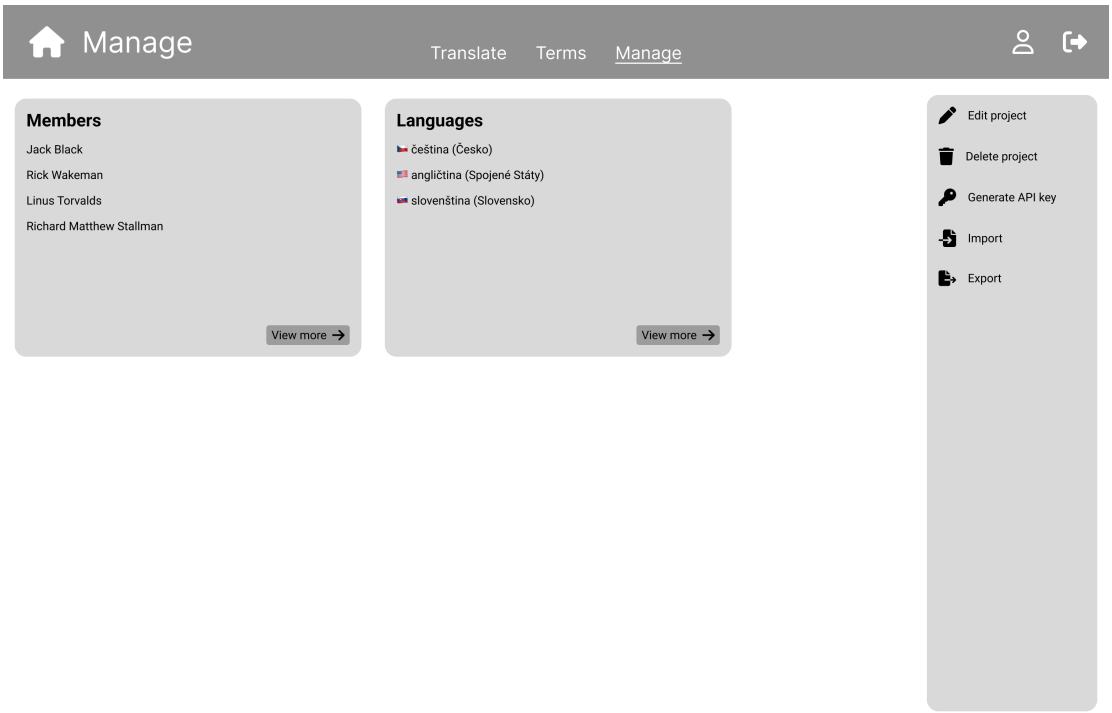
Obrázek B.2. Drátěný model zobrazování jazyků.



Obrázek B.3. Drátěný model překladů.



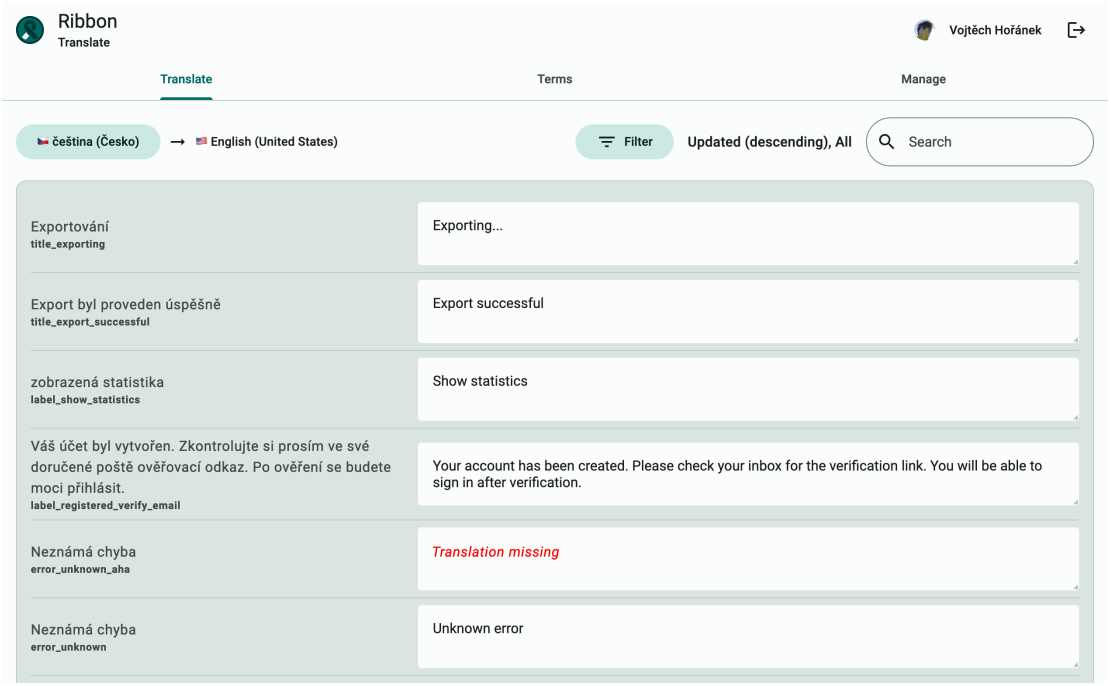
Obrázek B.4. Drátěný model správy frází.



