FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Glosář
Student:	Olivie Abigail Franklová
Vedoucí:	Ing. David Bernhauer
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Nejen pro oblast návrhu software je důležité sestavit glosář (slovník pojmů), v mnoha různých odvětvích je potřeba zdokumentovat a definovat pojmy, aby nedocházelo ke zbytečným nedorozuměním.

Proveďte rešerši existujících aplikací pro zaznamenávání znalostí (knowledge) s důrazem na glosář.
Popište přístupy pro poskytování dat na webu: SOAP, REST a GraphQL.
Na základně předchozích kroků proveďte softwarovou analýzu, návrh a vytvořte jednoduchou webovou aplikaci pro vytváření záznamů v glosáři a jejich prezentaci uživateli.
Zpřístupněte data v aplikaci jako webovou službu.
Demonstrujte využití takové služby v jiné webové, mobilní nebo desktopové aplikaci.
Využijte existující aplikace (př. PA1 e-learning; zobrazování významů slov/frází "dynamická alokace"), nebo si vytvořte vlastní (př. návody pro mladé; zobrazování významů slov/frází "vyvářka").

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 19. listopadu 2021 v Praze.

Bakalářská práce

GLOSÁŘ

Olivie Abigail Franklová

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. David Bernhauer
12. května 2022

Obsah

Chtěla bych poděkovat především vedoucímu svojí práce Ing. Davidu Bernhauerovi. Dále svojí rodině a přátelům, bez kterých by bylo moje studium obtížné.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla 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 12. května 2022 .....

Abstrakt

Práce se zaměřuje na návrh a implementaci softwaru pro udržování a sdílení informací a znalostí. Literární rešerše se věnuje analýze podobných aplikací a dále popisuje rozdíly mezi GraphQL, SOAP a REST.

V práci byl vytvořen návrh a aplikace, která pomůže shromažďovat, provazovat a uchovávat znalosti. Aplikace poskytuje API a webové rozhraní, přes které lze manipulovat s glosářem. Implementované řešení podporuje vytváření záznamů a vyhledávání v nich a případně v celém textu.

Na práci lze navázat vytvořením přídavných aplikací, využívajících tento software; například rozšíření, které vyznačuje klíčová slova v textu a k nim zobrazuje jejich významy. Toto rozšíření bylo implementováno jako funkce, která může být vložena do webové stránky.

Klíčová slova softwarové inženýrství, znalostní management, sdílení know-how, open-source glosář, API, SOAP, REST, GraphQL

Abstract

This work focuses on software for maintenance and sharing knowledge and information. Theoretical part is dedicated to analyse similar software and describe the difference between GraphQL, SOAP and REST.

Design of application was created together with whole application, which helps centralizing, linking and keeping knowledge. Application provides API and web interface to manipulate with glossary. Created solution supports creating records and searching in already created records. The solution also supports searching for records in text.

New extensions can be created and easily added to the solution described above. For example extensions that will highlight records from glossary in text using colors and adding their definition to text. This extension was also implemented as a function for web source codes.

Keywords software engineering, knowledge management, know how sharing, open-source glossary, API, SOAP, REST, GraphQL

Slovník pojmů a zkratek

API Aplication control interface. Představuje spojení mezi počítačovými programy.

Beta Testovací, či neúplná, verze softwaru ve kterém se mohou objevovat chyby.

cloud Software běžící na internetu a ne na lokálním počítači.

CRUD Create Read Update Delete, operace pro práci s daty.

element Dvojice tagů, mezi nimiž je nějaký text.

framework Software, který slouží jako podpora při programování.

git Verzovací systém.

glosa Záznam v glosáři.

GraphQL Dotazovací jazyk pro API.

HTML Hypertext Transfer Protocol.

HTTP Hypertext Transfer Protocol, protokol pro přenos dat na webu.

HTTPS Hypertext Transfer Protocol Secure, protokol pro přenos dat na webu.

JSON JavaScript Object Notation, datový formát.

koncept Zastřešení glosy, propojuje jazyk, definici a pojem.

lifecykle vývoje Životní cyklus vývoje popisuje fáze v jakých je prováděn vývoj.

Markdown Jazyk pro tvoření formátovaného textu.

plugin Přidává další funkcionalitu do programu.

request Metoda pro komunikaci mezi programy. Nejčastěji se jedná o požadavek daný jedním programem druhému.

REST Representational state transfer.

RPC Remote Procedure Call – vzdálené volání procedur je protokol, který zajišťuje komunikaci mezi dvěma programy přes internetovou síť. Využívá k tomu transportní protokoly jako je TCP/IP nebo UDP.

SOAP Simple Object Access Protocol, protokol pro přenos dat na webu.

URL Uniform Resource Locator, webová adresa.

wiki Jedná se o několik webových stránek popisujících projekt nebo problematiku s ním spojenou, navzájem na sebe odkazujících.

Workflow Sekvence úkolů, které dohromady dávají ucelenou činnost. Obvykle se využívá při řízení projektů.

Workspace Pracovní prostor na tvoření poznámek, zapisování nápadů, tvoření záznamů ze schůzek nebo vytvoření to-do listů.

WWW World wide web.

XML eXtensible Markup Language, značkovací jazyk.

Úvod

Vytvoření glosáře neboli slovníku pojmů je nedílnou součástí každého projektu. Glosář je využíván při vývoji nového softwaru, ale také pro vysvětlení významu slov nebo pro upřesnění významů napříč obory. Práce se zaměřuje na vytvoření aplikace, která pomůže vytvořit a spravovat glosář.

Cíl práce

Teoretická část práce porovná a analyzuje existující aplikace pro zaznamenávání znalostí s důrazem na glosář. Dále budou probrány rozdíly v přístupech poskytování dat na webu, přičemž se práce zaměří pouze na webové služby SOAP, REST a GraphQL. Budou probrána a porovnána možná řešení problematiky skloňování, nespisovných výrazů a slov v množném čísle pomocí normalizace slov.

V praktické části práce bude provedena softwarová analýza, návrh a následná implementace glosáře. Glosář bude implementován jako open-source. Pro glosář bude vytvořena webová aplikace, která umožní uživateli přidávání glos a následné jejich zobrazovaní. Dále bude vytvořeno GraphQL API umožňující provádět různé operace s glosářem. Pro demonstraci využití glosáře bude implementována funkce do webové stránky, která na webové stránce vyznačí glosy z glosáře spolu s jejich významy.

Motivace

Aplikace bude sloužit jako nástroj k ukládání a zprostředkování pojmů a informací. Výsledek práce bude prospěšný například při vytváření nových projektů. Glosář bude podporovat vývoj tím, že zajistí přehlednost pojmů. Umožní všechny pojmy ukládat na jedno místo a usnadní jejich vyhledávání. Na projektech se často využívá mnoho míst pro ukládání informací a kvůli velkému počtu zdrojů může docházet k nepřehlednosti a těžkopádnému způsobu vyhledávání. Jednotlivé pojmy není snadné propojovat mezi sebou. Také často neexistuje snadný způsob, jak na daný pojem a jeho význam odkazovat v jiném textu. Tento problém by měla aplikace řešit. Převedením pojmů do glosáře, kde budou na jednom místě, se urychlí předávání znalostí, ať už mezi stálými kolegy, nebo nováčky v týmu.

Plný potenciál aplikace bude v její rozšiřitelnosti dalšími podpůrnými aplikacemi využívajícími API glosáře. Například bude možné na webové stránce vyznačit slova uložená v glosáři a uvést u nich jejich význam. Tato funkcionalita by se dala využít například při čtení odborných článků z oboru, který není čtenáři blízký, ale je pro něj nezbytné problematiku rychle pochopit.

Struktura práce

První kapitola se zaměří na různé aplikace, které alespoň z části nabízejí funkcionalitu glosáře. Druhá kapitola představí technologie, které se využívají při poskytování dat na webu. Následně popíše poskytování dat na webu skrze SOAP, REST a GraphQL a porovná tyto tři přístupy. Třetí kapitola se bude zabývat analýzou, tedy specifikováním požadavků, jakým způsobem bude glosa uložena, analýzou API a webového frontendu.

Čtvrtá kapitola se zaměří na návrh aplikace, návrh uložení glosy, API, vyhledávání glos, webového frontendu a výběr jazyka, ve kterém bude práce implementována. Pátá kapitola se zaměří na implementaci a testování. Popíše technologie použité při vývoji, strukturu projektu, dokumentaci a postup nasazení aplikace. Pátá kapitola také poskytne náhled na možná rozšíření aplikace.

Kapitola 1
Rešerše glosáře

V této kapitole je popsán pojem glosář a jsou zde předvedeny dva pohledy, jak na využití glosáře nahlížet. Dále se kapitola zabývá aktuálním řešením problematiky glosáře, uchovávání dat a know-how. V závěru jsou porovnány vybrané nástroje.

Glosář je nejčastěji používán jako seznam pojmů a definic. Objevuje se na konci odborné práce, nebo jako součást dokumentace při vývoji softwaru. Dále se může používat jako datový slovník [1, p. 83]. Datový slovník udržuje informace o popisu dat, jako je například jejich význam nebo vztah k ostatním datům [2], jinými slovy datový slovník udržuje metadata [1, p. 99].

Glosář může zabránit nedorozumění při používání technických či specifických termínů. Může se totiž stát, že každý vnímá význam termínu jiným způsobem a přesná definice zapsaná v glosáři tento problém eliminuje. Zaznamenávat můžeme technické pojmy, nejasné pojmy, požadavky a nebo již zmíněná metadata. Pojmy mohou být specifické pouze pro uzavřenou skupinu, ve které je glosář používán. Ve většině případů není nutné a ani žádané popisovat všechny významy daného pojmu. Zaznamenáváme tedy jen termíny nějakým způsobem užitečné. Může se jednat o nejasné či nejednoznačné pojmy, ale také o dovysvětlení určitých pojmů. Vysvětlení pojmu se nemusí skládat pouze z definice, ale také z dovysvětlujících faktů a nebo seznamu synonymních slov.[1, kapitola 7]

V této práci je glosář považován za nástroj pro uchovávání informací. Glosa odpovídá dvojici pojem a jeho význam, jako je tomu ve slovníku. Pojem je popsán jednoslovným či víceslovným názvem. Významem pojmu je myšleno vysvětlení problematiky textovou formou, formou obrázků, diagramem, nebo například videem, případně kombinací různých forem. Glosa je tedy pro tuto práci záznam v glosáři.

1.1 Glosář pro softwarový vývoj

V softwarovém vývoji je vhodné vést glosář a to nejlépe od úplného začátku projektu [1, p. 99]. Je vhodné zde ukládat pouze data naprosto nutná a není žádané vysvětlovat pojmy, které jsou zřejmé.[1, p. 122]

Jaké pojmy jsou zřejmé, lze určit především z toho, kdo bude glosář využívat. Pokud jsou adresáti ze stejné skupiny a mají podobné znalosti, stačí zaznamenávat pouze pojmy nejasné a glosář může mít menší záběr. Pokud je glosář tvořen k tomu, aby se domluvily dvě různé skupiny s různými znalostmi (manager a vývojáři, vývojáři a zadavatel, zadavatel a zákazník, ...), musí se glosář zaměřit na slova nejen nejasná v jedné skupině, ale i na pojmy, které by byly užitečné, nebo nezbytné pro druhou skupinu. Může se tedy stát, že některé glosy budou pro jednu skupinu naprosto zřejmé, ale i přesto by se měly do glosáře zapsat.

Příkladem může být skupina vývojářů, která vyvíjí program simulující chemické reakce. Pro vývojáře je dobré znát, které sloučeniny jsou kyseliny a které jsou zásady, aby mohli program náležitě otestovat. Pro chemiky se naopak mohou hodit pojmy technického rázu, aby mohli software používat a správně specifikovat, co od programu potřebují. Glosář tedy bude obsahovat pojmy neznámé pro alespoň jednu skupinu, požadavky na software a pojmy nezbytné pro vývoj.

Je zřejmé, že některé glosy jsou pro jistou skupinu zbytečné a proto by bylo vhodné pojmy oddělit. Jedním ze způsobů, jak tohoto můžeme dosáhnout, je řádné označení *tagem*, který nám řekne pro jakou skupinu je pojem důležitý. *Tag* by měl být jednoslovný pojem popisující okruh, kterým se glosa zabývá, nebo se může jednat o klíčová slova vztahující se ke glose.

Glosář usnadňuje softwarový vývoj tím, že udržuje pojmy na jednom místě. Zjednodušuje schopnost mít přehled nad vývojem pro všechny osoby, které se na něm podílejí, a to včetně zadavatele a vedení. Díky glosáři je snadnější zpětná kontrola požadavků a je zmenšeno riziko nedorozumění. Noví členové týmu se mohou rychleji zorientovat v pojmech a snadno nalézt význam pojmu, který neznají, nebo si nejsou jistí, jak je v kontextu s projektem používán.

1.2 Glosář pro běžné užití

Glosář, nebo častěji známý pojem slovník, je používán i v běžném životě. Pokud je nějaký pojem nejasný, člověk si musí jeho význam vyhledat. Občas může být vyhledání pojmu obtížné, jelikož je mnoho výrazů, které mají jiný význam v jiném kontextu, ale jsou popsány stejným slovem. Pokud by existoval slovník, který by řadil slova do kategorií podle tématu, nemuselo by docházet k špatnému pochopení pojmu.

Jestliže se osoba pokusí vyhledat slovo *switch* může být překvapena mnoha významy tohoto slova. Jako například: výraz používaný v basketbalu pro změnu obrany, film, hudební skupina, hra, vypínač, přepínač v konzoli, příkaz v programovacích jazycích, ... Jaký význam je správný, je potřeba zjistit z kontextu, který může být určen právě *tagem*.

Pro vyhledávání na internetu by bylo velmi užitečné, pokud by v sobě měla webová stránka zabudovaný slovník. Slovník by si webová stránka nemusela vytvářet sama, ale mohla by používat již hotový. Ve vybraném slovníku by si stránka určila, kterou tématikou se zabývá, a tedy které glosy mají být ukázány uživatelům. Využití slovníku by pak již bylo na samotné webové stránce. Například by mohla zobrazovat významy slov, které jsou uloženy v glosáři, po přejetí kurzorem myši přes daný pojem.

1.3 Existující řešení

V současné chvíli chybí aplikace, která by umožňovala tvoření vlastního glosáře. Aplikace by měla zajišťovat vkládání pojmů a jejich vysvětlení. Následně by je zprostředkovávala buď to pomocí webového rozhraní a nebo pomocí API. Existují dvě skupiny aplikací, přičemž každá z nich pokrývá jinou část problematiky.

Jednou skupinou podobných aplikací jsou takzvané *note-taking apps* neboli aplikace na tvoření poznámek. Mezi tyto aplikace patří Obsidian, Roam, Notion a OneNote, na které se tato kapitola zaměří. Existuje mnoho dalších *note-taking apps*, jako například Apple Notes či Google Keeps.

Další skupinou jsou *wiki* pro daný projekt. Jedná se o několik webových stránek popisujících projekt nebo problematiku s ním spojenou, navzájem na sebe odkazujících. Stránky jsou většinou zastřešeny jednou aplikací, jako je například Confluence, FogBuzz, Wiki.js, BookStack. Obecně je těchto aplikací mnoho a tato práce se zaměří pouze na Confluence a BookStack. Aplikace byly vybrány pouze dvě, jelikož jejich funkcionality jsou často velmi podobné nebo dokonce stejné.

Existují aplikace sloužící jako překladové slovníky, tyto slovníky neplní požadované funkcionality glosáře a proto se jimi nebude rešerše zabývat.

Aplikace sloužící jako výkladové slovníky sice heslovitě popisují významy slov, což odpovídá

základní funkcionalitě glosáře, ale na základě provedené rešerše aplikace pro jejich tvorbu neexistují nebo jsou proprietární. Proto se práce nebude zabývat ani výkladovými slovníky.

U porovnávaných aplikací se práce zaměří na jejich použitelnost, zda jsou přístupné skrze API, zda dokáží jednoduše odkazovat na jiné záznamy vytvořené v aplikaci. Další důležitou funkcionalitou je přiřazování *tagů* k záznamům. Porovnání se zaměří i na možnost vyhledávání v aplikaci. Neméně důležitou součástí bude také cena a možnost přizpůsobit si aplikaci vlastním potřebám.

1.3.1 Obsidian

Obsidian je aplikace pomáhající propojení myšlenek a poznámek. Používá lokální úložiště a je založená na textech v *Markdown* formátu [3]. Aplikace je zdarma pro osobní užití, zatímco pro komerční užití je placená. Synchronizace mezi více zařízení, na kterých je Obsidian používán, je možná přes placený *plugin*. Obecně má Obsidian rozsáhlou komunitu, přispívající možnými rozšířeními aplikace ve formě *pluginů*. K datům v aplikaci lze přistupovat skrze API. Pro uživatele aplikace je nutností znát *Markdown* formát, jak ale v *Markdownu* psát, je v aplikaci podrobně vysvětleno.

Je zde velmi dobře propracováno odkazování na jiné poznámky. Odkaz se píše do dvou hranatých závorek a aplikace našeptává poznámky, na které bychom mohli chtít odkazovat. U každé poznámky je možné zobrazit, jaké jiné poznámky na ní odkazují, případně v jakých jiných poznámkách je zmíněna bez odkazu, který je možné jednoduše přidat. Zmíněné citace poznámek nelze filtrovat, ale je možné v nich vyhledávat a řadit je. Stejně to platí i pro obyčejné vyhledávání v poznámkách. Aplikace dále nabízí možnost podívat se na propojení poznámek v grafu, který zobrazuje všechna spojení jednotlivé poznámky s ostatními.

Aplikace je vhodná hlavně pro osobní poznámky, protože využívá lokální úložiště. Díky rozsáhle komunitě a přístupu k datům skrze API má velký potenciál.

1.3.2 Roam

Na první pohled je Roam velmi podobný Obsidianu. Jedná se o *note-taking app*, která umožňuje jednoduché propojení poznámek. Poznámky v Roam jsou tvořené jako body a případně podbody v seznamu. Odkazy na jiné poznámky se píší do dvojitých hranatých závorek. Každá poznámka u sebe má seznam poznámek, které jsou s ní propojeny. Mezi poznámkami je možné filtrovat, vyhledávat a řadit je.

Roam je přístupný jako webová aplikace a poznámky jsou ukládány na serveru. Pokud je uživatel offline, poznámky se uchovávají v cache paměti. Stejně jako v aplikaci Obsidian je zde možnost vizualizace spojení poznámek přes graf. Roam je pouze placená služba. Přístup k datům bude v brzké době přístupný skrze API [4].

Jelikož je aplikace placená, nebyla dostatečná možnost aplikaci prakticky odzkoušet.

1.3.3 EverNote

EverNote je *note-taking app* vytvořená pro udržování pořádku, vytváření poznámek a plánování projektů. Umožňuje sdílení poznámek, vyhledávání poznámek, propojení s Google kalendářem a ukládání webových stránek [5].

Omezená základní verze je zcela zdarma, pro vytváření vlastních šablon poznámek, procházení poznámek offline a většího prostoru pro poznámky je třeba zakoupit předplatné Premium. Ještě větší balíček Business umožňuje spolupracovat ve sdílených prostorech, zobrazovat historii nebo centralizovat správu účtů.

EverNote má webovou, desktopovou a mobilní aplikaci. Funguje na všech nejpoužívanějších operačních systémech. Uživatel se po nainstalování nebo spuštění aplikace nejprve setká s tu-

toriálem, který je jednoduše přístupný i po jeho vyzkoušení. Uživatel může tvořit poznámky, které se dají třídit do sešitů. K poznámkám se dají připsat *tagy*, přičemž aplikace našeptává, jaký *tag* použít. Uživatel si může vytvořit již předpřipravenou poznámku například s tabulkou. Vytvořené poznámky jsou v *Markdown* formátu stejně jako v Obsidianu, uživatel ale *Markdown* znát nemusí, protože je zde přístupný formátovací panel.

Další funkcí EverNote je *web clipper* – po stažení rozšíření do prohlížeče je možné si uložit celou webovou stránku jako poznámku. V aplikaci je propracované vyhledávání mezi poznámkami, lze vyhledávat například i v obrázcích, podle *tagů* nebo slov v poznámce. Nelze zde však odkazovat na jiné poznámky vytvořené v EverNote. EverNote má také API, které umožňuje všechny operace jako v aplikaci.

Aplikace je vhodná pro vytváření poznámek, sdílení poznámek v rámci týmu, ukládání webových stránek a plánování úkolů.