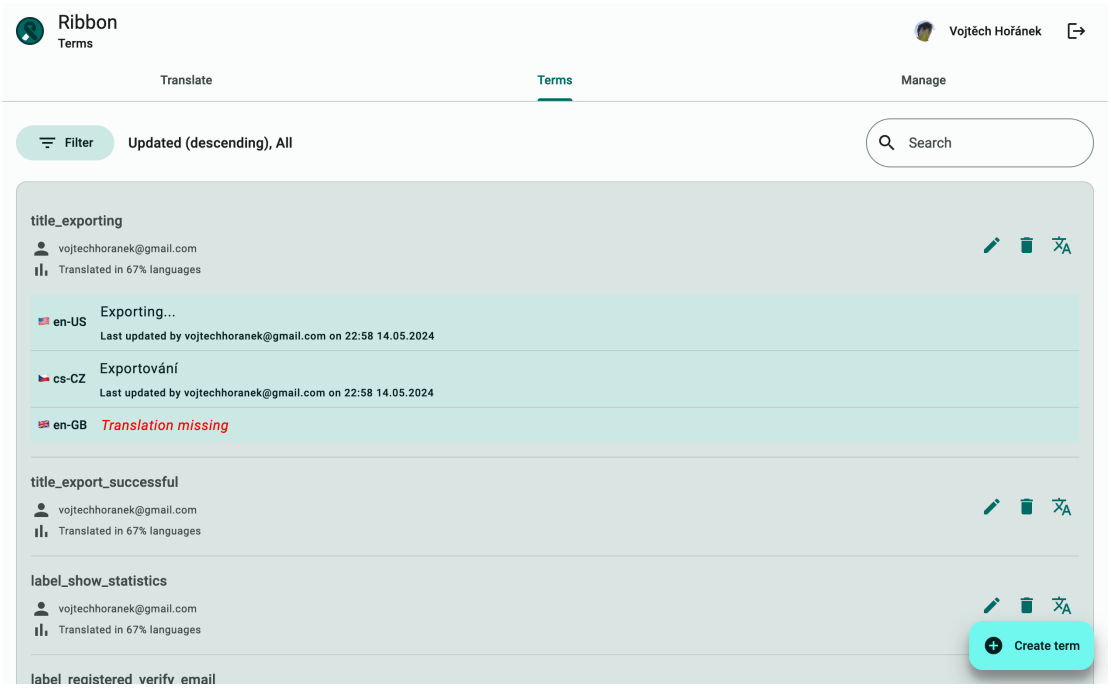
Obrázek B.5. Drátěný model nastavení projektu.

Příloha C
Webové rozhraní aplikace

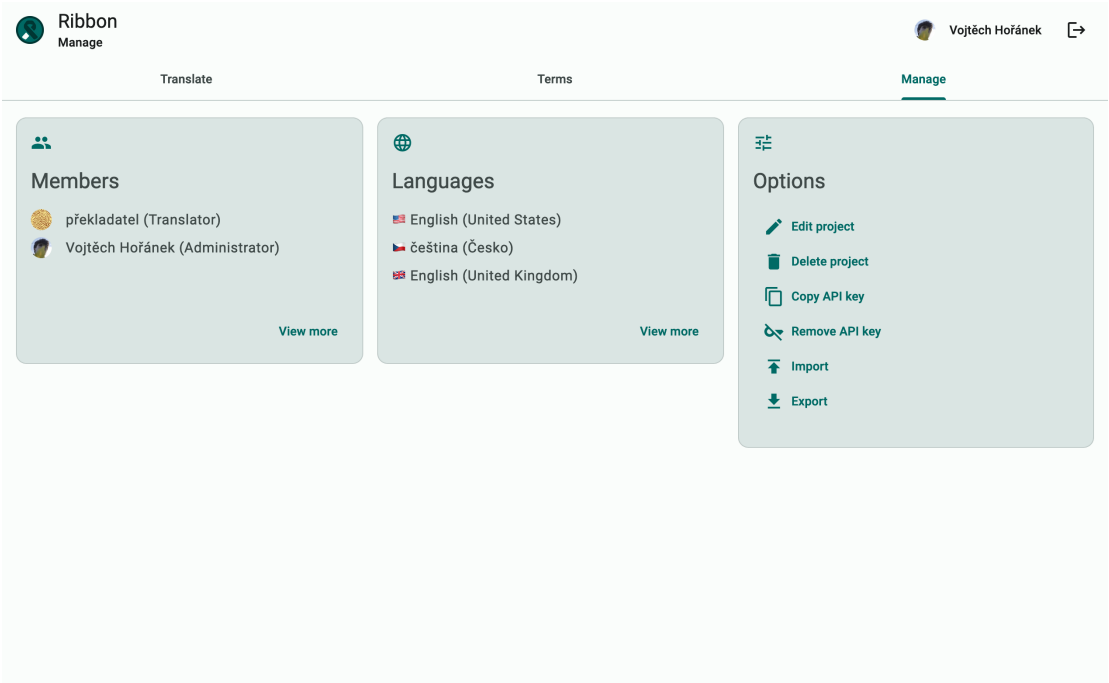
Obrázek C.6. Jazyky projektu v platformě Ribbon.



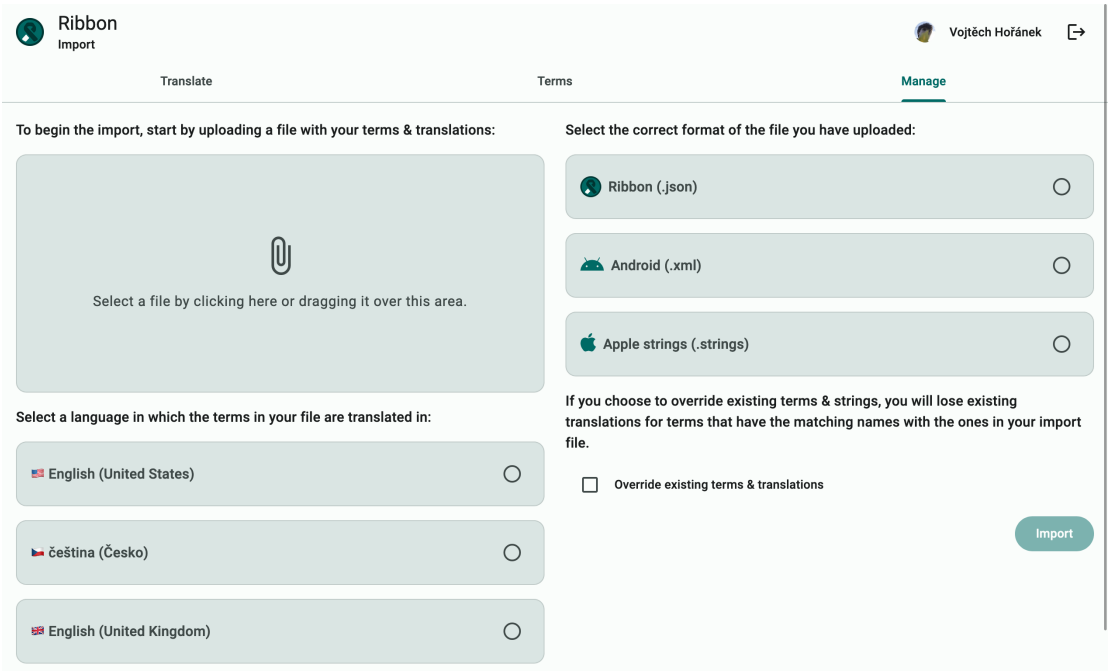
Obrázek C.7. Překlad frází pomocí platformy Ribbon.



Obrázek C.8. Správa frází pomocí platformy Ribbon.



Obrázek C.9. Nastavení projektu v platformě Ribbon.



Obrázek C.10. Import frází do projektu pomocí platformy Ribbon.

Příloha D
Obsah příloh

- `readme.txt` ..... stručný popis obsahu média
- `run.txt` ..... návod na spuštění implementace
- `wireframe` ..... drátěný model frontendu
  - `figma.fig` ..... klikatelný model ze softwaru Figma
  - `pic` ..... exportované snímky obrazovek z modelu
- `src`
  - `ribbon` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu OpTEX
- `showcase.mp4` ..... video ukázka práce s platformou
- `thesis.pdf` ..... text práce ve formátu PDF