1.3.4 Notion

Notion je webová *note-taking app* pro týmovou spolupráci. Je zde ucelený *Workspace*

pro spolupráci v týmu i pro osobní práci. V aplikaci se dají vytvářet databáze (tabulky) pojmů. Dále je zde snadné vytvářet *Workflow* v různých stylech pomocí předpřipravených šablon [6].

Notion má čtyři platební plány, základní zcela zdarma, verzi Pro, která umožňuje nahlížet na historii poznámek a další dva plány pro týmovou spolupráci se sdílenými přístupy a právy, případně s neomezenou historií úprav a pokročilým zabezpečením. Verze Pro je pro studenty zcela zdarma.

Notion má mnoho předpřipravených šablon, které jsou rozčleněny do skupin podle použití. Šablony je možné následně upravovat, ale není možné vytvářet nové. Jelikož je zde mnoho funkcí, aplikace se může zdát nepřehledná; to však jistým způsobem řeší pomocná složka, kde jsou připnutá YouTube videa s tutoriály jak s Notion pracovat. Vytvořené poznámky jsou v *Markdown* formátu a stejně jako v EverNote, uživatel nemusí *Markdown* znát, protože je zde přístupný formátovací panel.

Stejně jako EverNote, nabízí i Notion *web clipper* jako rozšíření do prohlížeče. Notion má také své API a to zatím pouze v *Beta verzi*. V aplikaci je možné se odkazovat na jinou stránku, nebo poznámku z aplikace pomocí odkazu. Pro získání odkazu na poznámku v Notion je potřeba kliknout přímo na soubor s poznámkou a vybrat kopírování odkazu, který poté můžeme přidat jako odkaz v jiné poznámce. Odkazování na jiné poznámky je zdlouhavé a neintuitivní.

V Notion není možné poznámky označit *tagem*, ale je zde slušně zpracované vyhledávání v poznámkách, přičemž je možné vyhledávat i podle základních filtrů jako: kdo poznámku vytvořil, hledání pouze v nadpisu, hledání pouze ve složce ...

Notion se hodí pro tvoření osobních poznámek i pro práci v týmu. Dají se zde zaznamenávat to-do listy, vytvářet *Workflow* nebo přiřazovat úkoly.

1.3.5 OneNote

OneNote je *note-taking app* vyvinutá Microsoftem jako aplikace na tvoření poznámek, umožňuje organizování poznámek a sdílení s ostatními lidmi [7].

OneNote se dá používat jako webová nebo jako desktopová aplikace. U webové aplikace na začátku déle trvá než se jednotlivé poznámky načtou, ale poté je proklikávání mezi nimi dostatečně responzivní. Desktopová aplikace lze stáhnout na macOS a MSWindows. Pro uživatele s Microsoft účtem je používání OneNote zdarma. Aplikace umožňuje vytvářet poznámky do jednotlivých textových souborů, tyto soubory mohou být rozděleny do složek, ale ve webové verzi nemohou být rozděleny do podsložek.

Je zde možné odkazovat na webové stránky nebo přímo na textové soubory v OneNote. Pro získání odkazu na poznámku v OneNote je potřeba kliknout přímo na soubor s poznámkou

a vybrat kopírování odkazu, který poté můžeme přidat jako odkaz v jiné poznámce. Odkazování na jiné poznámky je zdlouhavé a neintuitivní, stejně jako v Notion. K poznámkám lze přistupovat skrze API.

OneNote umožňuje základní textové úpravy a má předpřipravené značky jako je například zvýraznění, vykřičník, značka odkazu a jiné. Značky není možné upravovat ve webové aplikaci, ale pouze v desktopové. Poznámky lze sdílet s jednou nebo s více osobami nebo s nějakou určitou skupinou. Mezi poznámkami by se mělo dát vyhledávat, což bohužel není často dostatečně přesné.

Aplikace je vhodná, pokud celý tým pracuje s Microsoft Office technologiemi. Aplikace je spíše vhodná na tvoření poznámek, než přímo na tvoření dokumentace nebo slovníku pro projekt. V projektovém řízení by se dala využít pro tvoření záznamů ze schůzek, případně pro zaznamenávání myšlenek. Velkou nevýhodou je nekompatibilita některých funkcí (vytváření podsložek) v desktopové a ve webové verzi aplikace. Zároveň je desktopová aplikace pojata jinak než aplikace webová a je složité se zorientovat v druhé verzi.

1.3.6 Confluence

Confluence je aplikace pro tvoření dokumentů, sdílení know-how a znalostí v týmu. Je vytvořená společností Atlassian, která také provozuje ticketovací systém Jira nebo Bitbucket - software pro verzování založený na *gitu*. Základní verze je zdarma až pro desetičlenný tým, v placených verzích je možnost mít 22 000 až neomezeně členů, jsou zde navíc některé funkcionality a větší velikost úložiště. Confluence může být provozována jako *cloudová* služba nebo si ji uživatelská firma může provozovat sama [8].

Aplikace umožňuje vytvářet prostory a stránky. Každá stránka náleží do nějakého prostou. Stránky se dají vytvořit z předpřipravených šablon, nebo jako prázdné stránky. Každou stránku je možné označit jedním nebo více *tagy*. Je možné zde vytvářet rozsáhlé dokumentace, záznamy ze schůzek, vytvářet *Workflow* nebo přiřazovat úkoly. Stránky mohou být sdílené s celou společností, s jedním týmem, s určitými lidmi nebo mohou být čistě osobní. Operace se stránkami a čtení stránek je možné přes REST API.

Confluence je vcelku intuitivní a má propracovanou nápovědu, díky které je snadnější se v ni orientovat. Je možné propojit Confluence s dalšími aplikacemi od společnosti Atlasian a je tedy možné odkazovat například na úkoly vytvořené v ticketovacím systému Jira. Velmi snadno se zde vkládají odkazy na jinou stránku vytvořenou v Confluence a to přes seznam stránek, ve kterém je možné jednoduše vyhledávat. Obecně je v aplikaci velmi propracované vyhledávání, přičemž je možné filtrovat podle *tagů*, data úpravy, autora článku, prostoru atd.

1.3.7 BookStack

BookStack je open-source *self-hosted* [1] platforma pro tvoření *wiki* stránek. Pro používání *frameworku* je třeba mít MySQL nebo MariaDB databázi, PHP a Composer [2] [9].

Vytvářený obsah se řadí do poliček, knížek, kapitol a stránek. Platforma podporuje *Markdown* formát, ale je možné upravovat stránky pomocí panelu s nástroji. V aplikaci nejsou předpřipravené šablony, ale je možné si stránku označit jako šablonu a následně ji tak používat. Je zde možné nastavit *tag* stránce, knize nebo poličce. *Tagy* zde mohou mít název a hodnotu, příkladem může být například obtížnost textu, jako název je zde obtížnost a jeho hodnota uživateli určí, jak je text náročný na čtení. Operace se stránkami a čtení stránek je možné přes REST API.

V aplikaci je propracované vyhledávání s mnoha filtry, které jsou podrobně vysvětleny v dokumentaci. Je možné vyhledávat přes *tagy* a to i s filtrováním přes hodnoty. Vyhledávání je možné nejen pro celou *wiki*, ale také pro jednotlivé poličky či knihy.

[1] Hostování aplikace je pouze na uživateli, musí ji sám nasadit i provozovat.
[2] Nástroj pro správu knihoven v jazyku PHP.

Zajímavou funkcionalitou je odkazování se na stránky. Do jednotlivé stránky je možné přidat odkaz na jinou stránku nebo pouze na odstavec či část textu z jiné stránky.

Hlavní výhoda této platformy je, že si ji každý uživatel může nastavit podle svého uvážení.

1.3.8 Porovnání aktuálních řešení

V následující tabulce 1.1 jsou vyznačeny rozdíly mezi aplikacemi. Nejprve je porovnán způsob zápisu poznámek, přičemž *md* značí *Markdown* formát, *text* je čistý text a *seznamem* je myšlen bodový seznam. Znak ✓ značí, že daná funkcionalita v aplikaci existuje, znak ✓ značí existující funkcionalitu s nějakým problémem, ✗ značí, že funkcionalita v dané aplikaci neexistuje. Počet hvězdiček značí kvalitu dané funkcionality. Pokud má aplikace neplacenou verzi je to poznačeno ve sloupci *Free*, informace o tom, zda má aplikace placenou verzi je vyznačena ve sloupci *Placené*

Tabulka 1.1 Porovnání řešení.

	Zápis	Free	Placené	API	Tagy	Odkazy	Vyhledávání
Obsidián	md	✓	✓	✓	✗	⋆⋆⋆	⋆⋆
Roam	seznam	✗	✓	✓	✗	⋆⋆⋆	⋆⋆⋆
EverNote	md	✓	✓	✓	✓	✗	⋆⋆⋆
Notin	md	✓	✓	✓	✓	⋆	⋆⋆
OneNote	text	✓	✗	✓	✗	⋆	⋆
Confluence	text	✓	✓	✓	✓	⋆⋆⋆	⋆⋆⋆
BookStack	md	✓	✗	✓	✓	⋆⋆⋆	⋆⋆⋆

Všechny aplikace mají svoje využití, žádná z nich se, ale nehodí na použití glosáře. Všechny *note-taking app* jsou stylizovány jako prostředí složené ze složek, podsložek a souborů. *Wiki* jsou tvořeny stránkami a podstránkami. Je třeba, aby byl glosář jednoduchý a přehledný. Ke každému pojmu by měla existovat jedna stránka, kde mohou být vyznačené odkazy na další pojmy. Přístup by měl být možný přes API, mělo by zde být propracované vyhledávání a umožněno přidávat tagy.

Kapitola 2
Technologie

Kapitola se zaměřuje na poskytováni dat na webu a na normalizaci slov. V první části jsou popsány typy poskytování dat na webu. Nejprve se podkapitola zaměřuje na popis protokolu HTTP a formátů XML a JSON, které jsou pro webové služby důležité. Dále podkapitola popisuje SOAP, REST a GraphQL a porovnává rozdíly těchto tří webových služeb.
Druhá část kapitoly se zaměřuje na normalizaci slov. Stručně shrnuje techniku stematizace a lemmatizace. V kapitole je popsána stematizace pro český, bulharský a mad'arský jazyk.

2.1 Základní technologie

2.1.1 HTTP

HTTP (Hypertext Transfer Protocol) je protokol, který umožňuje posílání zpráv mezi WWW serverem a klientem. Funguje na principu zaslání *requestu* klientem, přičemž následuje zpětná odpověď od serveru. HTTP zpráva je složená z hlavičky a těla. Hlavička obsahuje metadata popisující *request*, případně odpověď. Tělo obsahuje zasílaný soubor (obecně data), který může být například ve formátu HTML, XML ... [10].

Hlavička *requestu* obsahuje vždy jméno metody a poté další metadata specifikující *request*, jako například z jaké webové stránky je *request* posílán, jaký formát nebo kódování je schopen klient přijmout, případně autorizační údaje. V hlavičce odpovědi se vyskytuje třímístný číselný status kód, který udává informaci o úspěchu vyhodnocení *requestu*. Pokud byl *request* vyhodnocen, úspěšně status kód začíná 2. Kódy začínající 4 značí problém na straně klienta, zatímco kódy začínající 5 značí problém na straně serveru. Kódy začínající 3 značí nejčastěji přesměrování, zatímco kódy začínající 1 mají informační charakter [10, kapitola 6]. Nejdůležitější metody HTTP *requestu* využívané pro API společně s jejich běžným významem jsou: GET (získání objektu), PUT (úprava již vytvořeného objektu), DELETE (smazání objektu), POST (vytvoření nového objektu), pro získání pouze hlavičky je možné použít metodu HEAD [10, kapitola 4.3].

Na příkladu 2.1 je vidět *request*, s metodou POST, pomocí které je prováděn dotaz na glosy, jejichž jméno obsahuje slovo *gloss*. V hlavičce je specifikována část URL a o jakou verzi HTTP se jedná, dále hostující server a identifikace klienta. Následuje informace o akceptovatelném formátu a o těle *requestu* – jeho velikost a typ. V *requestu* následuje tělo, které specifikuje GraphQL dotaz.

Odpověď na *request* je vidět na příkladu 2.2. Hlavička obsahuje status odpovědi, kdy byl *request* proveden a informace o těle (velikost, jazyk, typ). Tělo obsahuje všechny glosy odpovídající dotazu.

Výpis kódu 2.1 HTTP request

```
POST /api HTTP/1.1
Host: bc-glossary.herokuapp.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:97.0)
            Gecko/20100101 Firefox/97.0
Accept: */*
Content-Length: 47
Content-Type: application/x-www-form-urlencoded

"query=query{glosses(name: "gloss"){term{name}}}"
```

Výpis kódu 2.2 HTTP odpověď

```
HTTP/1.1 200 OK
Connection: keep-alive
Server: gunicorn
Date: Tue, 05 Apr 2022 10:34:18 GMT
Content-Type: application/json
Vary: Cookie, Accept-Language
X-Frame-Options: DENY
Content-Language: en
Content-Length: 77
Via: 1.1 vegur

{"glosses":[{"term":{"name":"Glossary"}},{"term":{"name":"Gloss"}}]}
```

2.1.2 XML

XML (eXtensible Markup Language) je jasně definovaný jazyk vytvořený pro ukládání a přenášení dat. Tímto jazykem je možné definovat dokumenty a jejich strukturu. Každý XML dokument musí obsahovat *root element*, který je rodičem ostatních *elementů*. *Element* je dvojice *tagů*, mezi nimiž je nějaký text. *Tagy* se píší do špičatých závorek, přičemž ukončující *tag* začíná levou špičatou závorkou a následuje lomítko a poté název *tagu* a ukončující závorka. Název začínajícího a koncového *tagu* se musí shodovat. U počátečního *tagu* se mohou objevovat atributy *elementu* [11, kapitola 3]. Tělo *elementu* může obsahovat další *elementy*. Každý XML dokument by měl začínat prologem, ve kterém je specifikována verze XML a kódování [11, kapitola 2.8]. Výpis kódu 2.3 zobrazuje příklad XML dokumentu.

Výpis kódu 2.3 XML příklad

```
<?xml version="1.0" encoding="UTF-8"?>
<gloss>
    <concept>
        <term>Pes</term>
        <tag>Biologie</tag>
        <tag>Zivocichove</tag>
        <tag>Savci</tag>
    </concept>
    <definition>Ctyrnoha domestikovana selma.</definition>
</gloss>
```

2.1.3 JSON

JSON (JavaScript Object Notation) je odlehčený datový formát. JSON je dobře čitelný a snadno zapisovatelný člověkem. Díky striktním pravidlům je snadné pro stroje JSON analyzovat a generovat [12].

Formát je složen z objektů, které jsou ohraničeny složenými závorkami. Objekt se skládá z párů *název* a *hodnota*, mezi název a hodnotu je vložena dvojtečka a páry jsou odděleny čárkou. Název je vždy v uvozovkách, hodnota může být string, číslo, boolean výraz, nebo pole [12]. Všechny možné případy jsou vidět na příkladu 2.4.

Výpis kódu 2.4 JSON příklad

```
{
  "term": {
    "name": "Limita a derivace funkce ",
    "language": {
      "name": "CZ"
    },
    "tags": ["szz", "zma"]
  },
  "definitions":
  [
      {
      "file": "Necht `f` je funkce definovana na okoli bodu `a`.Pokud
               existuje...",
      "abstract": "abstrakt 1.",
      "active": true
    },
    {
      "file": "Necht `f` je funkce definovana na okoli bodu `b`.Pokud
               existuje...",
      "abstract": "abstrakt 2.",
      "active": false
    }
  ]
}
```

2.2 Poskytování dat na webu

Poskytování dat na internetu zajišťují webové služby. Webové služby jsou zodpovědné za komunikaci mezi servery, počítači nebo počítačovými programy. Odbornější definici představil Gartner [13], který definuje webové služby jako málo provázané softwarové komponenty doručené přes standardní internetové technologie. Z malé provázanosti vyplývá, že by webové služby měly být nezávislé na jazyku, platformě nebo modelu a obecně samy mezi sebou.

Standardy poskytování dat na webu využívají SOA (service-oriented architecture). Při posílání dat vždy jedna strana data posílá a druhá strana data přijímá. Většinou se jedná o *requesty*, tedy zprávy, které obsahují nějaký požadavek (například požadavek na získání glosy) a odpovědi, ve kterých je jednak řečeno, zda byl požadavek zpracován, a případně obsahují také výsledek požadavku.

Mezi webové služby se řadí webová API (dále bude použito pro účely tohoto textu pouze API), která jsou klíčová pro přístup k softwaru přes internet. API slouží k přenosu informací z jedné služby do služby jiné a jsou specifická svým HTTP rozhraním [14].

Způsobů jak implementovat API je mnoho. Známými standardy jsou např. RPC, protokol SOAP, architektura REST, jazyk GraphQL, gRPC a další. Probrání všech způsobů implementace

by bylo vyčerpávající, a tak se tato kapitola zaměřuje pouze na tři vybrané koncepty (SOAP, REST a GraphQL).

2.2.1 SOAP

SOAP neboli *Simple Object Access Protocol* (jednoduchý protokol pro přístup k objektům) byl vymyšlen jako protokol, který definuje RPC mechanismus. Protokol definuje způsob komunikace mezi klientem a serverem přes internetovou síť. Pro transport dat využívá HTTP/HTTPS, avšak může využívat i jiné protokoly pro přenos (SMTP, TCP, UDP...). Data požadavků a jejich odpovědí jsou uchovávány v XML dokumentu, který je přenášen [15].

Vývoj SOAP začal v roce 1998, kdy se nejvíce zaměřoval na definování typového systému, tedy klasifikace výrazů podle jejich druhu. Při vývoji SOAP se neuvažovalo pouze využití XML, ale také jiných serializačních formátů. Rozhodnutí používat XML padlo kvůli tomu, že se ve stejnou dobu výzkumná skupina Microsoftu zabývala právě vývojem formátu XML. O rok později (1999) se XML ukázalo jako velmi slibný formát a tak se vývojáři SOAP na tomto formátu shodli [16].

XML zpráva v SOAP obsahuje až čtyři části. Nejprve je definovaný *root element*, následuje nepovinná hlavička (*header*) a povinné tělo (*body*) zprávy. Hlavička obsahuje dodatečné informace, které je potřeba poslat s HTTP *requestem*. Pokud je hlavička přítomná, musí následovat jako první *element* po *root* [15, kapitola 4]. Poslední částí je nepovinná informace o chybě (fault), pokud se nějaká chyba vyskytla během zpracování zprávy, informace o ní bude právě v tomto *elementu*.

Na příkladu 2.5 je vidět poslání *requestu* přes HTTP. V *root elementu* je specifikován *namespace* pro SOAP. Hlavička je pro tento *request* prázdná, mohla by ale obsahovat například číslo transakce s atributem *mustUnderstand* [15, kapitola 4.2.3]. Pokud je tento atribut nastaven na 1, klient musí rozumět danému *elementu* v hlavičce. Kdyby mu nerozuměl, *request* by se nezpracoval. V těle je specifikováno, že je volaná metoda `GetGloss`, která přijímá jeden atribut a to hledaný pojem a vrací definici pojmu.

Výpis kódu 2.5 SOAP poslání requestu

```
POST /api  HTTP/1.1
Host: bc-glossary.herokuapp.com
Content-Type: text/xml
Content-Length: 240

<SOAP:Envelopexmlns:SOAP="http://schemas.xmlsoap.org/soap/envelpe/">
  <SOAP:Header>
  </SOAP:Header>
  <SOAP:Body>
    <glossary:GetGloss xmlns:glossary="bc-glossary.herokuapp.cm/api">
      <name>Pes</name>
    </glossary:GetGloss>
  </SOAP:Body>
</SOAP:Envelope>
```

Na příkladu 2.6 je vidět odpověď na předchozí *request* . Je zde HTTP status kód odpovědi (200 tedy úspěch). V těle je vidět odpověď na *GetGloss*, tedy daná definice k glose s názvem *Pes*.

SOAP je standardizovaný protokol se striktními pravidly. Data mohou být uchovávána pouze v XML souborech. Zabezpečení, autorizace a správa errorů je zabudovaná přímo v protokolu. SOAP je vhodný pro velké projekty, kde je vyžadována vysoká bezpečnost a jsou zde prováděny složitější transakce či dotazy. SOAP API využívá například *PayPal* - americký internetový platební systém. Kvůli striktním pravidlům a masivnosti SOAP může být náročné se tento protokol naučit.

Výpis kódu 2.6 SOAP odpověď na request

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 294

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
       <glossary:GetGloss
        xmlns:glossary="bc-glossary.herokuapp.com/api">
           <definition>
                Ctyrnoha domestikovana selma. Jedna se nejspis
                o zdomacneleho vlka obecneho.
            </definition>
       </glossary:GetGloss>
   </SOAP:Body>
</SOAP:Envelope>
```

2.2.2 REST

REST (REpresentational State Transfer) byl poprvé představen v disertační práci Roye Fieldinga, který se podílel na vývoji HTTP 1.0 a 1.1. Jedná se o architektonický styl definující pravidla pro návrh aplikací, které přenášejí data a komunikují skrze HTTP. Přičemž se jedná o tato pravidla [17, kapitola 5]:

Klient-Server Jelikož uživatelské rozhraní a uložení dat by mělo zůstat nezávislé, tak by na sobě klient a server neměli být závislí. To znamená že klient a server by spolu měli komunikovat pouze na principu *request* a *response* (požadavek a odpověď).

Bezstavové Komunikace klient-server musí být bezestavová, tedy *request* musí obsahovat všechny potřebné informace.

Cache Odpověď serveru musí obsahovat informaci o tom, zda jsou *cacheable* nebo *non cacheble*, tedy zda je možné data uložit do mezipaměti. Pokud je možné data uložit do mezipaměti, klient se může vyhnout zbytečnému opakování dotazů.

Jednotné rozhraní *Requesty* od různých klientů by měly vypadat stejně. *Requesty* operující s jedním objektem by měly mít stejnou URL (jako například pro tagy URL */tags*). Jaká operace bude provedena, je určeno například HTTP metodou (GET, POST ...). URL by měly být tvořeny systematicky a tedy by mělo být snadné uhodnout URL, odkazující na specifický objekt.

Vrstvy v systému Pokud je mezi klientem a serverem nějaká další vrstva (například pro uchovávání mezipaměti), nemělo by to ovlivnit *request* ani jeho odpověď. Systém se tedy tváří, jako by tam taková vrstva ani nebyla.

Kód na vyžádání Klient může obdržet v odpovědi spustitelný kód. Jedná se o volitelné pravidlo, protože sice zlepšuje možnost rozšířitelnosti ale snižuje transparentnost.

REST API je často vytvořeno k aplikacím pro jejich snazší použití jinými aplikacemi. Například Twitter (sociální síť) umožňuje odesílat zprávy přes REST API. REST API přejímá metody od HTTP jako je GET - získání objektu, POST - vytvoření objektu, PUT - změna objektu a DELETE - smazání objektu. REST také přejímá status kódy HTTP, které značí úspěch či neúspěch *requestu* a nesou další informaci o něm (201 - vytvořeno, 403 - zakázání přístupu ...).

Z tohoto důvodů se REST implementuje zpravidla přes HTTP, ale může přenášet data v mnoha formátech jako například XML, JSON, CSV, YAML ...

Základem REST API jsou URL, které jsou volány pro specifickou HTTP metodu. Tyto URL by měly být intuitivní výstižné a co možná nejkratší [18]. Na příkladu 2.7 je vidět několik URL, první odpovídá dotazu, který by měl vrátit všechny glosy (bude využívat metodu GET). Druhá URL odpovídá dotazu, který by měl vrátit glosu s `id = 1`. Třetí URL představuje dotaz který by měl glosu vytvořit (bude využívat metodu POST). Pro vyhledání všech glos obsahujících slovo pes by mohla posloužit čtvrtá URL a pro obecné hledání, které může vyhledat glosu, jazyk nebo tag, je vytvořena pátá URL.

Výpis kódu 2.7 URL

```
/glosses
/glosses/1
/glosses
/glosses?q=pes
/search?q=pes
```

Je vhodné při návrhu API myslet na status kódy. Je třeba použít alespoň tyto tři status kódy: 200 - OK, 400 - Bad Request, 500 - Internal Server Error. Další možné rozšíření status kódů je o kódy informující o vytvoření, nemodifikaci, zamítnutém přístupu, nebo nenalezení (201, 304, 401, 404) [18].

Ukázka kódu 2.8 představuje poslání *requestu*. Volá se URL `api/glosses` s použitím metody POST, jelikož se jedná o vytvoření objektu. HTTP hlavička dále obsahuje specifikaci typu který je odesílán a přijímán. Po prázdném řádku již následují data, která zpracuje metoda REST API.

Výpis kódu 2.8 REST poslání requestu

```
POST /api/glosses HTTP/1.1
HOST: bc-glossary.herokuapp.com
Content-Type:application/json
Accept:application/json

{
    "gloss": {
        "definition": "Male zviratko, ktere se vejde do ruky a ma ocasek
                       a kolecko.",
        "name": "mys",
        "language": "cz"
  }
}
```

Na příkladu 2.9 je ukázka odpovědi. HTTP hlavička obsahuje status kód 201 - glosa byla vytvořena, typ odpovědi a její velikost. Po prázdném řádku následuje tělo, které obsahuje vytvořenou glosu.

2.2.3 GraphQL

GraphQL je dotazovací jazyk vyvinutý společností *Facebook (Meta Platforms)* v roce 2012, přičemž od roku 2015 se jedná o open-source [19]. Jazyk se začal vyvíjet ve chvíli, kdy se začaly přepisovat mobilní aplikace *Facebooku*. Vývojáři byli frustrovaní tím, jaký byl rozdíl mezi daty která potřebovali a mezi daty která přijali jako odpověď na *request*. Díky tomu se rozhodli vyvinout GraphQL, které umožňuje klientovi nadefinovat si strukturu odpovědi [20].

Pro operaci získání objektu (READ) se v GraphQL využívá *query*. Pro ostatní CRUD operace, tedy vytvoření nového objektu, smazání objektu nebo úprava objektu se používají *mutation*. Na server se vždy odesílá textový řetězec, který definuje jakou metodu má server provést a jak by

Výpis kódu 2.9 REST odpověď na request

```
HTTP /1.1 201 Created
Content - Type : application/json
Content - Length : 146

{
    "gloss": {
        "id": 1,
        "definition": "Male zviratko, ktere se vejde do ruky a ma ocasek
                       a kolecko.",
        "name": "mys",
        "language": "cz"
    }
}
```

měla vypadat odpověď. Odpověď tedy zrcadlí strukturu dotazu [20], jako je vidět na příkladech 2.10 a 2.12.

Ukázka kódu 2.10 představuje *request*, nejprve je definována volaná metoda. V tomto případě *glosses* - vrátí všechny glosy s danými parametry (jazyk glosy je čeština, tagy glosy jsou *animal* a *mammal*). Poté se definuje struktura dat, která mají být vrácena. U termu je *requestem* požadováno jméno a tagy, ale není požadován jazyk, u definice je *requestem* požadován obsah souboru a jazyk, ale není požadován abstrakt.

Výpis kódu 2.10 GraphQL request

```
query{
  glosses(language:"cz", tags:["animal", "mammal"]){
    term{
      name,
      tags{}
    }
    definition{
      file,
      language{
        name
      }
    }
  }
}
```

Odpověď na *request* může obsahovat nějaká data a nějaké errory a měla by být ve formátu JSON [21]. Jeho struktura by měla vypadat následovně 2.11, přičemž ani data ani pole errorů není povinné.

Výpis kódu 2.11 GraphQL struktura odpověďi

```
{
  "data": { ... },
  "errors": [ ... ]
}
```

Na *request* z příkladu 2.10 odpovídá ukázka kódu 2.12. V tomto případě se nevyskytly žádné chyby a tak je v odpovědi přítomný pouze objekt *data*, který obsahuje všechny glosy odpovídající dotazu. Lze si povšimnout, že odpověď skutečně obsahuje pouze ta data, o které *request* žádal.

Jazyk GraphQL je silně typovaný, tedy každá část dotazu odpovídá specifické části odpovědi. Pokud by dotaz neodpovídal struktuře nějakého objektu, dotaz nebude vůbec vyhodnocen a rovnou bude vyhozena výjimka [20]. GraphQL umožňuje dotazovat se na typy objektů, díky čemuž bylo vytvořeno IDE GraphiQL, které usnadňuje vytváření dotazů [20].

Výpis kódu 2.12 GraphQL odpověď

```
"data": {
    "glosses":
    [
       {
        "term": {
            "name": "kocka",
            "tags": [
                {
                  "name": "animal"
                },
                {
                  "name": "mammal"
                }
            ]
        },
        "definition": [
            {
                "file": "Kocka je selma. Existuje mnoho druhu
                         kockovitych selem.",
                "language": {
                    "name": "cz"
                }
            }
        ]
        },
    ]
}
```

Skládání složitějších dotazů je umožněno přes *fragmenty*. Jedná se o jednotku, která je strukturovaná jako dotaz na část objektu nebo objektů [22]. V jiném jazyce by se o *fragmentu* dalo uvažovat jako o proměnné s daným jménem a typem, případně by se dal brát jako funkce, která má definovaný návratový typ. *Fragment* se může vložit do dotazu, jako je vidět na příkladu 2.13, kde je použit v dotazu *glosses*, který je rozdělen na české a anglické glosy. Tím se může programátor vyhnout zbytečnému opakování kódu.

GraphQL *requesty* mohou být posílány přes HTTP, ale spojení může být zajištěno i jiným protokolem. Metoda GET v HTTP může vypadat jako zde 2.14, kde je posílán dotaz na všechny termíny a jejich jména.

Metoda POST, kde může být použito *mutation* i *query*, musí v HTTP hlavičce obsahovat `Content-Type: application/json`. Tělo může obsahovat až tři části: `query` ve které je napsán GraphQL string, přičemž začíná pojmenováním operace - v případě na ukázce 2.15 se jmenuje *TagQuery*. Druhá část `operationName` nese informaci o pojmenování operace a poslední část `variables` je množina dvojic argumentů a jejich hodnot.

Výpis kódu 2.14 GraphQL a GET

```
GET http://bc-glossary.herokuapp.com/api?query={terms{name}}} HTTP/1.1
```

Výpis kódu 2.13 Fragment

```
{
  czech: glosses(language: "cz"){
    ...glossField
  }
  english: glosses(language: "en"){
    ...glossField
  }
}

fragment glossField on GlossType{
  term{
    name
  },
  definition{
    file
  }
}
```

Výpis kódu 2.15 GraphQL a POST

```
POST /api/ HTTP/1.1
HOST: bc-glossary.herokuapp.com
Content-Type:application/json
Accept:application/json

{
    "query": "query TagQuery ($arg1: String)
        {
        tag(name: arg1){
            tagType{name}
            }
        }",
    "operationName": "TagQuery",
    "variables": {
        "arg1": "test"
    }
}
```

2.2.4 Porovnání

SOAP má zabudované zabezpečení, autorizaci a správu errorů. Protokol může používat pouze masivnější formát XML, který pro svou velikost může ovlivňovat rychlost přenosu. Pro transport dat přes internet může SOAP využívat i jiné protokoly než je HTTP. Vybudování SOAP API je velmi náročné, protože je potřeba rozumět všem protokolům, které využívá a je třeba se naučit přesně strukturovat zprávy. Jelikož má SOAP vysoké zabezpečení, používají ho především firmy zaměřené na finance, státní správa nebo velké korporátní společnosti.

REST umožňuje využívat jiné formáty než je XML, ale neumožňuje použít jiný transportní protokol než HTTP. Komunikace mezi klientem a serverem je samopopisná, takže není potřeba další dokumentace k API; přenášená data jsou kvůli tomu velkého rozsahu, což může mít za následek zpomalení přenosu dat. Status kódy jsou přejaté z HTTP, díky tomu a intuitivnosti je učící křivka rychlá. REST jako jediné API využívá HTTP caching (ukládání do mezipaměti). Jedním z problémů REST API je, že neexistuje žádná kontrola na dodržování pravidel pro jeho vytváření, tím vzniká až přílišná volnost v implementaci.

Hlavní výhodou GraphQL je, že umožňuje klientovi popsat, jaká data si žádá, čímž nejsou zbytečně přenášena data, která nejsou potřeba. GraphQL využívá pouze datový formát JSON, ale je schopné používat i jiný transportní protokol než HTTP. API nepoužívá číslované verze, pouze přidává nové funkcionality. GraphQL zprávy mohou být často velmi náročné, kvůli jejich vnořené struktuře, což může vést ke snížení výkonnosti. GraphQL je vhodné pro API, kde je pravděpodobné, že klient bude vyžadovat jen určitá data a kde nebude mnoho požadavků na různé úpravy objektů.

Tabulka 2.1 Základní porovnání

	SOAP	**REST**	**GraphQL**
Architektura	strukturovaná XML zpráva	6 pravidel	schéma s typy
Formát	XML	XML, JSON, CVS, YAML, HTML ...	JSON
Protokoly pro přenos	HTTP, SMTP, TCP, UDP...	HTTP	HTTP, TCP, WebSockets ...
Učící křivka	pomalá	rychlá	střední
Využití	finanční a telekomunikační služby, systémy které potřebují vysoké zabezpečení	veřejná API	mobilní API, micro služby

2.3 Normalizace slov

Přirozený jazyk obsahuje slova, která se mohou vyskytovat v různých tvarech, ať už důsledkem skloňování, časování nebo použitím množného čísla. Tato práce se zaměřuje na glosář a aby byl glosář opravdu dobře využitelný, je třeba zajistit, že i při použití jiného než základního tvaru slova, bude slovo v glosáři vyhledáno. To je možné zajistit vhodnou úpravou (normalizací) slova do základní formy, která je následně porovnána se slovy v glosáři. V tomto textu je popsána technika lemmatizace a stematizace.

2.3.1 Stematizace

Stematizace je technika normalizace slov, která nalezne kmen (předpony a jádro slova) slova pomocí odstranění koncovek (část slova na konci), a přípon (část slova za kořenem). V některých případech, podle jazyka a potřeby, je odstraněna i předpona (část slova před kořenem). Jedná se o odstranění částí slov vytvořených skloňováním nebo odvozením [23]. Na slově *poprosit* jsou vyznačeny všechny čtyři části slova včetně předpony (část slova před kořenem).

po | pros | i | t

Stematizace by měla nalézt kořen *pros* nebo *popros* (podle potřeby). Stematizace lze rozdělit na dvě části [23], nejprve se oddělí nejdelší možná koncová část slova, tak aby zbytek odpovídal jinému slovu. Druhá část se zabývá výjimkami v pravopisu - mírné odlišnosti v kořeni. Jelikož je každý jazyk jinak postaven, musí se ke každému jazyku přistupovat individuálně.

Stematizací pro český, bulharský a maďarský jazyk se zabývali Dolamic a Savoy [24]. Pro každý z těchto jazyků vytvořili několik pravidel, jak odstraňovat koncovky.

Maďarština je bezrodý jazyk, rozlišuje množný a jednotný rod, má tři časy, 18 pádů [25] a je velmi bohatá na složeniny slov. U vytváření pravidel pro stematizaci se Dolamic a Savoy zaměřili hlavně na skloňování a rozdělení slov pomocí rozkladového algoritmu [26]. Tento algoritmus funguje tak, že se nejprve analyzuje počet výskytů slov v nějakém textu a vytvoří seznam těchto slov. Pro nalezení rozkladu slova je slovo rozděleno na dvě, tak že se oddělí posledních k písmen. Počáteční hodnota k je rovna 4. Algoritmus se podívá, zda se tyto dvě slova vyskytují v seznamu, pokud se tam slova nevyskytují, zvětší k o jedničku. V opačném případě se rekuzrivně zavolá na nově vzniklá dvě slova. Algoritmus vrací strom možných rozkladů, přičemž pro správný rozklad je možné se rozhodnout například podle počtu výskytů slov v seznamu.

Bulharština je jazyk psaný cyrilicí (azbukou), má všechny tři rody, několik časů (dva budoucí, čtyři minulé, dva budoucí v minulosti, přítomný) a pouze dva pády. Zajímavostí bulharštiny je gramatický člen (*the* v angličtině), který se připojuje na konec slova. Dolamic a Savoy se zaměřili na precizní přepsání cyrilice do latinky, vytvořili 5 pravidel pro odstraňování členů, 8 pravidel na odstranění množného čísla, poté přidali několik pravidel pro normalizaci (odstranění nebo výměna písmen) a 3 pravidla určená k změkčení výrazů.

Čeština má 7 pádů, tři rody, tři časy a velmi složité stupňování přídavných jmen. Dolamic a Savoy vytvořili 52 pravidel, které pomohou při odstranění přípon spojených s číslem, pádem a rodem. Pro změnu samohlásek a změkčování vymysleli pět normalizačních pravidel. A nakonec implementovali odstranění často používaných přípon.

2.3.2 Lemmatizace

Další normalizační technikou je lemmatizace, která k určitému slovu vyhledá základní tvar - lemma. Technika lemmatizace nezredukuje slovo na kořen, ale nahradí koncovky a přípony jinou koncovkou, tak aby vznikl základní tvar slova. Výsledek lemmatizace může být v některých případech stejný jako výsledek stematizace například u slova *stanovat* bude v obou případech výsledek *stan*.

Při vytváření pravidel je nejprve třeba nalézt kořen slova, respektive tu část slova která je společná pro slovo v základním tvaru a slovo vyčasované/vystupňované. Podle toho je možné určit záměnu koncovek, tak aby výsledné slovo bylo rovné základnímu tvaru [27]. Tím pádem je lemmatizér určitým způsobem závislý na testovacích datech, protože pokud se nějaký tvar slova nevyskytl v trénovacích datech, lemmatizér nedokáže přesně odhadnout, jakou část slova má zaměnit.

Lemmatizace se také odvíjí od kontextu ve kterém se slovo vyskytuje, čímž mohou být správně rozpoznána dvojsmyslná a nejednoznačná slova. Díky kontextu je možné i rozhodnout o záměně koncovky u slov, která nebyla v testovacích datech, avšak účinnost není tak markantní jako u nejednoznačných slov [28].

Lemmatizér funguje tak, že dostane základní sadu slov, ze kterých se snaží odvodit pravidla pro záměnu konců slov, tak aby se došlo k základnímu tvaru slova. Občas dostává lemmatizér značky (anglicky *tags*), které mu dále určují jaký tvar slova má použit u nejednoznačných slov, případně si lemmatizér musí sám značky vytvořit z kontextu [28].

2.3.3 Porovnání

Stematizace je jednodušší na implementaci a rychleji najde výsledek, protože se pracuje vždy pouze s jedním slovem, od kterého se odstraní přípony a koncovky. Na druhou stranu je stematizace méně precizní, nepracuje totiž s kontextem a tedy nedokáže rozlišit homonyma (slova která mají stejnou podobu, ale jiný význam) jako je například slovo *los*, které může představovat zvíře, ale také lístek do loterie. Dalším problémem stematizace je neschopnost spojit slova, která mají stejný význam v základní formě, ale různý tvar ve formě vyčasované nebo vystupňované; jako například přídavná jména *dobrý* a *lepší*, nebo slovesa *bude* a *je*. Lemmatizace tyto dva problémy do určité míry řeší, avšak je složitější na implementaci a je výpočetně náročnější.

Kapitola 3
Analýza

Kapitola obsahuje specifikaci funkčních a nefunkčních požadavků. Dále poskytuje rozbor způsobů reprezentace a uložení glosy, společně s návrhem doménového modelu. V kapitole je popsáno, jaké funkcionality by mělo mít API. Následuje krátké shrnutí požadavků na webový frontend.

Cílem této práce je vytvořit aplikaci, umožňující zaznamenávat pojmy a *know-how* ve formě glos. Přičemž glosa je záznam složený z názvu a z definice. Záznam může být rozšířen o doplňující informace jako je jazyk, ve kterém je glosa napsána, nebo okruh, kterého se glosa týká. Aplikace by měla umožňovat vytvoření pojmu, jeho následnou úpravu, či případné smazání. Dále je žádané umožnit zobrazení všech glos a nebo zobrazení vyfiltrované části glos. Úprava a zobrazení glos je nutné zpřístupnit přes webovou aplikaci. Všechny funkcionality musí být dostupné skrze API.

3.1 Specifikace požadavků

Pomocí analýzy existujících řešení a nároků na aplikaci byly vybrány požadavky, které by měla aplikace splňovat. Aplikace by měla umožňovat vytváření záznamů glos a následně je upravovat a mazat. Glosář by dále měl umožňovat vytváření a přiřazování tagů ke glosám. Tagy budou chápány jako označení tématu, kterého se glosa týká. Jednotlivé glosy musí náležet do nějakého konceptu. Koncept spojuje glosy se stejným významem, ale v jiném jazyce. Glosa musí mít jazyk a nemůže existovat glosa, která má stejný jazyk a koncept jako jiná glosa. Může existovat koncept, který má více glos v různých jazycích.

Další funkcionalitou bude vyhledání glos v textu - aplikace nalezne všechny výskyty uložených glos v daném textu a tyto glosy následně zprostředkuje uživateli. Aplikace bude rozdělena na dvě části, a to webový frontend a API. Přes API bude možné provádět všechny operace s glosami, jako je přidání, změnění, smazání nebo vyhledání glosy, ale i dílčí úpravy glos, vytváření tagů a jazyků. Hlavní funkcionalitou webového frontendu bude zobrazování glos, bude možné glosy upravovat, přidávat, mazat a vyhledávat v nich, nebude umožňovat dílčí úpravy.

3.1.1 Funkční požadavky

F1: Zobrazení glosy Glosy musí být zobrazitelné. Musí být možné zobrazit jednu glosu, všechny glosy, nebo výběr glos.

F2: Vyhledání glosy Uživatel může vyhledat glosu podle názvu. Hledanou glosu může specifikovat přes jazyk nebo přes tagy. Glosa se mu vrátí. Glosa je definovaná konceptem a jazykem. Každý koncept může mít pouze jednu glosu v jednom jazyce.

F3: Vyhledání podle tagů Uživatel může vyhledat všechny glosy s daným tagem. Glosy se mu vrátí.

F4: Vyhledání v definicích na webu Na webovém forntendu musí být možné hledat pojmy nejen v názvu glos, ale také v definicích glos.

F5: Vyhledání glos v textu Uživatel může nechat vyhledat glosy v textu. Může specifikovat tagy a jazyk. Uživateli se vrátí názvy a definice všech nalezených glos.

F6: Přidání glosy Uživatel může vytvořit novou glosu s názvem pojmu a definicí.

F7: Úprava glosy Uživatel může změnit již vytvořenou glosu. Může změnit název, definici, přidat tagy, umazat tagy, změnit jazyk.

F8: Smazání glosy Uživatel může glosu smazat.

F9: Jazyk glosy Glosa musí být v nějakém jazyku.

F10: Vytvoření vícejazyčných glos Záznam o glose musí mít specifikovaný jazyk. Pro každý jazyk a koncept může existovat právě jedna glosa, která má právě jeden název glosy. Pro jeden koncept může existovat více glos v různých jazycích.

F11: Více stejnojmenných glos Může existovat několik glos se stejným jménem ve stejném jazyce, pokud popisují jinou věc. Čeho se glosa týká, je rozlišeno tagem, který poté musí být u obou glos přítomen.

F12: Vytvoření tagu Tag může být vytvořen bez toho, aby byl přiřazen k nějaké glose.

F13: Přidání tagu Uživatel může přidat tag nebo více tagů ke glose.

3.1.2 Nefunkční požadavky

N1: API Aplikace musí být přístupná přes API.

N2: Webový frontend Aplikace musí mít webový frontend.

N3: Markdown Aplikace musí zvládat Markdown formát a musí umožnit uživateli snadno tvořit a upravovat texty glos v tomto formátu.

N4: Spolehlivost Aplikace musí být spolehlivá.

N5: Licence Aplikace bude zcela zdarma a kód zveřejněn jako open-source.

N6: Intuitivnost Webový frontend aplikace musí být intuitivní.

3.2 Reprezentace a uložení glosy

Název glosy by měl být výstižný a nepříliš dlouhý, protože by měl reprezentovat pojem. Pojem může být slovo, sousloví nebo zkratka, které pojmenovává nějakou entitu (věc, vlastnost, osobu ...). Definice může být naopak rozsáhlejší, může se skládat z jednoho slova, z vět, z rozsáhlého textu, obrázků, diagramů, částí kódu atd. Zatímco název glosy může být uchováván jako prostý text, pro definici by bylo vhodné použít komplexnější textový formát. Jelikož se glosy budou uchovávat v databázi, budou přístupné přes API a budou zobrazovány na webu, je třeba aby byl formát snadno uchovávatelný, snadno zobrazitelný na webu a byl snadný na zápis.

Text by bylo možné uchovávat jako HTML, protože se jedná o jazyk vyvinutý pro zobrazování na webu. Avšak HTML je značkovací jazyk, který není uživatelsky příjemný na psaní definic. Další možností je psát definice v prostém textu, případně PDF formátu, což bude příjemnější pro

uživatele, ale nemusí být jednoznačná konverze do HTML tak, aby se text správně zobrazoval na webu. Jako přijatelná možnost se nabízí psát/ukládat definice v *Markdown* formátu. *Markdown* je jazyk vyvinutý přímo na konverzi textu na HTML tagy, je populární a snadno se v něm píše i čte.

Markdown je značkovací jazyk pro vytváření formátovaného textu s konverzi do HTML. Vytvořil ho John Gruber s pomocí Aarona Swartze. Je snadný na čtení i psaní, přičemž umožňuje psát nadpisy o šesti velikostech, paragrafy, seznamy a code bloky, obrázky, odkazy. Každá z těchto formátovacích funkcionalit má protějšek v HTML. *Markdown* a jeho konverze do HTML je vidět na příkladech 3.1, 3.2. Popis těchto technologií není cílem práce podrobnější informaci lze nalézt zde `https://daringfireball.net/projects/markdown/` pro *Markdown*, pro HTML lze nalézt dokumentaci zde `https://html.spec.whatwg.org/multipage/`.

Výpis kódu 3.1 Markdown text

```
# Heading level 1

1. First item
2. Second item



- Third item
- Fourth item


* First item

    * Second item


## Heading level 2

This is paragraph
**bold text**
__bold text__
_italic text_
*italic text*

![Nazev obrazku](/assets/images/tux
    .png)


`code`
```

Výpis kódu 3.2 HTML text

```
<h1>Heading level 1</h1>
<ol>
  <li>First item</li>
  <li>Second item</li>
</ol>

<ul>
  <li>Third item</li>
  <li>Fourth item</li>
</ul>
<ul>
  <li>First item</li>
    <ul>
      <li>Second item</li>
    </ul>
</ul>
<h2>Heading level 2</h2>

<p>This is paragraph</p>
<strong>bold text</strong>
<strong>bold text</strong>
<em>italic text</em>
<em>italic text</em>

<img src="/assets/images/tux.png"
    alt="Nazev obrazku" style="width
    :auto;">

<code>code</code>
```

Výše bylo zmíněno, že záznam může obsahovat doplňující informace, jako je jazyk nebo informace o okruhu, kterého se glosa týká. Zaznamenat, jakého okruhu se glosa týká, můžeme například spojením se všemi glosami, které se buď týkají dané glosy nebo se zabývají stejným tématem. Tím by byl pro daný záznam vytvořen okruh podobných glos, je možné si ho představit jako myšlenkovou mapu. Propojení glos by mohlo být velmi náročné, nejprve by se musely najít všechny glosy, se kterými by měl být daný záznam spojen. Nalezení glos by mohlo být strojové, například by se prošla definice a záznam by se propojil se všemi glosami, které se vyskytly v definici. Toto řešení by bylo časově náročné a značně nepřesné.

Alternativně by glosy, se kterými by měl být záznam spojen, vkládal uživatel. Uživatel by musel znát všechny glosy, nebo je složitě vyhledávat. Dalším problémem by bylo vyhledávání. Jelikož by mohlo existovat mnoho spojů, mohlo by se stát, že jedna glosa bude přes jinou glosu

spojena s pojmem, který s ní vůbec nesouvisí. Vyhledávání v takto provázané struktuře by bylo časově náročné, nehledě na to, že by bylo složité definovat, podle čeho se okruh má vyhledat. Navíc by spoje musely být obousměrné, aby bylo možné nalézt všechny glosy související s daným pojmem.

Místo vytváření okruhu pro glosu je možné zařadit glosu do již existujícího okruhu. Každá glosa může mít přiřazeno několik tagů, které definují téma nebo část tématu, do kterého glosa spadá. Tag by mohl mít i stejný název jako jiná glosa, ale nebyl by glosou a tím by nedocházelo ke spojení glos přímo, ale skrze tagy. Pro uživatele by bylo jednodušší zadat okruh, kterého se glosa týká, než vytvářet spojení s každou glosou. Vyhledání glos by bylo také snazší a intuitivnější. Tag by měl být nejlépe jednoslovný název, definující téma kterého se glosa týká (IT, živočichové, plaz, ...), také by mohl obsahovat název a hodnotu (náročnost 5, spokojenost 4, ...), nebo název a obrázek (autor, fotka). Jelikož může být mnoho možností, jak by měly tagy vypadat, je důležité umožnit rozšiřitelnost tagů pro budoucí použití.

Bylo by vhodné dokázat propojit glosy, které se týkají stejného pojmu, ale jsou napsány v jiném jazyce. Například pojem *myš* s tagy *zvíře*, *savec*, *hlodavec* popisuje stejného živočicha jako pojem *mouse* s tagy *animal*, *mammal*, *rodent*. Díky propojení pojmů by mohl být glosář použit i na překlady, přičemž by nemuselo dojít k nepříjemné záměně pojmů s homonymy při prostém přeložení názvu glosy.

Pro některé typy záznamů by bylo vhodné uchovávat historii definic. Historie definic se může hodit pro korekci chyb nebo pro obecný přehled o vývoji definice. Praktická potřeba historie definic by byla, pokud by glosář uchovával zákony nebo pravidla, kde je vhodné znát přesné znění předchozí verze zákona nebo pravidla.

3.2.1 Tvorba doménového modelu

Na základě poznatků rozepsaných výše byl navrhnut doménový model tak, aby vyhovoval požadavkům specifikovaným v kapitole 3.1. Doménový model (obrázek 3.1) se zaměřuje na uchovávání dat, tedy glos, v glosáři.

Glosa samotná je zastřešena konceptem, který uchovává její ID a spojuje pojem s jeho definicí. V glosáři mohou existovat vícejazyčné glosy, které mají stejný koncept a značí stejný objekt (například *car* v anglickém jazyce a *auto* v českém jazyce). Aby se rozlišilo, která definice náleží kterému pojmu, existuje ještě další spojení a to přes jazyk. Jestliže pojem a definice náleží do stejného konceptu a zároveň jsou spojeny se stejným jazykem, pak definují glosu.

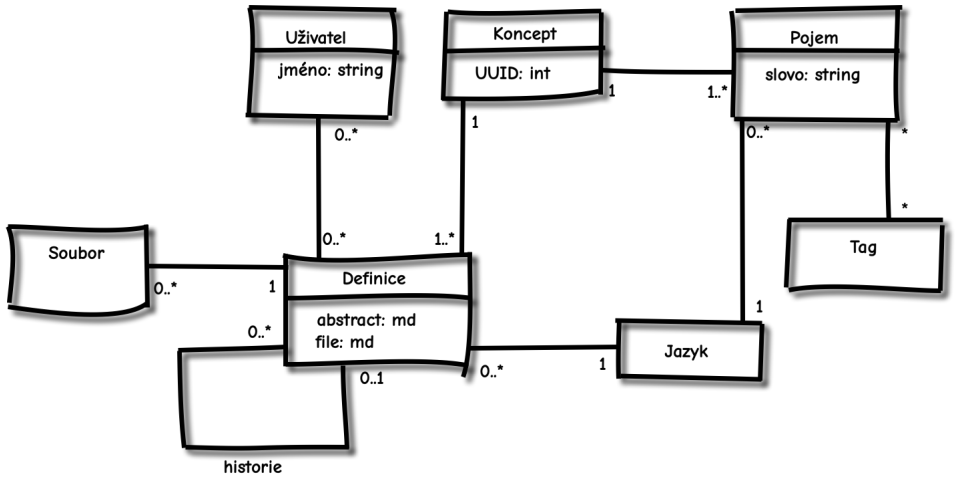
Pojmy mohou být označeny *tagem*, nebo více *tagy*. *Tag* je v tomto případě modelován jako jednoslovný či víceslovný pojem definující okruh, kterého se glosa týká. Je možnost rozšířit aplikaci o další typy *tagů*, jako je například *tag* složený z pojmu a číselné hodnoty.

Pro uchovávání historie změn byla vymodelována závislost definic. Jedna definice může mít nejvýše jednoho přímého rodiče. Z jedné definice může vycházet několik dalších definic. Tímto namodelováním je umožněno vytvářet více úprav vycházejících z jedné definice. K definici může být přiřazen uživatel, který jí vytvořil, aby bylo možné zjistit, kdo kterou změnu provedl.

Definice budou uchovávány například jako *Markdown* soubory. Definice bude složená ze souboru, ve kterém je definice popsána, a nepovinného abstraktu, který se může používat při zobrazení náhledu významu. Pokud budou v textu definice použity obrázky, diagramy nebo jiné externí soubory, je třeba je také uchovávat. To je zde vymodelováno pomocí entity *soubor*.

3.3 Zpřístupnění přes API

Aplikace musí být přístupná přes API. Aplikace by mohla mít přístupné některé operace přes API a některé operace přes webový frontend. API by mohlo být například pouze *read only*, tedy pouze pro získání objektů. Další možností je zpřístupnit přes API všechny operace, přičemž je



Obrázek 3.1 Doménový model

nutné specifikovat, co *všechny* znamená. Poslední dále probranou možností bude zpřístupnění některých operací některým uživatelům.

Pokud by bylo API *read only*, umožňovalo by uživatelům pouze číst glosy. API by mělo mít tyto funkcionality:

- **Získání glosy** na základě jejího id (koncept + jazyk), nebo na základě jména, tagů, případně jiných filtrů. Glosa by měla obsahovat *pojem*, *definici*, *jazyk* a případné *tagy*.
- **Získání všech glos**, přičemž by bylo možné glosy vyfiltrovat (podle jazyka, tagu, ...)
- **Získání všech tagů** pro zjištění, jaké okruhy jsou dostupné
- **Získání všech jazyků** pro zjištění jaké jazyky jsou dostupné
- **Získání historie ke glose**, respektive získání historie definic. API by vrátilo všechny předchozí definice k nějaké glose dané jazykem a konceptem. Každá definice by byla označena datem a případně uživatelem, který ji vytvořil.
- **Získání pojmu** – zde jsou důležité obě funkcionality: získání všech pojmů i získání jednoho pojmu na základě nějakých filtrů nebo přesného určení přes jazyk a koncept.
- **Získání všech definic** není tak důležitou funkcionalitou, protože definice samy o sobě nedávají úplnou informaci. Funkcionalita by samozřejmě mohla být implementována. Například pokud by uživatel chtěl získat definice k již nalezeným glosám.
- **Získání všech glos obsahující slovo** je funkcionalita, která zajistí vyhledání glosy ve které se vyskytuje nějaké slovo ať už v definici, názvu nebo některém z tagů.
- **Získání všech glos v textu** – uživatel odešle text a jako odpověď získá všechny glosy, které se v textu vyskytovaly (text obsahoval název glosy). Uživatel může specifikovat jazyk a tagy.

Výhodou *read only* API je, že je jednoduché na implementaci a jednoduché na pochopení. Nevýhodou je, že přidání nebo změna glos musí probíhat jiným způsobem, například přes webový frontend. Případně by glosy musel přidávat správce glosáře. Pro lepší rozšířitelnost aplikace by bylo vhodnější umožnit i vytváření nových glos. Uživatel by například mohl chtít importovat glosy z již vytvořených souborů. Těchto souborů by mohlo být mnoho a bylo by jednodušší je vložit do glosáře skrze API, než aby je musel přidávat správce, nebo je ručně přidávat přes

webový frontend. Navíc správce nemusí být uživateli vůbec znám a nemusí nic vědět o pojmech, které chce uživatel vložit. Tím by mohlo docházek k nesprávnému a pomalému vkládání glos.

Použitelnost aplikace se zvýší přidáním dalších funkcionalit, jako je vytvoření nebo úprava glosy. Důležité je uvědomit si, jaké všechny funkcionality jsou potřebné, například:

- **Vytvoření glosy**: uživatel zadá jméno, abstrakt, text definice, jazyk a tagy a vytvoří se definice a pojem spojené konceptem a jazykem.
- **Vytvoření tagu**: vytvoření nového okruhu kterého se glosa může týkat.
- **Vytvoření jazyka**: přidání jazyka ve kterém mohou být glosy uloženy.
- **Vytvoření glosy v jiném jazyce**: uživatel může vytvořit glosu v jiném jazyce s jiným názvem, definicí a tagy, která popisuje stejnou věc jako jiná glosa. Tyto glosy budou propojeny přes koncept.
- **Smazání glosy, pojmu, definice**. Pokud bude smazána glosa, smaže se pojem i všechny jeho definice. Pokud bude smazán pojem, smažou se i všechny jeho definice.
- **Smazání tagu**: zde jsou dvě možnosti, jak funkcionalitu implementovat. Pokud se smaže tag, mohou se smazat i všechny glosy obsahující tag. Druhá možnost je, že se glosy nesmažou, ale nebudou již mít tento tag. První možnost je vhodná, pokud vznikne potřeba smazat glosy s jedním tagem, avšak na to by bylo vhodnější vytvořit novou operaci. Pokud by totiž po smazání tagu byly smazány všechny glosy, neexistuje rozumný způsob jak smazat tag, pokud již není vyhovující.
- **Smazání jazyka**. Pokud je smazán jazyk jsou smazány i všechny glosy náležící tomuto jazyku.
- **Smazání konceptu**. Pokud je smazán koncept, je smazána i glosa.
- **Úprava glosy** může znamenat úpravu definice, názvu, jazyka nebo tagů.
- **Úprava pojmu**.
- **Úprava definice**.

Zpřístupnění všech předcházejících funkcionalit všem uživatelům může vést k nepatřičnému zacházení s glosářem. Mohou být mazány glosy, které smazány být nemají, nebo naopak přidávány glosy, které neodpovídají využití daného glosáře. Aby bylo možné zabránit nechtěným operacím, je potřeba rozdělit uživatelům práva. Například obyčejný uživatel může pouze číst z API, autorizovaný uživatel může přidávat glosy a případně je mazat. Zde je otázka, zda dovolit autorizovanému uživateli mazat všechny glosy, glosy které sám vytvořil, a nebo glosy, které budou nějak specifikované. Dále zde může být vytvořeno více uživatelů s různými oprávněními. Například obyčejný autorizovaný uživatel může přidávat glosy a mazat svoje glosy, nemůže však vytvořit nový jazyk ani tag. Jelikož je autorizace uživatelů nad rámec této práce, práce se jí již dále nebude zabývat.

3.4 Webový frontend

Webový frontend by měl sloužit jako aplikace pro správu glos a jejich zobrazování. Glosy by mělo být možné vytvořit, upravit a smazat. Stránka na úpravu/vytvoření glosy by měla poskytovat vhodné prostředí na psaní textu pro pohodlné zapsání definice, abstraktu a názvu glosy. Dále by zde měla být možnost přiřadit jazyk a tagy ke glose, případně vytvořit nový tag. Vyhledávání by mělo umožnit filtrovat glosy podle jazyka a tagů. Bylo by vhodné zobrazit náhled vyhledaných glos s názvem, abstraktem, jazykem a tagy. Mělo by být možné si zobrazit kteroukoliv z vyhledaných glos s plným textem.

Webová aplikace by také mohla zajišťovat přihlášení a podle rozdělených práv umožňovat uživatelům přidávat, upravovat a mazat glosy. Například nepřihlášení uživatelé by si mohli glosy pouze zobrazovat. Obyčejní přihlášení uživatelé by mohli vytvářet nové glosy a mazat pouze své glosy. Přihlášení uživatelé s většími právy by mohli upravovat a mazat i glosy, které nevytvořili.

Kapitola 4
Návrh

Kapitola se zaměřuje na návrh aplikace. Popsána bude datová struktura použitá pro uložení glos, výběr a návrh API, postup vyhledávání jednotlivých glos a postup vyhledávání glos v textu. Na konci kapitoly bude stručný návrh webu a popis výběru jazyka a frameworků.

Návrh byl vytvořen tak, aby odpovídal vyspecifikovaným požadavkům, přičemž se návrh zaměřil na vyhledávání glos a rozšířitelnost aplikace. Návrh vychází z předchozí analýzy a pro jednoduchost čtení je i podobně členěn.

4.1 Návrh uložení glosy

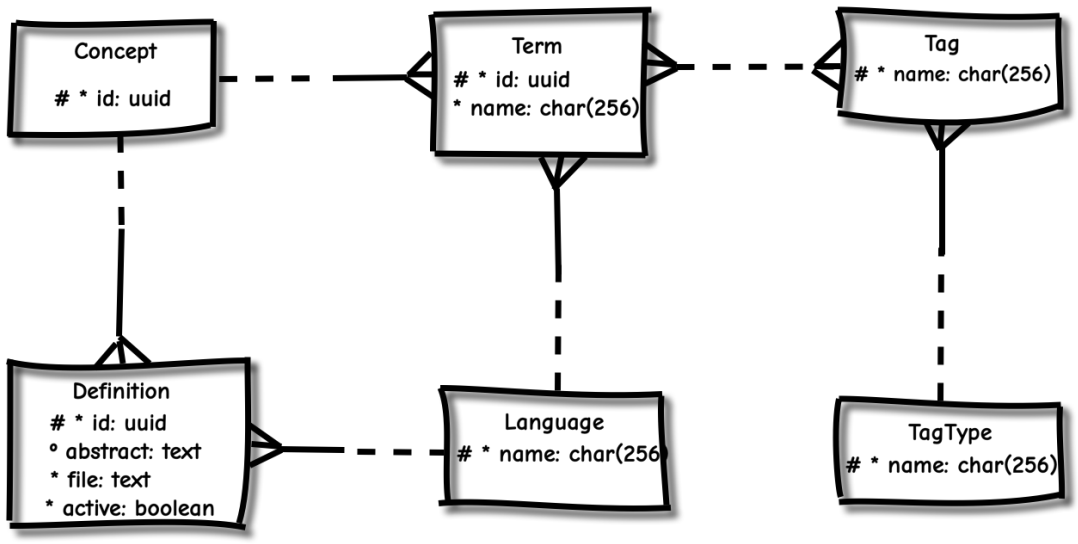
Pro uložení glosy byl navrhnut databázový model, obr. 4.1, který je v této kapitole popisován. Bylo rozhodnuto, že glosa bude uložena jako složení tří entit koncept (`concept`), definice (`definition`) a pojem (`term`). Glosa bude mít oddělenou definici a pojem. Důvodem je snazší doplnění případného rozšíření o funkcionalitu uchovávání historie definic, které v tuto chvíli nebude implementováno. Koncept je důležitý pro to, aby mohly být snadno spojeny glosy se stejným významem, ale v jiném jazyce. Jazyk (`language`) propojuje pojem a definici a společně s konceptem tvoří identifikátor glosy. Pojem může být spojen s tagy (`tag`) a každý tag může mít svůj typ. Typy mohou být použity například k oprávnění zadávání jednotlivých tagů (jakýkoliv uživatel, pouze admin) a nebo třídění tagů (vygenerovány strojově, dokumentace glosáře, přidány uživatelem). To, že má tag typ, je vhodné pro další rozšíření aplikace. Definice bude psána v *Markdown* formátu. Jedná se totiž o formát, který se v poslední době těší velké oblibě, je jednoduchý na psaní a snadno se převádí do HTML, tedy glosy bude snadné zobrazit na webu. U definice bude uchováván atribut *active*, který určí zda se má definice zobrazovat či ne (je zastaralá, není úplná . . .). Jedná se o přípravu na pozdější přidání funkcionality ukládání historie glos.

4.1.1 Entity a jejich vztahy

Term má povinné atributy `id` – identifikátor a `name` – jméno glosy. Musí náležet právě do jednoho konceptu a musí být napsán v právě jednom jazyce. `Term` může mít libovolné množství *tagů*.

Definition má povinné atributy `id` – identifikátor, `file` obsahující text v *Markdown* formátu a `active` určující, zda má být tato definice zobrazována. Dalším atributem je nepovinný `abstract`, který bude zobrazován jako náhled na webu, případně může mít i další využití; měl by obsahovat krátké shrnutí definice. Definice musí náležet právě do jednoho konceptu a musí být napsána v právě jednom jazyce stejně jako `term` (pojem).

Obrázek 4.1 Databázový model



Language má povinný atribut `name`, který je i jeho identifikátorem. Jazyk může být spojen s libovolným počtem *termů* a *definic*.

Concept má povinný atribut `id` – identifikátor a může být spojen s libovolným počtem *termů* a *definic*.

Tag má povinný atribut `name`, který je i jeho identifikátorem. Může být spojen s libovolným počtem *termů* a s právě jedním typem tagu.

TagType má povinný atribut `name`, který je i jeho identifikátorem. Může být spojen s libovolným počtem *tagů*.

4.1.2 Integritní omezení

- pokud je smazán koncept, je smazána definice i pojem
- pokud je smazán jazyk, jsou smazány všechny pojmy a definice v tomto jazyce
- pokud je smazán typ tagu, jsou smazány všechny tagy s tímto typem
- pokud je smazán tag, není smazán žádný pojem
- smazání pojmů a definic nemá vliv na smazání jazyka
- pokud jsou smazány všechny pojmy i definice od jednoho konceptu, je smazán i koncept
- pokud je smazán pojem, jsou smazány všechny definice náležící do stejného konceptu se stejným jazykem
- aby mohla být uložena definice, je potřeba aby existoval pojem (*term*) ve stejném konceptu a se stejným jazykem.
- pojem (*term*) nesmí mít stejný jazyk a zároveň koncept jako jiný pojem
- pokud má pojem stejný název (*name*) a jazyk jako jiný pojem, musí mít tagy které jsou rozdílné

Výpis kódu 4.1 Glosa

```
type GlossType {
  term: TermType
  definition: [DefinitionType]
}
```

4.2 API

S ohledem na probrané funkcionality API v kapitole 3.3 je potřeba rozhodnout, jaký typ API je nejvhodnější. V kapitole 2.2 byly probrány tři typy: SOAP, REST a GraphQL. SOAP je masivní webová služba, která je vhodná pro větší projekty s důrazem na zabezpečení. Zabezpečení není pro glosář prioritou, důležitější je, aby byla data předávána dostatečně rychle a v menším a čitelnějším formátu než je XML. Dotaz na glosy může být totiž častý a zároveň může obsahovat mnoho dat.

Jelikož byl SOAP, kvůli jeho masivnosti, jako adept na API vyřazen, zbývá rozhodnutí mezi RESTem a GraphQL. Výhoda GraphQL je, že si uživatel může zvolit, jaká data bude chtít přijímat. Glosář sice nemá mnoho entit, ale má mezi nimi vysokou provázanost, proto by prezentovat výsledky vyhledávání lépe zvládlo GraphQL. Dotaz na glosu by mohl obsahovat definici, jazyk, pojem i všechny tagy a nemuselo by být posíláno několik dotazů. Na druhou stranu REST je pro uživatele lépe pochopitelný a navíc používá HTTP status kódy, které přesně označují výsledky dotazů. GraphQL vrací status kód 200 (tedy OK), pokud se vyskytla chyba na straně klienta. Informaci o erroru přenáší v těle zprávy. Při vytváření glosy se může stát, že uživatel zadá špatný jazyk a glosa se nevytvoří, informaci o tom ale ponese tělo zprávy a status kód zůstane 200 OK, jako by bylo vše v pořádku.

Bylo rozhodnuto, že pro návrh API a následnou implementaci bude použit jazyk GraphQL. Důvody pro výběr GraphQL byly následující. GraphQL je navrženo tak, aby výsledné API bylo snadno rozšířitelné, pokud by bylo nutné přidat nové funkcionality a nebo změnit schéma (přidání entit, atributů, vztahů). Glosář bude využíván hlavně pro vyhledávání pojmů. Pro uživatele nemusí být všechny atributy glosy přínosné, přičemž GraphQL umožní uživateli vybrat si pouze data, která potřebuje. V neposlední řadě má glosář velkou provázanost mezi entitami a na práci s nimi se spíše hodí GraphQL než REST.

GraphQL typy GraphQL schema se skládá z objektů, *query* a *mutation*. Nejprve byly navrženy typy objektů podle databázového modelu. Glosa - `GlossType` 4.1 se skládá z pojmu (*term*) a definice (*definition*). `TermType` 4.2 představuje název glosy, proto obsahuje atribut *name*, dále obsahuje pole tagů. Spojení *Term - Definition* je přes stejný jazyk (*language*) a koncept (*concept*), oba tyto atributy náleží typu `TermType` i `DefinitionType`. `DefinitionType` 4.3 obsahuje navíc atributy *file* pro text v Markdownu a nepovinný *abstract*.

Výpis kódu 4.2 Pojem - název glosy

```
type TermType {
  id: ID!
  name: String!
  language: LanguageType!
  concept: ConceptType!
  tags: [TagType!]
}
```

Výpis kódu 4.3 Definice

```
type DefinitionType {
  id: ID!
  abstract: String
  file: String!
  concept: ConceptType!
  language: LanguageType!
}
```

Jazyk je definován typem `LanguageType` 4.4, který obsahuje jméno (typu *enum* 4.5), pole pojmů (*termSet*) a pole definic (*definitionSet*), které jsou v daném jazyku napsány.

Výpis kódu 4.4 Jazyk

```
type LanguageType {
  name: GlossaryLanguageNameChoices!
  termSet: [TermType]!
  definitionSet: [DefinitionType]!
}
```

Výpis kódu 4.5 Enum jazyka

```
enum GlossaryLanguageNameChoices {
  CZ,
  EN
}
```

Bylo navrženo, že každý tag má nějaký typ, proto `TagType` 4.7 obsahuje jméno (*name*) a nešťastně pojmenovaný atribut *tagType*, který značí typ tagu, zatímco `TagType` značí GraphQL typ *Tag*. `TagTypeType` 4.6 reprezentuje typ tagu určený jménem (*name*). `TagTypeType` obsahuje pole všech tagů, které mají daný typ.

Výpis kódu 4.6 Typ tagu

```
type TagTypeType {
  name: String!
  tagSet: [TagType]!
}
```

Výpis kódu 4.7 Tag

```
type TagType {
  name: String!
  tagType: TagTypeType!
}
```

Koncept zastřešuje ID glosy, proto má `ConceptType` 4.8 pouze *id* a dále pole všech pojmů a definic, které náležejí danému konceptu.

Výpis kódu 4.8 Koncept

```
type ConceptType {
  id: ID!
  termSet: [TermType]!
  definitionSet: [DefinitionType]!
}
```

Schéma GraphQL API obsahuje *queries* reprezentující dotazy, které nemění objekty a pouze vracejí odpověď a *mutations*, které vytvářejí, upravují nebo mažou objekty.

Bylo navrženo, že všechny objekty, až na koncept, by měly být přístupné skrze API. Výpis všech glos je základní funkcionalitou glosáře, stejně tak přístup k jednotlivým glosám. Vypsání všech pojmů a definic je možné přímo přes vypsání všech glos, přesto byly tyto funkcionality zachovány i samostatně. Přístup ke specifické definici nebo pojmu je možný přes ID daného objektu. API by mělo umožnit vypsat všechny dostupné jazyky, tagy a typy tagů a zajistit přístup k jednotlivým objektům.

Queries pro přístup ke všem objektům daného typu jsou následující: *definitions*, *tagTypes*, *tags*, *languages*, *terms*, *glosses*, které po řadě vracejí všechny objekty definic, typů tagu, tagů, jazyků, pojmů a glos (definice + pojem). Pro přístup k jednotlivému objektu daného typu byly navrženy následující *queries* (po jménu query následují argumenty v závorce): *definition(id: ID!)*, *tagType(name: String!)*, *tag(name: String!)*, *language(name: String!)*, *term(id: ID!)*, *gloss(language: String!, concept: Int!)*

Pro query *glosses* je možné specifikovat, jaké glosy mají být vráceny, pomocí argumentů.

Glosy mohou být filtrovány přes jazyk, koncept, tagy a název. Uživatel se může rozhodnout, zda glosa musí splnit všechny dané parametry nebo alespoň jeden, pomocí atributu *explicit*. Příklad 4.9 ukazuje dotaz na všechny glosy, které splňují alespoň jednu z následujících vlastností: jsou v českém jazyce, v jejich názvu se vyskytuje *glo*, jeden z tagů je *doc* a nebo náleží konceptu 2.

Výpis kódu 4.9 Query glosses

```
query{
  glosses(language: "cz", name: "glo",
    tags: ["doc"]
    concept: 2, explicit: false){
    term{
      name,
      tags{
        name
      }
    }
  }
}
```

Další specifickou `query` je `find`, která přijímá argumenty `term`, specifikující hledaný výraz; `exact` určující zda se má vyhledat přesná shoda nebo cokoliv obsahující daný `term`; `lemmatize`, které říká zda se má vyhledávat za pomoci normalizace. Výsledkem jsou všechny glosy, ve kterých byl hledaný termín přítomen.

Velmi důležitou query je `findIn`, která vrací všechny glosy, které byly vyhledány v daném textu. Text je předán jako parametr `query` společně s případnou specifikací jazyka a tagů. Vyhledávání glos v textu bude vysvětleno v podkapitole 4.3.

V analýze byla zmíněna funkcionalita získání historie ke glose. Jelikož bylo rozhodnuto pro tuto práci historii neuvažovat, nebude tato funkcionalita přes API zpřístupněná.

Dále byly navrhunty `mutations`, které obsahují vytvoření, smazání a úpravu konceptu, jazyka, tagu, typu tagu, definice, pojmu a glosy. Vytvoření glosy vyžaduje získání všech atributů patřících pojmu i definici (koncept, jazyk, název, text definice, abstrakt a tagy). Zatímco pro ostatní objekty stačí pro smazání pouze jejich ID, pro smazání glosy je třeba zadat jazyk a koncept.

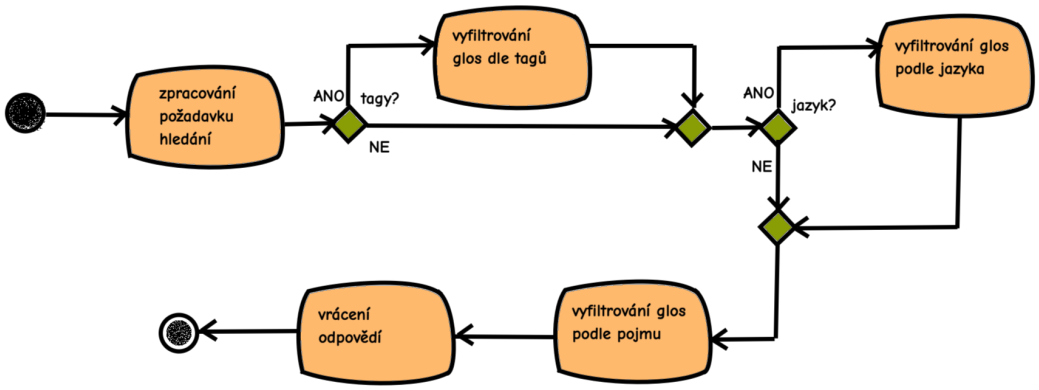
4.3 Vyhledávání glos

Vyhledání glos lze rozdělit do tří kategorií: vyhledání glosy pomocí filtrů, vyhledání glosy podle určitého slova a vyhledání glos v textu. Vyhledání glosy pomocí filtrů bude umožněno pomocí API voláním query *glosses* nebo pomocí vyhledávání na webu. Uživatel zadá nějaké slovo, které chce vyhledat (například *myš*) a může specifikovat tagy (například *zvíře*) a případně jazyk (například *cz*), ve kterém má být glosa vyhledána. Aplikace prohledá pouze názvy pojmů, přičemž vyfiltruje výsledky podle tagů a jazyka, pokud je uživatel zadal. Nebude hledána přesná shoda, ale všechny pojmy obsahující zadané slovo. Funkcionalita byla navržena stejně, jako je tomu u konvenčních vyhledávačů, aby byla dostatečně intuitivní. Možné *Workflow* funkcionality reprezentuje diagram 4.2.

Druhou kategorií je vyhledávání glos, které obsahují určité slovo ve jménu, definici nebo mezi tagy. Tato funkcionalita je potřebná pro vyhledání všech glos, ve kterých je zmíněno nějaké slovo, a bude prováděna pomocí query *find*. U slova bude možné určit, zda je hledána přesná shoda. Příklad: hledaný výraz musí obsahovat slovo *myš* – pokud by byla nutná přesná shoda, pojem se jménem *myška* by nebyl vyhledán.

Slovo zadané uživatelem může být vhodné normalizovat a až poté vyhledávat. Pro normalizaci byla navržena lemmatizace, která však potřebuje ke svému správnému fungování znát jazyk. Zde jsou dvě možnosti: buď může jazyk zadat uživatel a nebo bude slovo lemmatizováno pro všechny jazyky daného glosáře. Dále se bude vyhledávat pouze lemmatizovaná forma ve všech

Obrázek 4.2 Vyhledávání dle filtrů

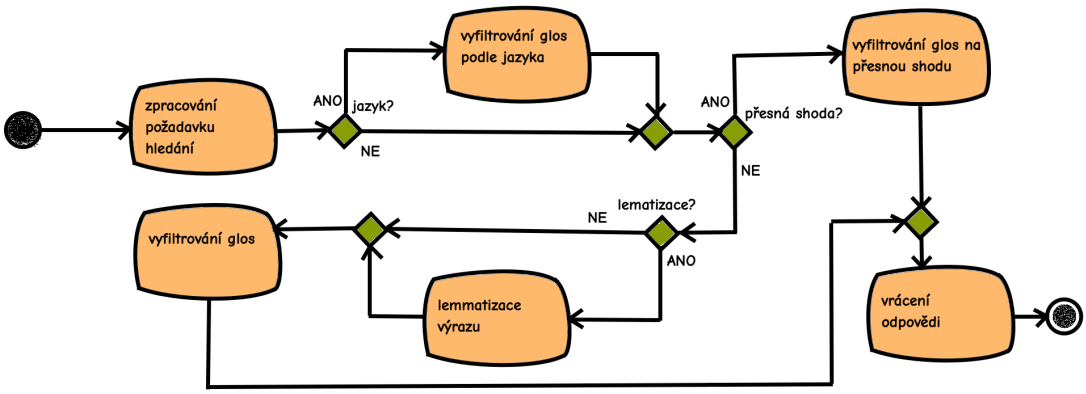


glosách daného jazyka. Pokud uživatel zadá, že chce přesnou shodu, lemmatizace samozřejmě nebude provedena. Mohlo by se stát, že by o lemmatizaci uživatel nestál a tak může být vhodné ponechat uživateli volbu použití lemmatizace. Vyhledat zda se slovo nevyskytuje v tagu nebo názvu glosy je snadné. Pokud by uživatel požadoval normalizaci, slova bylo by vhodné vyhledávat v i v normalizovaných názvech glos.

Vyhledání v definicích je složitější. Pro přesnou shodu je vyhledání snadné: v textu definice se vyhledá přesná shoda bez rozlišení velkých a malých písmen. Pro obyčejné vyhledání je možné použít předzpracovanou definici a vyhledat pouze v předzpracovaných datech anebo vyhledat slovo v celé definici. I přesto, že je vyhledávání v celé definici časově více náročné, bylo rozhodnuto, že pro účely této práce stačí implementace tímto způsobem. Vyhledání v předzpracované definici by bylo rychlejší, pro správnou a precizní funkčnost by byla však potřeba další analýza. Analýza i implementace by byla časově náročná nad rámec této práce, a proto bude vhodné ponechat rozšíření o tuto funkcionalitu na budoucnost.

Uživatel tedy zadá hledaný výraz a případně jazyk ve kterém má být vyhledán, zda má být výraz lemmatizován anebo zda se hledá přesná shoda. Možné *Workflow* funkcionality je předvedeno na obrázku 4.3.

Obrázek 4.3 Vyhledávání výrazu



Poslední kategorií je vyhledání slov v textu. Text je potřeba rozdělit na slova nebo slovní spojení a poté slova spojit s glosami v glosáři. Pro správné spojení je třeba slova normalizovat.

Jakým způsobem lze slova normalizovat, bylo popsáno v kapitole 2.3. Pro normalizaci textu bude vhodnější lemmatizér, protože vezme celý text a pomocí kontextu znormalizuje slova do správných základních forem. Aby toto mohl lemmatizér provést, je potřeba znát jazyk, ve kterém je text napsán. Jazyk zadá uživatel při volání funkce na vyhledání glos v textu. Pokud uživatel jazyk nezadá, nezbývá jiná možnost, než text nenormalizovat a pouze ho rozdělit po slovech. Výsledná rozdělená slova, ať už znormalizovaná nebo ne, mohou být vyhledána v glosáři.

Zbývá vyřešit, jak vyhledat slova v glosáři. Nejjednodušší je spojit slovo s glosou, která má přesně stejný název. Pokud uživatel zadá jazyk, bude se samozřejmě vyhledávat pouze v glosách daného jazyka. Problémem může být, že název nemusí být normalizovaný a kvůli tomu se nespojí s hledaným slovem; na to je jednoduché řešení, název glosy při jejím ukládání normalizovat. Samozřejmě je nutné uchovávat obě formy názvu.

Následkem normalizace přichází další problém a to, zda vyhledávat vždy v normalizovaných názvech a nebo nejprve v nenormalizovaných a poté až v normalizovaných. Vyhledávat nadvakrát může zpomalit celý proces vyhledávání, na druhou stranu vyhledávat pouze v normalizovaných názvech může vést k nepřesnosti, protože více slov může být normalizováno na stejný tvar. Jelikož bude ale sám text normalizovaný, je logičtější vyhledávat v názvech normalizovaných. Je důležité si uvědomit, že ne všechna slova bude schopný lemmatizér znormalizovat. Schopnost zlemmatizování slov se liší pro každý jazyk, pro anglický jazyk bude mít lemmatizace lepší výsledky než pro jazyk český. Pro vyhledávání v anglickém textu je lepší normalizovat text a poté vyhledávat v normalizovaných názvech. Pro vyhledání v českém textu je výhodnější normalizovat text, vyhledat nejprve v nenormalizovaných názvech a následně v normalizovaných, nenalezne-li se shoda.

Přirozeně vyvstává myšlenka, že ani slova v textu nemusí být vhodné ihned normalizovat, protože tím můžeme přijít o spojení slova a glosy které mají specifický tvar, ale po normalizaci odpovídají více glosám. Pokud by ani text nebyl nejprve normalizovaný, čas vyhledávání by se ještě více prodloužil. Rozhodnutí zda text normalizovat, nebo nejprve prohledat a poté normalizovat, závisí na tom, zda je důležitější vyhledat slova rychle a nebo jich vyhledat co nejvíc. Rychlé vyhledávání je na úkor toho, že některá slova nenaleznou svůj protějšek, i když je v glosáři uložený. Vyhledání co nejvíce slov je zase výrazně pomalejší.

Další možností je nespojovat slova pouze s názvem glosy, ale také nějakým způsobem s definicí. Jedním ze způsobů, jak spojit slova s definicí, je vybrání důležitých nebo často se vyskytujících slov v definici. To může být zajištěno pomocí lemmatizace definice při ukládání a následného vybrání nejčastějších slov - *strojových tagů*. Přičemž by slova měla být omezena například délkou (aby se nestávalo, že nejčastějšími slovy, které definují definici jsou předložky, spojky či členy). Pro vyhledávané slovo by se aplikace pokoušela nalézt i spojení se *strojovými tagy*. Tento proces by mohl být zdlouhavý a nepřesný, ale mohl by být použit pro návrh tagů uživateli.

Tagy jsou také důležitou součástí vyhledávání. Pokud by chtěl uživatel zajistit, aby se vyhledávaly glosy jen z daného oboru, mohl by specifikovat tag nebo tagy kterých se mají glosy týkat. Slova by se následně spojovala pouze s glosami, které by obsahovaly správný tag.

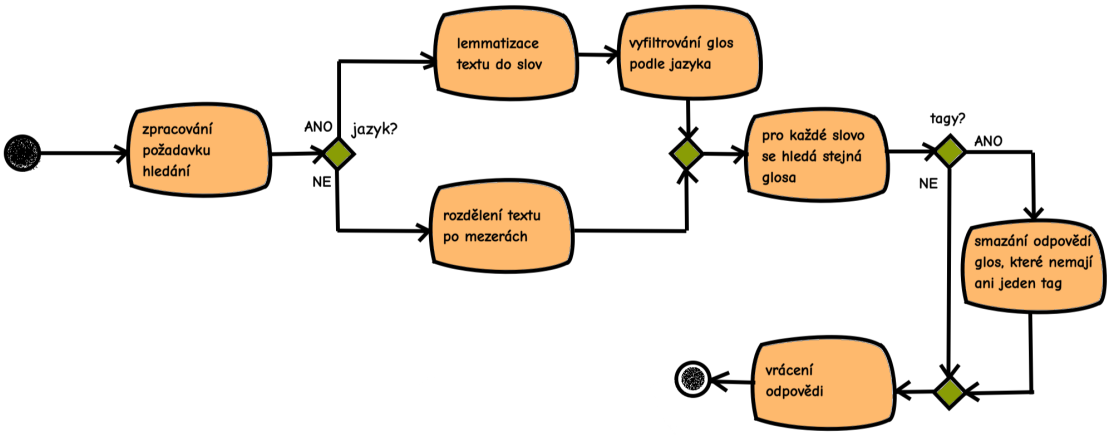
Na vyhledání slov v textu bude využita query *findIn*, která pro daný text nalezne všechny glosy, které se v něm vyskytují. Aby v textu byly vyhledány glosy ve správném jazyce a ohledně správného tématu, bude možné v query zadat jazyk a tagy, kterých by se měla glosa týkat.

Text bude třeba nejprve rozdělit na slova. Pokud bude zadán jazyk, text bude lemmatizován a rozdělen na normalizovaná slova, v opačném případě bude text rozdělen po mezerách. Následně bude pro každé slovo hledána glosa a to propojením pojmu a slova. Pokud budou slova lemmatizovaná, budou se spojovat s jejich lemmatizovanými protějšky ve správném jazyce. Pokud slovo nebude normalizované, bude se hledat shoda s nenormalizovanými termíny. Spojování pouze s lemmatizovanými pojmy bylo vybráno, jako v tuto chvíli, nejlepší řešení, které balancuje mezi rychlostí a přesností.

Pokud byly zvoleny tagy, slova budou vybrána tak, aby odpovídala alespoň jednomu tagu. Tedy pokud text bude obsahovat slovo *myš* a tagem pro vyhledávání bude *IT*, bude glosa pojednávat o myši připojené k počítači a ne o zvířeti. V odpovědi budou všechny nalezené glosy

a slova, pro které byly nalezeny. Možné *Workflow* vyhledání v textu je předvedeno na obrázku 4.4

Obrázek 4.4 Vyhledávání v textu



4.4 Webový frontend

Byli navrženy tři stránky pro webový frontend. Jedna pro zobrazení glos a vyhledávání, jedna pro vytvoření a úpravu glosy a jedna pro bližší náhled glosy.

Hlavní stránka bude rozdělena na polovinu. První polovina stránky bude rozdělena na dvě části - vyhledávání a výsledky. Vyhledávání se bude skládat z vyhledávacího panelu a filtrů. Filtrovat může uživatel podle jazyka a podle tagů, přičemž si může zvolit zda výsledek musí obsahovat všechny tagy a nebo alespoň jeden. Výsledky budou zobrazeny v pravé části levé poloviny stránky. Vždy bude zobrazen název glosy, jazyk, tagy a krátký úryvek definice anebo abstrakt. V pravé části stránky bude zobrazena první z vyhledaných glos, případně glosa, kterou si uživatel vybere z vyhledaných. Glosu bude možné upravit nebo smazat.

Při vytváření glosy zvolí uživatel jazyk glosy a vyplní nepovinné pole pro tagy a abstrakt a povinné pole pro název glosy a text její definice. Text definice bude psán v *Markdownu*, ale pro uživatele, kteří v něm nebudou zdatní, bude zpřístupněn formátovací panel. Pokud neexistuje tag, který by chtěl uživatel přiřadit, může si ho přímo při vytváření glosy vytvořit. Bylo rozhodnuto, že uživatel bude muset nejprve tag vytvořit a až poté přidat (tag se automaticky nevytvoří, pokud byl připsán k vybraným tagům), aby nedocházelo k vytváření tagů omylem.

Po uložení vytvořené glosy se uživateli zobrazí stránka s glosou. V levé části stránky se zobrazí vyhledávací panel s výsledky a v pravé části stránky se zobrazí daná glosa. Na stránku dané glosy se bude možné dostat také z hlavní stránky.

Celý wireframe může být viděn v příloze A.

4.5 Výběr jazyka a frameworků

Pro vývoj aplikace byl vybrán jazyk Python, kvůli jeho jednoduchosti a velké podpoře knihoven (normalizace slov, převod *Markdownu* do HTML, ...). Přičemž byl použit framework Django, který se zaměřuje na vývoj webů.

Pro implementaci GraphQL API byla použita knihovna Graphene respektive Graphene-Django. Graphene pomáhá vytvořit GraphQL schéma pomocí vytváření objektů, queries a mutations. Graphene má svoje vlastní typy, odpovídající GraphQL typům.

Kapitola 5
Implementace a Testování

Kapitola se zaměřuje na implementaci. Nejprve jsou popsány technologie, které jsou využity při vývoji aplikace. Je zde popsáno verzování, vývojové prostředí a nasazení. Dále je popsán postup nasazení aplikace pro vlastní užití. Následuje popis struktury projektu a vysvětlení důležitosti jednotlivých souborů. Ve třetí podkapitole je popsána implementace API společně s dokumentací k API. Následuje popis implementovaného webu a podkapitola zabývající se testováním. Poslední část kapitoly se zaměřuje na vytvořenou funkci využívající glosář a další možnosti rozšíření aplikace. Nejprve popisuje možná vylepšení samotné aplikace a následně popisuje možná rozšíření, využívající aplikaci.

Implementace probíhala v jazyce Python pomocí frameworku *Django*. Bylo implementováno GraphQL API s pomocí knihovny *Graphene-Django*. Webový frontend je psán v HTML s knihovnou *Bootstrap* s použitím jazyka *JavaScript*. Práce byla verzovaná na gitu ve veřejně přístupném repozitáři `https://gitlab.fit.cvut.cz/frankoli/glossary`. Webová aplikace je přístupná přes `http://bc-glossary.herokuapp.com` a webové API je přístupné na `http://bc-glossary.herokuapp.com/graphql` s IDE a na `http://bc-glossary.herokuapp.com/api` bez IDE. Funkce využívající glosář byla psána převážně v jazyce *JavaScript* a je dostupná na `https://gitlab.fit.cvut.cz/frankoli/glossary-function`.

5.1 Technologie

Pro implementaci práce bylo nutné vybrat nástroje, ve kterých se bude aplikace vyvíjet a nástroje, které budou pomáhat při vývoji případně při běhu aplikace.

Verzování Pro verzování práce byl vybrán GitLab - ucelená DevOps platforma určená pro lifecykle vývoje. GitLab umožňuje: verzování projektu pomocí Gitu, Issue tracking a CI/CD (Continuous Integration/Continuous Deployment), které umožní ihned po nahrání na repozitář kód otestovat a následně nasadit. Vývoj práce může probíhat ve větvích (*branch*), což umožňuje oddělení vývoje jednotlivých částí projektu. Oddělení větví zamezuje nechtěnému přepsání kódu jedné vyvíjené funkcionality při vývoji jiné. Spojení větví a tedy i funkcionalit se provádí přes takzvaný *merge*. Při *merge* mohou nastat konflikty v kódu (část kódu byla přepsána v obou větvích), tento konflikt se musí individuálně vyřešit. V tomto projektu byly kromě větve *master* vytvořeny další čtyři větve (*api, web, models, cleanup*).

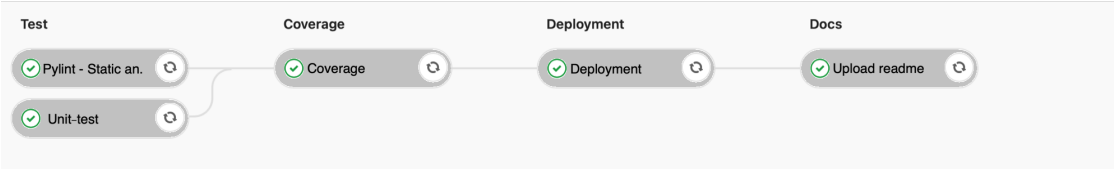
5.1.1 GitLab CI/CD

V této práci byl GitLab používán pro verzování a CI/CD. Aby mohlo být použito CI/CD, je potřeba vytvořit konfigurační soubor *.gitlab-ci.yml* do root (hlavní) složky repozitáře. Tento soubor je ve formátu YAML, což je snadno čitelný jazyk, který je často využíván v konfiguračních souborech. Soubor popisuje, jakým způsobem má být projekt sestaven (*build*). Po tom, co jsou do repozitáře nahrány nové commity (pomocí *git push*) je spuštěná takzvaná *Pipeline*. *Pipeline* je složená ze *Stages*, které jsou dále složené z jednotlivých *Jobs*. Jedna *Stage* může obsahovat více *Jobs*, které se provádějí paralelně. *Stages* představují jednotlivé fáze nasazení. *Jobs* se stará o jednu určitou operaci, která je spuštěná v novém prostředí (*enviroment*). Každý *Job* nám může vyhodit chybu a tím případně zastavit celý běh *Pipeline*.

Dále budou popsány jednotlivé části souboru *.gitlab-ci.yml*, použité pro tento projekt. Na začátku souboru je nejprve specifikováno, jaký *docker image* má být použit při vytváření *Jobs*. Jelikož je práce psaná v Pythonu, je zde použit docker image *Python:latest*. Následuje definice *Stages* – jedná se o *Test*, *Coverage*, *Deployment* a *Docs*.

Každý *Job* potřebuje nainstalovat potřebné knihovny. Seznam potřebných knihoven je v souboru *requirements.txt*. Instalace knihoven je zajištěna blokem `before_script:`, který spustí `-pip install -r requirements.txt` před začátkem každého *Job*. Následuje definice pěti *Jobs*, definice se vždy skládá z názvu *Jobu*, názvu *Stage* ve které se nachází, a skriptu který se má provést. Ve *Stage Test* lze nalézt spouštění testů a statické analýzy (Pylint). Ve *Stage Coverage* je spouštěn skript, který by měl zjistit pokrytí kódu testy. Ve *Stage Deployment* je aplikace nasazena na *Heroku* (cloudová platforma pro sestavování a spouštění aplikací). Poslední *Stage Docs* nahraje do produkční databáze data z README.md a případně další potřebná data. Diagram jedné *Pipeline* je vidět na obrázku 5.1

Obrázek 5.1 Pipeline diagram



Docker *Docker* je nástroj umožňující izolaci aplikací do kontejnerů se všemi knihovnami a soubory potřebnými pro běh dané aplikace. Kontejner běží nezávisle na operačním systému a, pokud není nastaveno jinak, odděleně od ostatních kontejnerů. Na jednom operačním systému může zároveň běžet několik kontejnerů, které sdílí výpočetní sílu počítače. *Dockerfile* obsahuje příkazy, dle kterých je vytvořený *image*. *Image* obsahuje soubory, aplikace, knihovny potřebné pro vytvoření kontejneru. Spuštěním *image* se vytvoří běžící kontejner.

Většina hojně používaných *image* je ukládána na veřejném repozitáři *dockerhub*, ze kterého je možné *image* stáhnout. Díky tomu nemusí vývojář pokaždé psát nový *Dockerfile*. V této práci je z *dockerhub* stahován *image* `Python:latest` využívaný v CI/CD.

Sentry *Sentry* je nástroj pro monitorování aplikace. Odchytává a zaznamenává výjimky a chyby na všech běžících instancích aplikace. Záznamy obsahují informace o chybě, času vzniku události a popis prostředí, na kterém se stala. *Sentry* je v této práci používána, nastavení lze nalézt v *settings.py*.

PyCharm a virtuální prostředí *PyCharm* je vývojové prostředí pro jazyk Python. Toto prostředí umožňuje spouštění, debug a profiling aplikace. Dále umožňuje generování code coverage reportu, práci s dockerem, připojení k databázi atd. Pro správné používání je zásadní

nastavit správnou verzi Pythonu (v tomto projektu 3.9 a výš). Nejčastějším způsobem, jak spravovat Python projekty je pomocí virtuálního prostředí – *Virtual Enviroment*. Virtuální prostředí je čisté prostředí, do kterého mohou být nainstalovány potřebné knihovny a frameworky. Ve virtuálním prostředí je spuštěna aplikace.

Následuje ukázka práce s virtuálním prostředím 5.1. Nejprve je vytvořena složka s prostředím, ve které je následně spuštěn aktivační skript (změní cesty k binárním souborům Pythonu). Na třetím řádku jsou do aktivovaného prostředí instalovány potřebné knihovny. Poté může být aplikace spuštěna. Pro deaktivaci prostředí je použit příkaz *deactivate*. Pro spuštění aplikace glosáře ve virtuálním prostředí je možné použít skript *run-server.sh*.

Výpis kódu 5.1 Instalace virtuálního prostředí

```
python3 -m venv venv_dir
source venv_dir/bin/activate
pip install -r requirements.txt
# run the python app
deactivate
```

5.1.2 Heroku deployment

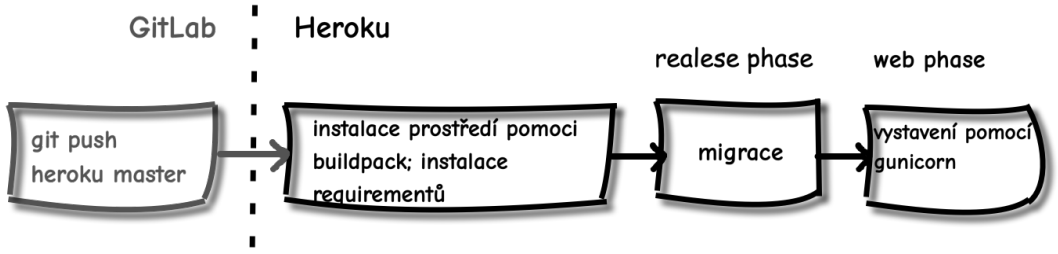
Pro nasazení aplikace bylo využito Heroku. Heroku je cloudová platforma, na které se dají sestavovat a spouštět aplikace a následně s nimi operovat. Aplikace mohou být veřejně přístupné na internetu (pro tuto aplikaci na URL `http://bc-glossary.herokuapp.com/`).

GitLab i Heroku umožňují uchovávat systémové proměnné. K proměnným je možné přistupovat v kódu, který je na GitLabu či Heroku uchováván. Použitím těchto systémových proměnných může být zamezeno úniku citlivých dat. Na GitLabu jsou uchovány proměnné

`HEROKU_API_KEY` a `HEROKU_EMAIL`, které jsou využívány při nasazování na Heroku v předposlední fázi GitLab CI/CD (*Job Deployment*). Na GitLabu se nejprve vytvoří `git remote` na Heroku a následně je na Heroku odeslán (`git push`) adresář shodný s aktuálním *commitem*.

Po odeslání dat na git repozitář Heroku se automaticky pustí Heroku *Pipeline*, která zjistí, v jakém jazyce je aplikace napsána (například hledá manage.py) a podle toho vytvoří odpovídající prostředí, ve kterém bude aplikace běžet. Prostředí jsou vytvářena na základě *buildpacks*. *Buildpack* pro tuto aplikaci nastaví Python prostředí, nainstaluje *requirements.txt* a spustí příkazy specifikované v *Procfile*. *Procfile* obsahuje příkazy pro vytvoření migrací a pro spuštění aplikace přes *gunicorn* (Python HTTP server). *Procfile* může být rozčleněn do fází, v tomto projektu jsou to fáze *release* a *web*. S každým novým nasazením aplikace jsou spuštěny obě fáze, nejdříve *release*, poté *web*. Při vypnutí a následném zapnutí aplikace se spouští pouze fáze *web*. Deployment diagram je na obrázku 5.2 Na Heroku jsou proměnné `DATABASE_PASSWORD`, `DJANGO_DATABASE`, `DJANGO_DEBUG`, které jsou použity v souboru *settings.py* který Heroku využívá při spuštění aplikace. `DATABASE_PASSWORD` značí heslo do produkční databáze, `DJANGO_DATABASE` určuje, která databáze má být použita, `DJANGO_DEBUG` určuje zda aplikace poběží v *debug* módu.

Obrázek 5.2 Heroku deployment



5.2 Nasazení aplikace

Následující text popisuje, jak nasadit a provozovat vlastní instanci aplikace. Návod bude pouze pro systémy UNIX. Ve všech případech je nejprve nutné naklonovat repozitář pomocí příkazu `git clone https://gitlab.fit.cvut.cz/frankoli/glossary`.

Manuální nasazení Do operačního systému je nutné nainstalovat *python3* přes příkazy `apt-get update`, `apt-get install python3.9 python3-venv python3-dev build-essential` a *PostgreSQL* přes příkaz `apt-get install postgresql postgresql-contrib`.

Pro spuštění aplikace stačí spustit předpřipravený skript z hlavní složky *glossary* pomocí příkazu `./run-server.sh`. Skript nejprve ověří, zda je nainstalován Python správné verze a pokud není nainstalováno virtuální prostředí (venv), nainstaluje ho. Následně aktivuje prostředí, nainstaluje do něj potřebné *requirements*, provede migrace databáze a spustí aplikaci.

Docker Druhou možností jak spustit aplikaci je pomocí *Dockeru*, který musí být předem nainstalovaný. V repozitáři je připravený soubor *Dockerfile*. Pomocí příkazu `docker build . -t glossary:latest` se vytvoří *Docker image glossary* a pomocí `docker run glossary` se aplikace spustí.

PyCharm Aplikaci je možné spustit pomocí PyCharm. Po naklonování repozitáře a otevření aplikace v IDE je třeba vytvořit virtuální prostředí (venv) PyCharm → Preferencs → Project:Glossary → Python Interpreter → kolečko nastavení → Add → Virtual Enviroment a následně vytvořit konfiguraci pro *run* pomocí Run → Edit Configurations → `+` → Django Server → vyplnění jména a portu. Aplikace může být spuštěna přes Run → run *jméno konfigurace*.

Nastavení databáze Glosář jako výchozí databázi používá *SQLite – db.sqlite3*, protože je vhodná pro vývoj. Na produkci aplikace využívá databázi *PostgreSQL*, protože se jedná o open-source a je stejně rychlá jako její konkurenti. Pro přidání nové databáze je nutné v *settings.py* do `DATABASE_AVAILABLE` přidat konfiguraci databáze. Při spuštění aplikace je potřeba nastavit systémovou proměnnou `DJANGO_DATABASE` na jméno databáze z `DATABASE_AVAILABLE`.

Logování zaznamená děje v aplikaci jako je vytvoření glosy, změna glosy, chyba při běhu aplikace atd. Logování je nastaveno v souboru *glossary/settings.py*. V tuto chvíli se logují všechny zprávy závažnosti alespoň INFO (existuje 5 typů závažností logů: DEBUG, INFO, WARNING, ERROR, CRITICAL). Pro logování je možné nastavit formát, v jakém se má log zobrazovat, *handler*, který určí, kam se má logovaná zpráva zapsat a filtery, které mohou nastavit závažnost logovaných zpráv pro každý *handler*. Přesná dokumentace je na adrese `https://docs.djangoproject.com/en/4.0/topics/logging/`.

Při spuštění aplikace na produkčním prostředí je nejlepším řešením použít HTTP server *Gunicorn*. Aby byla aplikace spuštěna přes *Gunicorn*, stačí v skriptu *run-server.sh* změnit poslední řádek z `python3 manage.py runserver` na `gunicorn glossary.wsgi`.

5.3 Struktura projektu

Projekt je psán pomocí frameworku Django, který pomáhá při vývoji webů i API. Implementace webu i API je v jednom projektu, což umožňuje sdílet některé soubory mezi těmito aplikacemi (například models.py). V hlavní složce jsou uloženy složky *glossary*, *tests* a *docs* a následující soubory. Do hlavní složky se také generují neverzované soubory jako logy (do verzované složky *logs*), databáze, coverage report atd.

.gitignore soubor specifikující soubory a složky, které má Git ignorovat.

.gitlab-ci.yml konfigurační soubor pro CI/CD, který byl popsán výše.

Deckerfile soubor obsahující příkazy pro vytvoření *image* a následné spuštění aplikace.

Procfile soubor obsahující příkazy (migrace, spuštění aplikace), které jsou spuštěny na *Heroku* po nahrání nové verze.

README.md soubor obsahující informace k projektu (nastavení, jak projekt spustit, jak změnit jazyk, logování ...)

insert_readme.py soubor který obsahuje skript pro vložení README.md do databáze. Databázi také připraví na to, aby mohlo být README.md vloženo (vytvoří tagy, jazyk ...).

manage.py soubor umožňující spravovat Django aplikace.

migrate.sh soubor obsahuje skript, který provede migrace a vloží do databáze *README.md*.

requirements.txt obsahuje všechny potřebné knihovny s verzemi. Aby byly requirements.txt aktuální, stačí spustit `pip freeze > requirements.txt` z hlavní složky.

run-server.sh soubor obsahující skript pro spuštění aplikace.

V adresáři *tests* je vytvořen pouze jediný test, a to na verzi Pythonu. Adresář *logs* je prázdný, a jsou do něj generovány logy při běhu aplikace. Složka *docs* obsahuje konfiguraci pro generování dokumentace. Adresář *glossary* obsahuje složku *api*, ve které je implementace API, adresář *web*, ve kterém je implementace Webu a adresář *local*, ve kterém jsou v tuto chvíli uloženy soubory pomáhající k překladu webu do češtiny. Můžou zde být přidány i jiné jazyky. Složka *glossary* dále obsahuje následující soubory:

glossary/admin.py soubor nastavuje, které modely (objekty) by měly být přístupné adminovi a jak se mu budou jednotlivé modely zobrazovat.

glossary/models.py soubor obsahuje implementaci všech objektů podle databázového modelu a navíc model *user* pro budoucí rozšíření. U modelu *Term* jsou přepsány funkce `save` a `delete` a u modelu *Definition* je přepsána funkce `save`, tak aby implementace odpovídala integritním omezením definovaným v kapitole 4.1.2.

glossary/settings.py soubor obsahuje Django nastavení. Nastavuje se zde *Graphene* (pro psaní GraphQL API), *templates* (pro vytváření webu). Dále je zde nastavená databáze a to jak produkční *PostgreSQL* (na Heroku), tak defaultní a testovací *SQLite*. Mimo jiné je zde nastaveno reportování do *Sentry*, logování a některé konstanty.

glossary/urls.py soubor obsahuje všechny webové, API URLs a URL k dokumentaci.

Další výčet souborů se bude zabývat složkou *glossary/api*, která jednak obsahuje implementaci API a také obsahuje testy k API.

glossary/api/schema.py soubor obsahuje implementaci API.

glossary/api/tests.py soubor obsahuje testy k API, tyto testy využívají takzvané *snapshots*, které jsou uloženy v složce *glossary/api/snapshost* v souboru *snap_tests.py*.

glossary/api/urls.py soubor obsahuje URL potřebné k API. V tomto případě pouze dvě: *graphql* se zapnutým IDE a *api* bez IDE.

Implementace webu je ve složce *glossary/web*, která obsahuje adresáře *static*, *templates* a *templatetags* a další soubory, mimo jiné soubor *urls.py* obsahující všechny webové URL této aplikace.

glossary/web/static adresář obsahuje soubor *style.css*, který definuje css styly. Dále obsahuje složku *img*, ve které je obrázek *favicon.ico* používaný pro webovou stránku jako ikonka v prohlížeči. Také obsahuje adresář *js*, ve kterém jsou funkce použité při implementaci webu.

glossary/web/templatetags/markdown_extras.py soubor obsahuje funkci, která převede *Markdown* do HTML.

glossary/web/templatetags/objects_access.py soubor obsahuje funkce, které přistupují k modelům nebo s nimi operují. Tyto funkce jsou využívané v *templates* například pro načtení abstraktu.

glossary/web/templates složka obsahuje složku *glossary*, která je rozdělená do dvou podsložek: *pages* obsahující stránky a *components* obsahující různá tlačítka ve složce *buttons*, různá pole ve složce *fields*, řádek tagů v souboru *tag_row.html*, vyheldávací panel v souboru *search_bar.html*, hlavičku glosy v *term_row.html* a alert v souboru *alert.html*. Složka *templates* obsahuje adresář *docs*, ve kterém se nachází *template* pro zobrazování dokumentace.

glossary/web/.../pages/base.html soubor obsahuje základní hlavičku společnou pro všechny stránky.

glossary/web/.../pages/main.html soubor představuje hlavní stránku složenou z různých komponent.

glossary/web/.../pages/editor.html soubor obsahuje kód stránky pro vytvoření a úpravu glos.

glossary/web/.../pages/error_page.html představuje stránku, která je zobrazena po chybě.

glossary/web/.../pages/gloss_page.html představuje stránku glosy.

glossary/web/.../components/buttons složka obsahuje čtyři soubory, editovací tlačítko (s obrázkem tužky), tlačítko pro smazání (s obrázkem koše) a kombinaci těchto tlačítek. Poslední soubor je tlačítko, u kterého lze parametry doplnit (jméno a kam odkazuje).

glossary/web/.../components/fileds složka obsahuje políčka pro vložení abstraktu a tagu, komponentu pro vytvoření tagu a komponenty pro výběr jazyka.

5.4 Implementace API

API je psáno pomocí jazyka GraphQL, proto je využívána knihovna *Graphene-Django* vybudovaná na knihovně *Graphene*. Díky knihovně je snadné přidávat GraphQL funkcionality. API má implementováno 21 *mutations* a 15 *queries*. *Mutation* pro vytvoření objektů: `createConcept`, `createLanguage`, `createTagType`, `createTag`, `createDefinition`, `createTerm`, `createGloss`. *Mutation* pro úpravu objektů: `updateTagType`, `updateTag`, `updateDefinition`, `updateTerm`, `updateGloss`. *Mutation* pro smazání objektů: `deleteConcept`, `deleteLanguage`, `deleteTagType`, `deleteTag`, `deleteDefinition`, `deleteTerm`, `deleteGloss`.

Protože je možné nejprve vytvořit koncept a následně k němu přidávat pojem a jeho definici mohlo by nastávat, že v databázi budou uloženy nevyužité koncepty. Pro odstranění těchto konceptů je implementována *mutation* `deleteUnusedConcepts`.

Pro budoucí rozšíření bylo do modelů implementováno generování automatických tagů k definici. Tyto tagy mohou být následně využity pro vyhledávání nebo jako nápověda tagů uživateli. Generování tagů nejprve lemmatizuje definici a následně vybere nějaký počet nejčastějších slov (počet je nastaven v *settings.py*), aby se co nejvíce zamezilo vybírání předložek a spojek, je výběr slov omezen jejich délkou. Pokud by byla změněna funkce pro generování tagů, je třeba přegenerovat všechny již vytvořené tagy pro všechny definice. To zajišťuje *mutation* `updateDefinitionTags`.

Pro přístup k objektům byly vytvořeny *queries*. Pokud chce uživatel získat všechny objekty daného typu, použije tyto *queries*: `terms`, `languages`, `tags`, `tagTypes`, `definitions`. Pro přístup k jednotlivému objektu přes identifikátor jsou následující *queries*: `term(id)`, `language(name)`, `tag(name)`, `tagType(name)`, `definition(id)`. Pro získání všech glos (pojem + definice) je možné využít *query* `glosses`. Tato *query* umožňuje definovat glosy podle jazyka, konceptu, názvu a tagu, přičemž je možné určit, jestli glosa musí obsahovat všechny tagy nebo alespoň jeden (`glosses(language:"cz", concept:1, name:"pes", tags:["animal"], explicit:true)`).

Pro získání jedné glosy se používá *query* `gloss(language, concept)`, kde musí být vyplněn jak jazyk, tak koncept, protože je jím glosa definovaná. Pro vyhledání pojmu je možné použít *query* `find` a nebo *query* `termByName`. *Query* `find` přijímá vyhledávaný pojem a vrací glosy, uživatel může určit, jestli si přeje přesnou shodu a jestli chce pojem lematizovat, případně v jakém jazyce. *Query* `termByName` naopak vrací pouze pojem (term) a uživatel může zadat název, nebo čím má název začínat, případně co má obsahovat. Pro vyhledávání glos v textu slouží *query* `findIn`, která přijímá text a případně tagy a jazyk; vrací glosy společně se slovem, se kterým byla glosa spojena.

Dokumentace Dokumentace k API je přístupná přímo v GraphiQL IDE na URL */graphql*, popřípadě `http://bc-glossary.herokuapp.com/graphql`. Dokumentace k API je psána v kódu, a to u definice každé *query* v poli `description=` a nebo na začátku definice *mutation* třídy.

Dokumentace ke kódu je psaná pomocí *docstring* přímo v implementaci a po spuštění aplikace je přístupná na `http://bc-glossary.herokuapp.com/docs`. Dokumentace je generovaná pomocí nástroje *Sphinx*. Ručně je možné ji vygenerovat pomocí příkazu `make html` ze složky *docs*; výsledek bude uložen ve složce *docs/_build/html*. Dokumentace je aktualizována při každém nasazení na Heroku.

5.5 Implementace webu

Webový frontend byl implementován pomoci frameworku Django, který usnadnil přístup k modelům v databázi a k funkcím uložených ve složce *glossary/web/templatetags*. Web je připraven na to, aby bylo snadné přeložit texty do požadovaného jazyka. V tuto chvíli je připraven pouze překlad do češtiny. Pro překlad do jiného jazyka je třeba nejprve spustit příkaz `django-admin makemessages -l <LANG> -i venv` přičemž `<LANG>` značí kód jazyka, do kterého má být přeložen

(en, cs, es ...). Ve složce *glossary/local* bude vytvořena nová složka se stejným jménem, jako je kód jazyka. V této složce bude vytvořen soubor *django.po*, ve kterém je nutné přeložit texty. Následně stačí spustit `django-admin compilemessages` aby se překlady propsaly. Web je přístupný na `http://bc-glossary.herokuapp.com`.

Hlavní stránka je rozdělena na dvě části. V levé části je možné vyhledat glosy a v pravé části je vždy jedna glosa zobrazena. Pravá část se vždy skládá z názvu glosy, jejích tagů a její definice. V horním pravém rohu jsou dvě tlačítka – tužtička a koš. Tužtička značí editovací tlačítko, po stisknutí se objeví editor glosy. Po kliknutí na tlačítko s obrázkem koše je glosa smazána. Po kliknutí na definici nebo název glosy je zobrazen větší náhled dané glosy.

V levé části je možné vyhledávat přes zadání slova do vyhledávacího panelu. Dále je možné určit tagy, podle kterých mají být glosy vyhledány, přičemž je možné nastavit, zda musí glosy obsahovat všechny tagy, nebo alespoň jeden, pomocí přepínacího tlačítka. Jazyk glosy je možné zvolit pomocí zakliknutí jazykového tlačítka. Po stisknutí tlačítka *search* se zobrazí náhledy vyhledaných glos se jménem, jazykem, tagy a úryvkem definice. Po kliknutí na jakoukoli glosu bude tato glosa zobrazena v pravé části. Levá část dále obsahuje tlačítko *Main Page*, odkazující na hlavní stránku bez zadaných vyhledávacích filtrů, a tlačítko *Add New* odkazující na editor glosy.

Stránka glosy obsahuje v levé části seznam glos a vyhledávací panel spolu s tlačítky *Main Page* a *Add New*. Na pravé straně je samotná glosa, podobně rozložená, jako je tomu na hlavní stránce.

Editor slouží k vytvoření nebo editování glosy. Pokud je vytvářená nová glosa přes tlačítko *Add New*, stránka není předvyplněná, v případě upravování glosy přes tlačítko tužtičky jsou vyplněny již uložené informace o glose. V pravé části stránky je nutné vyplnit název glosy a její definici, která se vyplňuje do *Markdown* editoru. Každá glosa musí mít zvolený jazyk, který lze změnit překliknutím na ikonku jiného jazyka v levé horní části stránky. Dále je možné přidat a případně vytvořit tagy. Pro vytvoření tagu je nutné zvolit název a poté tisknout tlačítko *Create*, následně může být tag přidán mezi tagy přidělené ke glose. Pro přidání abstraktu je třeba využít pole v levé dolní části stránky. Pro uložení glosy je třeba stisknout tlačítko *Save* v pravém dolním rohu. Pokud chce uživatel změny zahodit, musí stisknout tlačítko *Main Page*, které ho odkáže na hlavní stránku.

5.6 Testování

Testování je důležitou součásti softwarového vývoje, protože může odhalit chyby nebo nedostatky v programu. Testování je možné rozdělit na statické a dynamické testování. Statické testování nevyžaduje běh programu, v této práci je z této kategorie prováděna statická analýza kódu. Dynamické testy vyžadují běh programu, v této práci se jedná o testování API a modelů. Další rozdělení testů může být *black box*, tedy že tester nezná kód a testuje pouze vstupy a výstupy, nebo *white box*, kdy tester zná kód a testuje vnitřní funkcionality. Testy je dále možné rozdělit na manuální, které provádí člověk a automatické, které jsou napsané programátorem a vykonává je tedy software. V této práci byly testy především automatické, ale při vývoji webového frontendu byl frontend manuálně testován. Testy je možné rozdělit na *unit testy* (například testování metod), *testy funkcionalit*, *integrační testy* a další. Probrání všech testovacích metod není cílem této práce a proto zde nejsou uvedeny všechny.

API a modely bylo otestovány po jejich částečné implementaci, přičemž testy objevily některé chyby, které byly následně opraveny. Statická analýza kódu byla průběžně spouštěna a nalezené závažné chyby byly opraveny. Testy jsou spouštěny v každé *Pipeline* na GitLabu. Pro spuštění testů lokálně je nutné provést příkaz `python manage.py test`, který spustí všechny testy.

Testování API U API jsou otestovány *mutations* pro vytvoření, upravení a smazání jednotlivých objektů. Dále jsou otestovány funkcionality *termByname, glosses, find* a *findIn*. Testování je provedeno pomocí knihoven *snapshottest*, *django.test.testcases* a *from graphene.test*. *Snapshottest* umožňuje zápis očekávaných výsledků do souboru (*glossary/api/snapshost/snap_tests.py*). Záznamy ze souboru jsou následně porovnávány s výsledky testu. Očekávaný výsledek testu se zapíše do souboru *snap_tests.py* po prvním proběhnutí testu. Pokud je nutné aktualizovat *snapshots*, musí se spustit testy s přepínačem pro aktualizaci `python manage.py test --snapshot-update`, příkaz přepíše výsledky v souboru *snap_test.py*.

Testování modelů Jelikož jsou přepsány metody *delete* u modelů *Term* a *Definition* a metoda *save* u modelu *Term*, musely být tyto metody otestovány. Testování je provedeno pomocí knihovny *django.test.testcase*.

Statická analýza kódu je prováděna pro ověření, zda kód odpovídá určitým standardům. Statická analýza je prováděna v první *Stage* v *Pipeline* pomocí softwaru *Pylint*. Software kontroluje, jestli kód odpovídá standartu PEP8 (best practices pro psaní Python kódu), neobsahuje syntaktické chyby, neobsahuje code smell (duplikující se kód, magická čísla, ...), je dostatečně dokumentován atd. Zpráva o analýze obsahuje skóre (čím vyšší, tím lepší); pro tuto práci je považováno za dostatečně dobré, pokud je skóre větší než 7.

Pokrytí kódu Pokrytí kódu testy nebo také *code coverage* je metrika, která určuje kolik procent kódu je otestováno. Většinou je spočítána z počtu řádků, které byly navštíveny při testování. Přičemž se také může měřit, jestli test prošel všechny větve (if else). V tomto projektu je zjištění pokrytí kódu spuštěno při každém běhu *Pipeline*, pokud testy proběhly úspěšně. Lokálně může být spuštěna pomocí `coverage run --source=./ -m manage test` a následného spuštění *coverage report*. Nejprve jsou tedy spuštěny testy a je zaznamenáno, jaké řádky kódu byly navštíveny, následně je výsledek vypsán do konzole.

5.7 Funkce do webové stránky

Pro demonstraci využití glosáře je implementována funkce v jazyku *JavaScript*, dostupná na `https://gitlab.fit.cvut.cz/frankoli/glossary-function`. Přidá-li se funkce do webové stránky, vyznačí v jejím textu slova, která nalezla v glosáři, spolu s jejich významem.

Funkce přijímá nepovinné parametry představující odkaz na glosář, jazyk ve kterém mají být glosy vyhledány, a tagy, z nichž alespoň jeden musí vyhledané glosy obsahovat. Funkce prochází webovou stránku a postupně vždy vybere text ohraničený HTML tagy `<p></p>` - paragraf. Text paragrafu odešle v `query findIn` do aplikace glosáře a zpět dostane vyhledaná slova, která se vyskytla v textu, spolu s jejich významy. Funkce následně projde paragraf, nalezená slova barevně vyznačí a navíc ke každému takto vyznačenému slovu doplní definici a odkaz do glosáře. Definice spolu s odkazem do glosáře se objeví po najetí myší na vyznačený text anebo po kliknutí prstem na zařízení s dotykovou obrazovku.

Funkce je vložená do aplikace glosáře a vyznačuje pojmy zelenou barvou. Funkce je vyzkoušena vložením do stažené webové stránky PA1. Stránka musela být stažena, protože autor práce nemá práva upravovat stránky PA1. Kopie stažené stránky (`https://courses.fit.cvut.cz/BI-PA1/elearning/functions/index.html`) z webu PA1 je přístupná na `http://bc-glossary.herokuapp.com/example` v níž je použita implementovaná funkce. Zobrazování glos v tuto chvíli s jistotou funguje v prohlížeči *Chrome*.

5.8 Další možnosti rozšíření

Aplikace glosáře může být dále rozvíjena, a to buď samotná aplikace jako taková, anebo mohou být dopsána různé rozšíření využívající aplikaci nebo podporující či spravující její chod.

5.8.1 Vylepšení aplikace

V tuto chvíli aplikace neumožňuje zaznamenávat historii úprav glos, tudíž se uživatel nemůže vrátit k původní verzi nebo porovnat novou verzi s původní. Pokud by byla sledována pouze historie úprav definice, stačí k definici přidat datum, kdy byla vytvořena, aby bylo možné sledovat, která definice předchází jiné. Sledovat, která definice předchází jiné, je možné i přes spojení definic s *rodičem*. Každá definice má parametr *active*, tento parametr lze využít pro rozhodnutí, která z definic má být použita. Samozřejmě by muselo být doimplementováno omezení zajišťující, že právě jedna definice každé glosy bude aktivní.

Dalším možným rozšířením je umožnit v aplikaci přihlášení, autorizaci uživatele, přidělování práv a přístupů. V souvislosti s přihlášením by se mohlo zaznamenávat, kdo jakou glosu vytvořil nebo upravil, pomocí přiřazení uživatele k definici, pojmu a nebo tagům. Společně s uchováváním historie by tyto dvě funkcionality tvořily kompaktní celek.

Vyhledávání a zvláště normalizace výrazů by si zasloužila vylepšení. V tuto chvíli není vyřešeno vyhledání víceslovných glos v textu. Text je totiž strojově lemmatizován anebo prostě rozdělen na slova, která jsou následně hledána v glosáři. Pro implementaci vyhledání víceslovných glos v textu by bylo potřeba udělat rešerši, která byla nad rámec této práce, nebo implementovat vyhledávání pomocí postupného procházení textu. Text by byl prohledáván postupně a byla by v něm vyhledávána přesná shoda s víceslovným pojmem. Bylo rozhodnuto, že tento způsob vyhledávání by byl časově náročný a proto pro tuto chvíli stačí, že jsou v textu vyhledány jednoslovné glosy.

Dále by se mohlo vylepšení vyhledávání zaměřit na vyhledávání i v definicích, například předzpracovat definice a následně vyhledávat ve zpracované formě. Normalizace slov by mohla být prováděna pro různé jazyky jinak. Pro některé jazyky bude výhodnější nejprve vyhledat nenormalizovaný pojem a normalizovat ho až následně, pro jiné jazyky bude výhodnější slovo normalizovat rovnou. Vyhledání v textu nikdy nebude stoprocentní, avšak může být náležitě vylepšeno a zdokonaleno.

Aplikace by dále mohla být rozšířená o našeptávání tagů při vytváření glos, o různé druhy tagů, o uložení videí a obrázků (v tuto chvíli je možné do definice vložit obrázek přes odkaz na internetu, obrázek však není uchováván v databázi), o nové nebo vylepšené funkcionality přístupné přes API a nespočet dalších vylepšení.

5.8.2 Nadstavby nad aplikací

V tuto chvíli je implementována funkce, kterou může programátor vložit do svojí webové stránky. Funkce vyznačí slova na stránce, která jsou uložená v glosáři. Druhou možností je tuto funkcionalitu zpřístupnit jako rozšíření do prohlížeče (addon), přičemž by si uživatel mohl určit, jaký jazyk má rozšíření používat, mohl by nastavit tagy a nebo, pokud by existovalo více glosářů, který glosář využít.

Dále je možné vytvořit rozšíření, které by vkládalo do glosáře glosy například z Wikipedie nebo z často firmami využívané Confluence. Nejprve by byly získány stránky pomoci API (které má Confluence i Wikipedie), následně by byly vytvořeny glosy, každá glosa by představovala jednu stránku. Takto vytvořený glosář by mohl být následně použit na jiných stránkách pomocí již implementované funkce. Do glosáře by mohly být ukládány odkazy na původní webovou stránku. Funkce by jako odkaz na glosu nemusela dávat odkaz do glosáře, ale přímo na původní stránku. Využití této funkcionality by mohlo být výhodné pro firmy. Pokud by byl vytvořen glosář

z firemní Confluence, a následně by byly glosy zobrazovány na stránkách, které zaměstnanec využívá, mohlo by docházet k rychlejšímu a přesnějšímu pochopení pojmů, ať už nováčky v týmu anebo stálými zaměstnanci, kteří potřebují pochopit nové pojmy. Díky možnosti importování glos z Confluence (nebo z jiné firemní wiki) by se uživatelé nemuseli učit s novým nástrojem, ale měli by novou funkcionalitu glosáře.

Glosář by mohl být použit pro vytvoření aplikace s recepty a návody. Každý recept, postup, nebo nástroj by byl jednotlivou glosou. Recepty by byly označeny tagem *recipe*, postupy tagem *method*, nástroje tagem *tool*. Aplikace by zobrazovala všechny recepty (vybrala by je pomocí tagu). Při zobrazení receptů by byl text receptu prohledán, přičemž by v něm byly hledány postupy a nástroje. Postupy a nástroje by byly vyznačeny a bylo by možné si u nich zobrazit náhled vysvětlení, případně přejít na vysvětlení postupu. Aplikace by mohla pomoci lidem, kteří začínají vařit a nejsou si jistí některými postupy. Například každý nemusí vědět co znamená *vyšleháme do pěny*, jak se dělá *jíška* nebo jak se *vaří vajíčka*. Pokud by taková instrukce byla v receptu, bude vyznačena a uživatel si může jednoduše zobrazit, jak takovou instrukci provést. Možností pro využití a rozšíření glosáře je mnoho a mohl by pomoci nejen v pracovní sféře, ale i v běžném životě.

Závěr

Cílem práce bylo navrhnout a napsat aplikaci pro tvorbu glosáře. Nejprve byly zanalyzovány podobné aplikace, následně byla provedena analýza poskytování dat na webu přes služby SOAP, REST a GraphQL a byly probrány dva způsoby normalizace slov – lemmatizace a stematizace. Výsledky analýzy byly použity při tvorbě funkčních a nefunkčních požadavků na aplikaci. Dále byla provedena analýza a návrh API a webového frontendu. Práce se dále zaměřila na problematiku vyhledávání slov a jejich úpravu pomocí lemmatizace.

Bylo vytvořeno GraphQL API podporující operace s objekty v glosáři a umožňující funkcionalitu vyhledání glos v textu. Zpřístupnění glos pro jejich čtení, vytváření, úpravu a mazání bylo zajištěno přes webovou aplikaci. Využití glosáře bylo demonstrováno pomocí funkce, která může být vložena do webové stránky, v níž pak vyznačí slova, nacházející se na stránce i v glosáři. Funkčnost byla vyzkoušena na stažené stránce z předmětu PA1 a je možná vložit do libovolné webové stránky.

Aplikace může být rozšířena dalšími nadstavbovými aplikacemi, například vkládání glos z wiki, nebo vyznačování glos v pdf editoru či jiných textových formátech. Aplikace samotná může být dále vylepšována a rozšiřována, například přidáním přihlašování, přidáním historie úprav, umožněním stahování glos nebo vylepšením vyhledávání.

Výsledek práce může být prospěšný pro urychlení sdílení znalostí v týmu. Glosář společně s výše zmíněnou funkcí může vyznačovat pojmy na firemních stránkách. Díky možnosti sdílení informací o projektu pomocí glosáře a této funkce bude jednodušší začlenit nové členy týmu a tím urychlit proces poznávání projektu i vývoje. Zmíněná funkce může být přidána do jakékoliv webové stránky a tím může čtenáři usnadnit rychlejší porozumění neznámých pojmů.

Bibliografie

1. LARMAN, Craig. *Applying UML and patterns*. Prentice Hall Englewood Cliffs, NJ, 1998. https://personal.utdallas.edu/~chung/SP/applying-uml-and-patterns.pdf (navštíveno 23.1.2022).
2. UHROWCZIK, Peter P. Data dictionary/directories. *IBM Systems Journal*. 1973, roč. 12, č. 4, s. 332. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.468&rep=rep1&type=pdf (navštíveno 23.1.2022).
3. Obsidian [online]. [B.r.] [cit. 2022-01-29]. Dostupné z: https://obsidian.md.
4. Roam [online]. [B.r.] [cit. 2022-01-29]. Dostupné z: https://roamresearch.com.
5. Evernote [online]. [B.r.] [cit. 2022-01-29]. Dostupné z: https://evernote.com/intl/cs/why-evernote.
6. Notion [online]. [B.r.] [cit. 2022-01-29]. Dostupné z: https://www.notion.so/product.
7. Microsoft OneNote [online]. [B.r.] [cit. 2022-01-28]. Dostupné z: https://www.microsoft.com/en-us/microsoft-365/onenote/digital-note-taking-app.
8. Confluence [online]. [B.r.] [cit. 2022-01-29]. Dostupné z: https://www.atlassian.com/software/confluence/features.
9. BookStack [online]. [B.r.] [cit. 2022-01-29]. Dostupné z: https://www.bookstackapp.com/docs/admin/installation/.
10. FIELDING, Roy T.; RESCHKE, Julian. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [RFC 7231]. RFC Editor, 2014. Request for Comments, č. 7231. Dostupné z DOI: 10.17487/RFC7231.
11. BRAY, T; PAOLI, J; SPERBERG-MCQUEEN, CM; MAILER, Y; YERGEAU, F. *Extensible Markup Language (XML) 1.0 5th Edition, W3C recommendation, November 2008*. ed, [b.r.].
12. Json. *JSON* [online]. [B.r.] [cit. 2022-04-07]. Dostupné z: https://www.json.org/json-en.html.
13. VECCHIO, D. *Legacy software: junkyard wars for web services?* 2001. Tech. zpr. Proceedings of the Gartner Symposium IT Expo Presentation, Orlando, Florida.
14. SOHAN, SM; ANSLOW, Craig; MAURER, Frank. A case study of web API evolution. In: *2015 IEEE World Congress on Services*. 2015, s. 245–252.
15. BOX, Don; EHNEBUSKE, David; KAKIVAYA, Gopal; LAYMAN, Andrew; MENDELSOHN, Noah; NIELSEN, Henrik Frystyk; THATTE, Satish; WINER, Dave. *Simple object access protocol (SOAP) 1.1*. 2000.

16. BOX, Don. A Brief History of SOAP. *XML.com* [online]. 2001 [cit. 2021-11-12]. Dostupné z: `https://www.xml.com/pub/a/ws/2001/04/04/soap.html`.

17. FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

18. MULLOY, Brian. *Web API design*. Academic Press, 2013.

19. What is the GraphQL Foundation? [Online]. [B.r.] [cit. 2022-04-07]. Dostupné z: `https://graphql.org/foundation/`.

20. BYRON, Lee. *GraphQL: A data query language* [online]. 2018 [cit. 2022-04-07]. Dostupné z: `https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/`.

21. Serving over http [online]. [B.r.] [cit. 2022-04-08]. Dostupné z: `https://graphql.org/learn/serving-over-http/`.

22. Queries and Mutations [online]. [B.r.] [cit. 2022-04-08]. Dostupné z: `https://graphql.org/learn/queries/`.

23. LOVINS, Julie Beth. Development of a stemming algorithm. *Mech. Transl. Comput. Linguistics*. 1968, roč. 11, č. 1-2, s. 22–31.

24. DOLAMIC, Ljiljana; SAVOY, Jacques. Stemming approaches for east european languages. In: *Workshop of the Cross-Language Evaluation Forum for European Languages*. 2007, s. 37–44.

25. BOTH, Csaba Attila et al. Noun Cases of Hungarian Language in Romanian. *Acta Universitatis Sapientiae, Philologica*. 2014, roč. 6, č. 3, s. 295–315.

26. SAVOY, Jacques. Report on CLEF-2003 monolingual tracks: Fusion of probabilistic models for effective monolingual retrieval. In: *Workshop of the Cross-Language Evaluation Forum for European Languages*. 2003, s. 322–336.

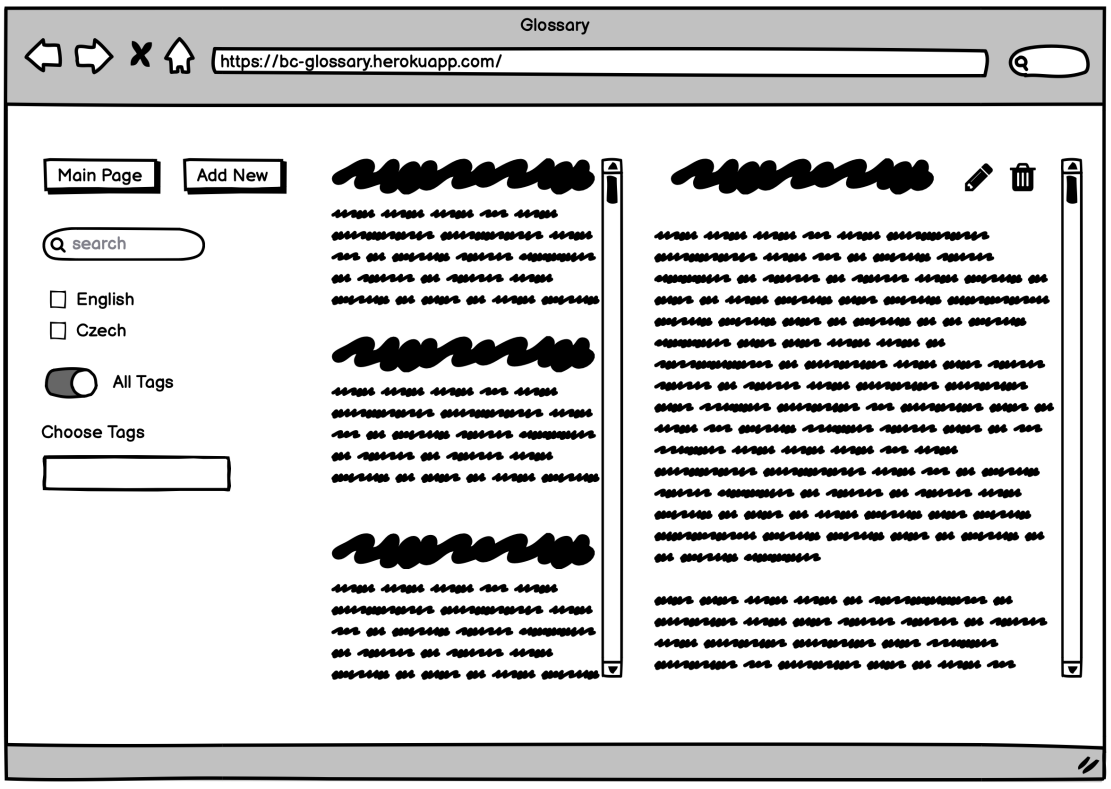
27. PLISSON, Joël; LAVRAC, Nada; MLADENIC, Dunja et al. A rule based approach to word lemmatization. In: *Proceedings of IS*. 2004, sv. 3, s. 83–86.

28. BERGMANIS, Toms; GOLDWATER, Sharon. Context sensitive neural lemmatization with lematus. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. 2018, s. 1391–1400.

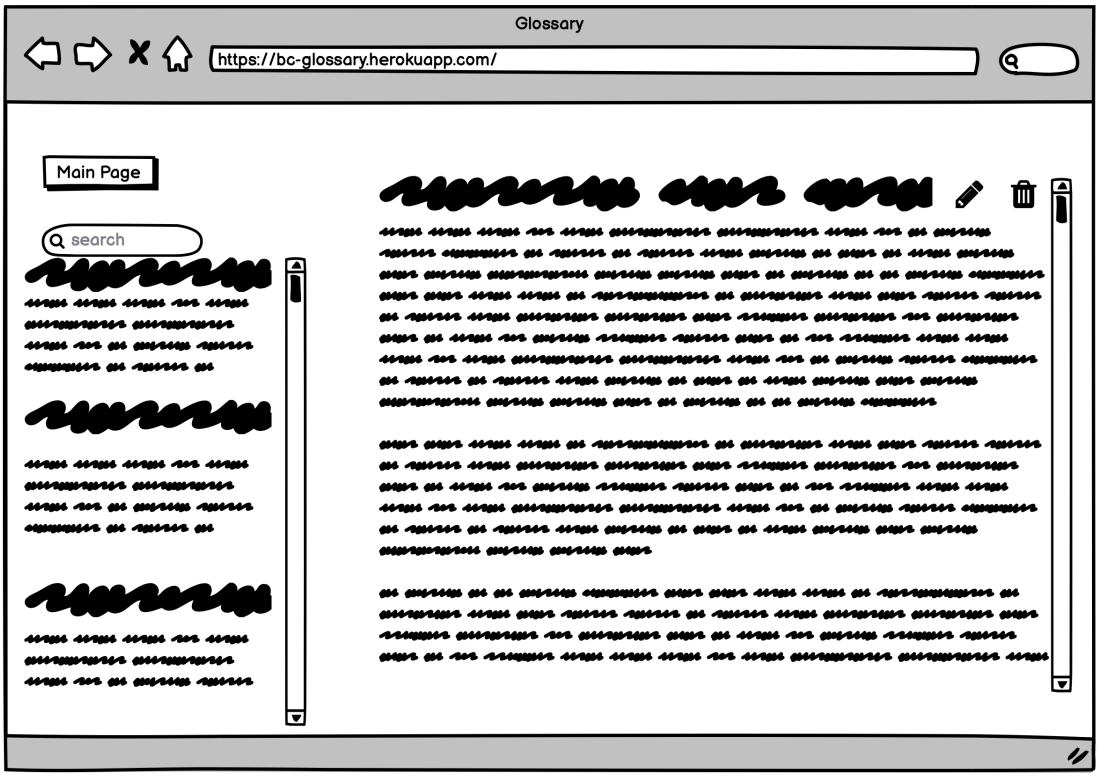
Příloha A

Wireframe

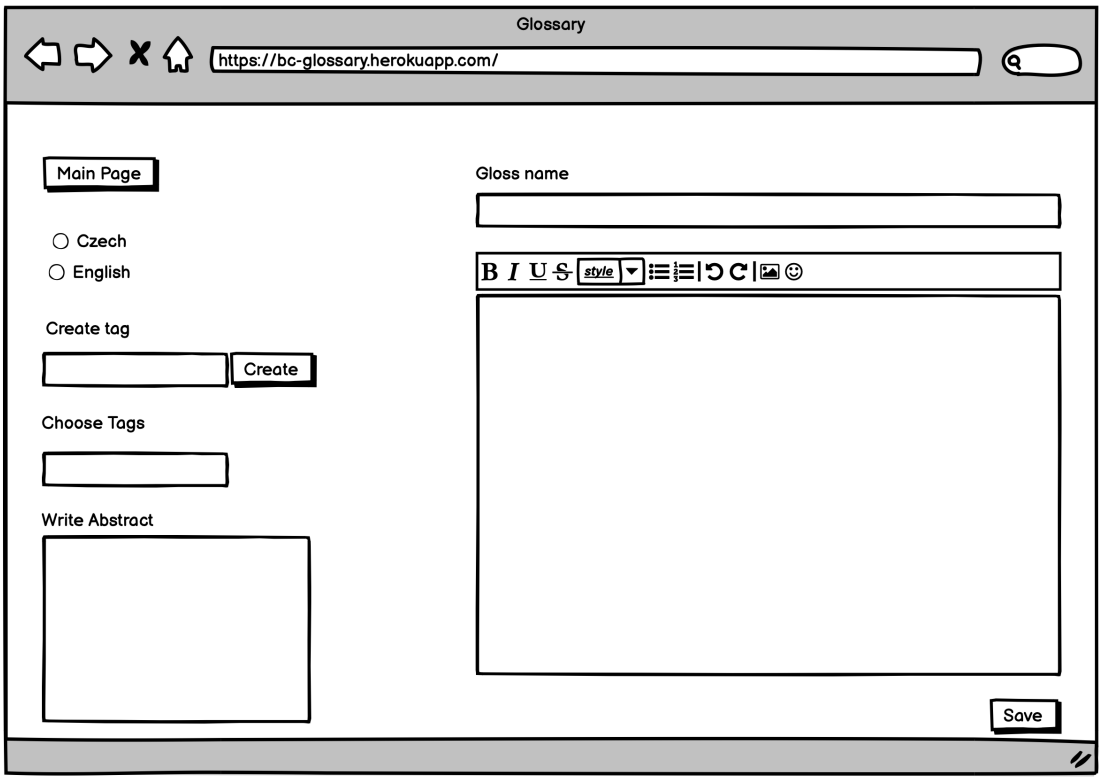
Obrázek A.1 Hlavní stránka



Obrázek A.2 Stránka glosy



Obrázek A.3 Editor



Obsah přiloženého média

- `README.md` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `glossary` ..... zdrojové kódy implementace glosáře
    - `glossary-function` ..... zdrojové kódy funkce do webové stránky
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF