FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	GraphWiki
Student:	Bc. Tomáš Greger
Vedoucí:	Ing. Marek Sušický
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat webový systém nazvaný GraphWiki. Cílem je uchovávat vazby mezi osobami a firmami, které mohou lidé volně zadávat, případně může jít o automatický import na základě těžby NLP, či využití wikidata.org. Každá takto zadaná vazba musí mít informaci o zdroji a musí být možné zařídit proces schvalování, tzn. vazby je nutné nejprve odsouhlasit. Je možné si na existující vazbu stěžovat.
Systém bude umožňovat vizualizaci vazeb a fulltextové vyhledávání informací. Systém bude kompatibilní s formátem grafu, který nabízí program ClueMaker a také s běžným standardem GraphML. Bude podporovat alespoň dvě rozložení grafu - hierarchické a organické.
Uzly budou moci odkazovat na Wikipedii, počítá se s užším napojením na portál Wikidata. Data budou uchovávána v grafové databázi JanusGraph.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 6. ledna 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

GraphWiki

Bc. Tomáš Greger

Katedra softwarového inženýrství
Vedoucí práce: Ing. Marek Sušický

21. května 2020

Poděkování

Chtěl bych poděkovat především své rodině, přítelkyni a přátelům, kteří mě během studia plně podporovali. Dále bych chtěl poděkovat vedoucímu své diplomové práce Ing. Markovi Sušickému za spolupráci, rady a připomínky během tvorby práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 21. května 2020 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Greger, Tomáš. *GraphWiki*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Cílem této diplomové práce je analyzovat, navrhnout, implementovat, otestovat a zdokumentovat webovou aplikaci, která bude založena na myšlence grafové Wikipedie. Uživatelům bude v této fázi poskytovat informace o vztazích mezi subjekty typu osoba a firma, a to v různých podobách. Práce obsahuje rešerši dostupných řešení s podobnými funkcionalitami, společně s vyhodnocením jejich pozitiv i nedostatků. K implementaci byly použity moderní nástroje podporující škálování a distribuované zpracování. Výsledná aplikace obsahuje možnost zobrazování vztahů mezi subjekty ve dvou grafových rozloženích, správu těchto dat a v neposlední řadě je kompatibilní s běžných grafovým formátem GraphML a s nástrojem ClueMaker, který je vyvíjen společností Profinit EU, s.r.o.

Klíčová slova grafová Wikipedie, vztahy osob a firem, JanusGraph, Spring Boot, Angular

Abstract

The aim of this thesis is to analyze, design, implement, test and document a web application based on the idea of graphical Wikipedia. At this stage, the application will provide information in various forms about the relationships

between people and companies. This thesis contains research of solutions with similar functions together with the evaluation of their advantages and disadvantages. During the implementation phase, modern tools supporting scaling and distributed processing were used. The final application allows to display relationships between entities in two graphical layouts, manage this data and, last but not least, is compatible with the common graph format GraphML and the tool ClueMaker from Profinit EU, s.r.o.

Keywords graphical Wikipedia, relationships between people and companies, JanusGraph, Spring Boot, Angular

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Webové portály s velkým množstvím veřejně dostupných dat z různých oborů jsou bezesporu pro širokou veřejnost velikým přínosem. Lze totiž prakticky na jednom místě najít téměř veškeré informace, které potřebujeme v našem každodenním životě. Vytváření takových portálů patří mezi hlavní iniciativu nadace Wikimedia Foundation [1], mezi jejíž projekty patří Wikidata (databáze vědomostí), Wikipedie (otevřená encyklopedie), Wikiknihy (výukové knihy a příručky) a mnoho dalších.

Projekt GraphWiki vychází z myšlenek a projektů této nadace. Bere si za cíl vytvoření obdoby projektu Wikipedie, ale v tomto případě v grafické podobě. Grafické zobrazení je pro uživatele atraktivní hlavně v případech, kdy potřebují pochopit nejrůznější vztahy mezi daty.

V této práci bude vytvořen zárodek projektu GraphWiki. Cílem je vytvořit webovou aplikaci, která bude v této verzi podporovat vztahy mezi osobami a firmami. Vzhledem k plánovanému přidávání dalších druhů entit, bude však kladen důraz na rozšiřitelnost a škálovatelnost celého řešení.

Kapitola 1

Současný stav a řešení

Jak již bylo řečeno v úvodu, tato práce bude podporovat v současné verzi data týkající se vztahů mezi osobami a firmami. Na českém trhu existuje několik nástrojů s obdobnými funkcionalitami, které jsou požadované od výsledné aplikace. Většinu z těchto nástrojů lze charakterizovat tím, že jejich primárním cílem je poskytovat o subjektech různé ekonomické ukazatele, mimo to však zobrazují i vztahy mezi těmito subjekty.

Rešerše obsahuje stručné zhodnocení vybraných existujících řešení a je provedena pouze z hlediska relevantního pro projekt GraphWiki.

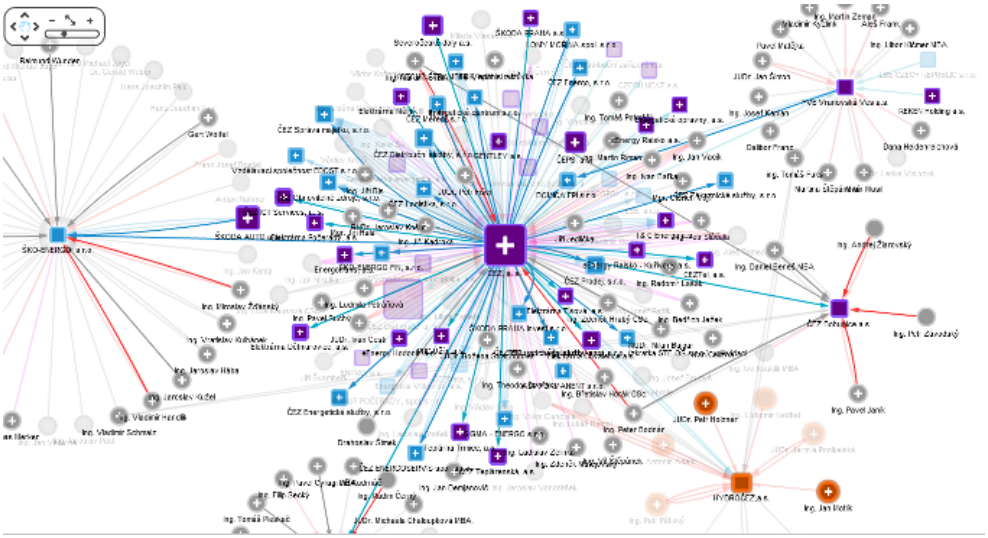
1.1 Portál www.podnikani.cz

1.1.1 Popis

„*Webové stránky podnikání.cz poskytují vizuální zobrazení nejdůležitějších informací z českého obchodního rejstříku v jednoduché a snadno přehledné grafické podobě. Tyto stránky jsou určeny pro každého, kdo potřebuje snadno a rychle zjistit vazby a souvislosti mezi jednotlivými lidmi a firmami v podnikání v ČR a jejich historii.*“ [2]

1.1.2 Zhodnocení

Tento portál zobrazuje vazby mezi osobami a firmami přehledně, data lze interaktivně procházet a u každé osoby/firmy lze zobrazit podrobnější informace. Každá vazba má uvedený svůj typ a začátek trvání. Nespornou výhodou portálu je jeho dostupnost pro širokou veřejnost. Jako nedostatky lze uvést, že portál využívá data pouze z českého obchodního rejstříku. Při používání portálu byla také zpozorována značná časová prodleva při hledání komplexnějších subjektů.



Obrázek 1.1: Vizualizace vztahů subjektů z portálu www.podnikani.cz [2]

1.2 Portál www.firmo.cz

1.2.1 Popis

„*FIRMO.CZ zpracovává informace z živnostenského a obchodního rejstříku, sbírky listin, insolvenčního rejstříku a veřejného registru dlužníků. Zároveň v něm najdete údaje ohledně spolehlivosti plátců DPH, veřejných zakázek, dotací z EU, volných pracovních míst, patentů a obchodních známek.*“ [3]

Portál mimo zobrazení konsolidovaných dat z veřejně dostupných rejstříků nabízí také vyhledávání vazeb mezi dvěma subjekty, a to včetně časové souslednosti, nebo přehledný výběr ekonomických ukazatelů. [4]

1.2.2 Zhodnocení

Nabízející funkce dělají z FIRMO.CZ kvalitní nástroj pro analýzu jednotlivých subjektů a vztahů mezi nimi. Informace z různých rejstříků zobrazuje na jednom místě, což při prozkoumávání většího množství subjektů může znamenat značnou časovou úsporu. Zásadní komplikací k užívání tohoto řešení širokou veřejností je však fakt, že drtivá většina poskytovaných funkcí je placená.

1.3 Merk

1.3.1 Popis

„*Merk není jen databáze firem, ale základní obchodní a marketingový nástroj, který denně používá přes 15 000 uživatelů ve více než 2 500 společnostech. Pomůže získat nové zákazníky, lépe segmentovat ty stávající a monitorovat trh či konkurenci.*“ [5]

1.3.2 Zhodnocení

Merk je komplexní nástroj, který poskytuje data o všech firmách v České republice. Primárním zájmem tohoto nástroje jsou ekonomické a marketingové ukazatele, u jednotlivých subjektů však obsahuje i další, pro projekt GraphWiki zajímavé, entity, jimiž jsou například adresy a telefonní nebo emailové kontakty. Nástroj rovněž nabízí grafické znázornění vazeb mezi subjekty. Tento nástroj je opět placeným, a tak není možné jeho užívání širokou veřejností.

1.4 Bisnode

1.4.1 Popis

Jedním z produktů společnosti Bisnode je nástroj BIZguard, který provádí pokročilou analýzu obchodních dat z České a Slovenské republiky.

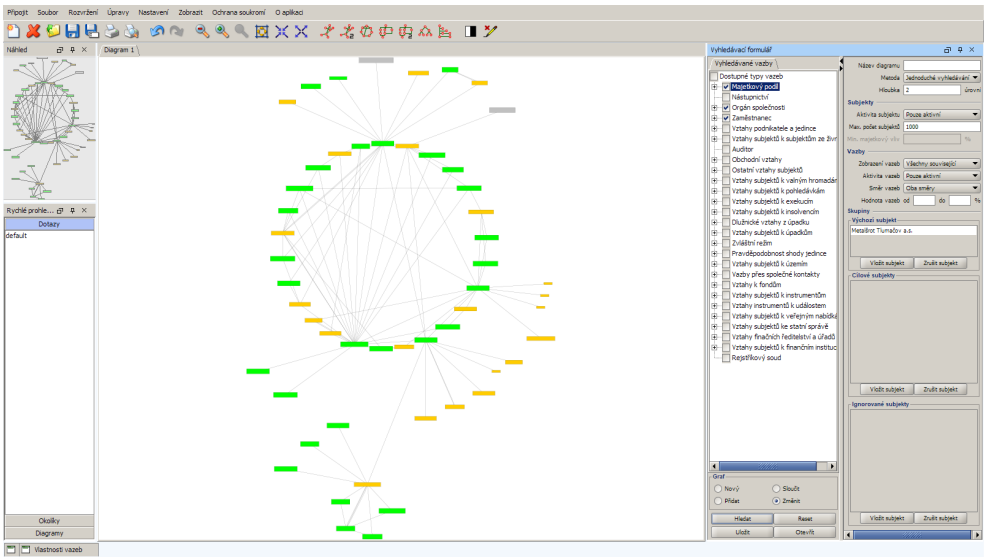
„*Bisnode BIZguard je aplikace pro vizualizaci a grafickou analýzu vztahů a vazeb mezi ekonomickými subjekty na trhu. Snadno zobrazí okolí zvoleného subjektu i vykreslí propojení mezi více subjekty. Aplikace obsahuje nástroje pro dynamickou práci s diagramy, jejich editaci, slučování nebo filtrování.*“ [6]

1.4.2 Zhodnocení

Tento nástroj je určen především pro útvary firem jako jsou řídící management, právní nebo obchodní oddělení a další. Opět se jedná o placený nástroj, tudíž je pro širokou veřejnost znovu nepoužitelný.

1.5 Shrnutí

Hlavním cílem projektu GraphWiki je poskytnout data široké veřejnosti. Tento požadavek splňuje pouze portál www.podnikani.cz. Jeho nevýhodou je však již zmíněná omezenost zdroje dat. Ostatní nástroje, které byly v rešerši probrány, jsou typicky primárně určené pro obchodní účely nebo marketing. Zpoplatnění těchto nástrojů také znemožňuje jejich použití širokou veřejností.



Obrázek 1.2: Vizualizace vztahů subjektů pomocí Bisnode BIZguard [6]

Kapitola 2

Analýza a návrh

V této kapitole budou mimo jiné rozebrány funkční a nefunkční požadavky kladené na výslednou aplikaci, případy užití, návrh uživatelského rozhraní nebo komplexnější procesy, které v aplikaci budou probíhat. Tato kapitola neobsahuje návrh samotné architektury aplikace, které se věnuje kapitola 3.

V této a v následujících kapitolách bude uváděn pojem *grafová entita*, kterému v kontextu tohoto dokumentu budeme rozumět jednu z následujících možností – osoba, firma nebo vztah mezi libovolnou dvojicí takových subjektů.

2.1 Funkční požadavky

Funkčním požadavkem rozumíme popis služby, kterou musí systém poskytovat. Funkce systému není nic jiného než vstup, jeho chování a výstup. Za funkci považujeme libovolný výpočet, manipulaci s údaji, obchodní proces, interakci s uživatelem nebo jakoukoliv jinou specifickou funkci. [7]

Následuje výčet jednotlivých funkčních požadavků, jejichž kompletní shrnutí je dostupné v tabulce 2.1.

Tabulka 2.1: Přehled funkčních požadavků

ID	Funkční požadavek
FR-1	Vizualizace vazeb
FR-2	Zobrazení relevantních dat
FR-3	Přidávání nových grafových entit
FR-4	Schválení grafové entity
FR-5	Původ dat
FR-6	Stížnost na existující grafovou entitu
FR-7	Zobrazení historie
FR-8	Zobrazení úkolů
FR-9	Kompatibilita s jinými formáty

2.1.1 Vizualizace vazeb

Vizualizace vazeb mezi osobami a firmami je primárním požadavkem na tuto aplikaci. Relevantní záznamy (viz 2.1.2) budou dostupné přes fulltextové vyhledávání a data bude možné zobrazovat ve dvou grafových rozloženích (hierarchické a organické).

2.1.2 Zobrazeni relevantních dat

Každá uživatelská role bude mít dostupné pouze grafové entity v určitém stavu. Typický scénář je, že správce aplikace bude mít přístupné i entity, které ještě nebudou schváleny nebo ty, které budou zrušené, na rozdíl od klasického uživatele, kterému budou přístupné pouze entity aktivní. Aktivní entitou se zde rozumí entita, která je ve stavu bezprostředně následujícím po schválení správcem aplikace. Podrobnější popis stavů grafové entity je dostupný v kapitole 2.4.

2.1.3 Přidávání nových grafových entit

Uživatelům se specifickou rolí bude umožněno v aplikaci přidat libovolnou grafovou entitu. Po tomto kroku bude následovat proces schvalování, viz 2.1.4.

2.1.4 Schválení grafové entity

Pro uživatelem přidanou grafovou entitu vznikne automaticky žádost, kterou je potřeba nejdříve zkontrolovat a schválit správcem aplikace. Grafová entita bude dostupná všem uživatelům až po schválení této žádosti.

2.1.5 Původ dat

Každá grafová entita v aplikaci musí mít informaci o svém zdroji, která bude dostupná všem uživatelům. V tomto případě se bude jednat o adresu webové stránky.

2.1.6 Stížnost na existující grafovou entitu

Uživatel s patřičnou rolí bude mít možnost si stěžovat na existující grafovou entitu. Stížnost bude opět podstoupena správci aplikace k posouzení a následnému schválení či zamítnutí.

2.1.7 Zobrazení historie

Na detailu grafové entity bude dostupná historie přechodů mezi jednotlivými stavy (viz 2.4) a historie o založení navázaných žádostí/stížností.

2.1.8 Zobrazení úkolů

Pro správce aplikace budou dostupné schránky, kde uvidí seznam úkolů, které je potřeba zpracovat. Úkoly bez přiřazeného řešitele budou dostupné v týmové schránce, úkoly přiřazené na daného správce pak v jeho soukromé schránce. Přiřazený úkol bude umožněno vrátit do týmové schránky.

2.1.9 Kompatibilita s jinými formáty

Systém bude kompatibilní s běžným standardem GraphML [8] a s formátem grafu, který nabízí program ClueMaker [9].

2.2 Nefunkční požadavky

Nefunkčním požadavkem rozumíme atribut kvality softwarového díla, mezi které lze mimo jiné zařadit požadavky na škálovatelnost, přenositelnost, zabezpečení nebo udržovatelnost. Nesplnění nefunkčních požadavků může vést k systému nesplňujícímu potřeby uživatele. [7]

Obdobně jako v předchozím případě je shrnutí jednotlivých nefunkčních požadavků uvedeno v tabulce 2.2.

Tabulka 2.2: Přehled nefunkčních požadavků

ID	Nefunkční požadavek
NFR-1	Zabezpečení grafových dat dle role
NFR-2	Uživatelská hesla v šifrované podobě se solí
NFR-3	Modulární architektura
NFR-4	Princip HATEOAS
NFR-5	Použití technologií – JanusGraph, Elasticsearch, Cassandra
NFR-6	Cílová platforma – stolní počítač
NFR-7	Podpora definované verze prohlížeče Google Chrome

2.2.1 Zabezpečení

Následují požadavky kladené na zabezpečení aplikace:

- Grafová data budou zabezpečena dle role, viz kapitola 2.1.2.
- Uživatelská hesla musí být uložena v šifrované podobě s použitím soli.

2.2.2 Rozšiřovatelnost a udržovatelnost

Na výsledný produkt jsou v této oblasti kladeny následující požadavky:

- Aplikace bude členěna na jednotlivé moduly dle logických celků tak, aby byla možná náhrada jednotlivých částí.
- Jednotlivé moduly budou z důvodu jednoduššího sestavení a nasazení aplikace svázány do tzv. *multimodule* projektu.
- Při implementaci bude použit princip HATEOAS s možnými výjimkami spojenými s použitými technologiemi.

2.2.3 Použité technologie

Pro zajištění škálovatelnosti je požadováno, aby aplikace byla implementována za použití těchto technologií:

- grafová databáze – JanusGraph [10],
- fulltextový vyhledávač – Elasticsearch [11],
- úložiště – Cassandra [12].

2.2.4 Podpora

Mezi požadavky na cílovou platformu aplikace patří:

- Aplikace bude určena pro stolní počítače.
- Aplikace bude podporovaná webovým prohlížečem Google Chrome ve verzi 80.0.3987.163 a vyšší.

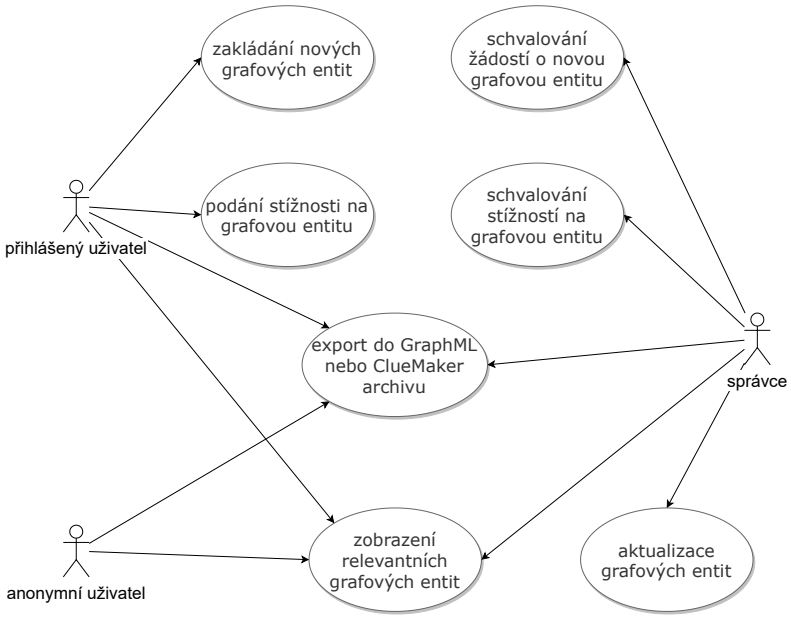
2.3 Případy užití

Z předchozích požadavků vyplývá, že celý systém musí obsahovat několik typů rolí. Z návrhu vyplynuly tři typy aktérů – anonymní uživatel, přihlášený uživatel a správce. Každý aktér bude mít dostupný jen omezený výčet funkcí, přehled těch nejdůležitějších funkcí systému společně s tím, kteří aktéři je budou moct využívat, je zobrazeno na diagramu 2.1.

2.4 Stavový diagram grafové entity

Jak již bylo zmíněno v kapitolách 2.1.2, samotná grafová entita se během svého životního cyklu může nacházet v různých stavech. Každý tento stav má své specifické chování, tj. například dostupnost uživatelům k zobrazení nebo dostupnost akcí pro práci s danou entitou.

Životní cyklus grafové entity je zobrazen na diagramu 2.2.



Obrázek 2.1: Diagram případů užití

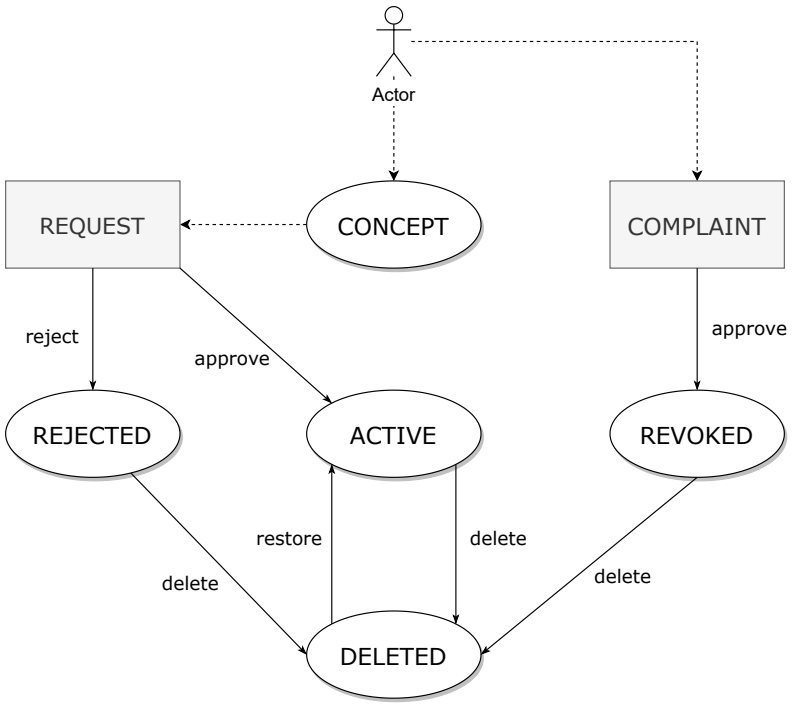
2.5 Diagram aktivit

V této kapitole budou rozebrány a vizualizovány vybrané procesy s komplexnější strukturou, které budou v systému probíhat.

2.5.1 Založení a schválení/zamítnutí nové grafové entity

Přihlášeným uživatelům bude v systému dostupná možnost přidávat nové grafové entity. Tyto entity však budou muset být nejdříve schváleny správcem aplikace. Po založení uživatelem bude entita uložená do databáze se stavem *concept* (tato entita se uživatelům zobrazovat nebude, dokud nebude schválená). Současně s uložením nové entity vznikne nová žádost, na základě této žádosti pak nový úkol, který bude dostupný správcům v jejich týmové schránce. V případě, že tým správců žádost zamítne, stav žádosti i k ní náležící grafové entity se změní na *rejected*. V opačném případě, kdy žádost bude schválena, se žádost přesune do stavu *approved* a náležící grafová entita do stavu *active*. V tuto chvíli se grafová entita začne zobrazovat všem uživatelům. V obou případech se do historie grafové entity zanesou záznamy o vzniku žádosti a o následné změně stavu.

Celý proces schválení je zobrazen na diagramu 2.3.



Obrázek 2.2: Životní cyklus grafové entity

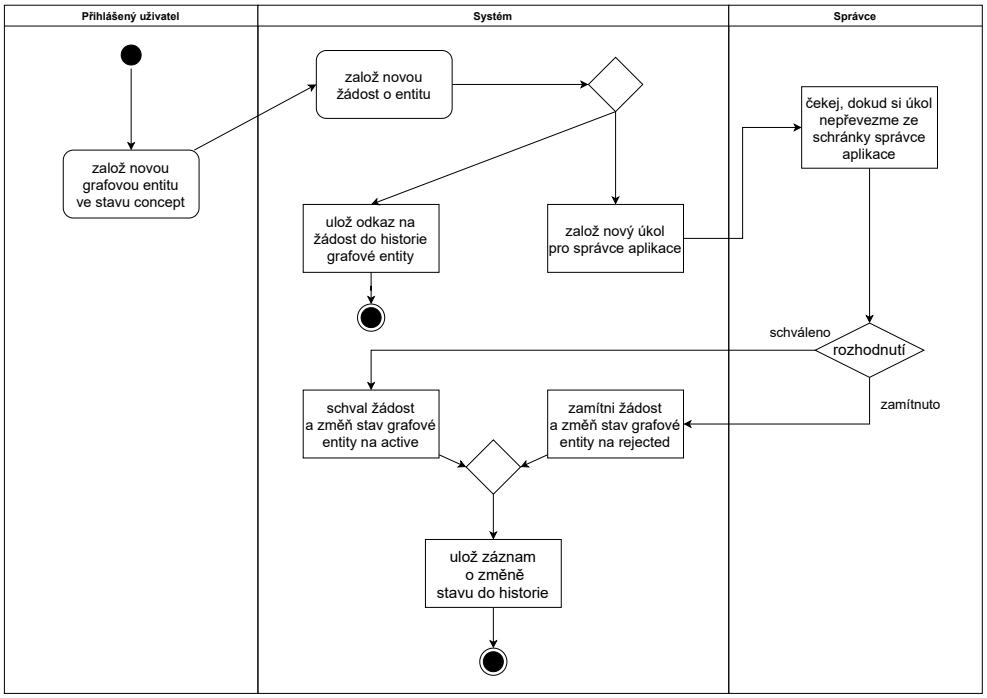
2.5.2 Stížnost na existující grafovou entitu

Přihlášený uživatel bude mít možnost stěžovat si na již existující grafovou entitu. U každé takové entity bude dostupná historie již založených stížností (současné i historické). Ke každé stížnosti budou evidovány přidružené komentáře. Po založení se stížnost předá k rozhodnutí týmu správců, kteří ji najdou v týmové schránce úkolů. Správce může stížnost schválit (tímto se stav stížnosti změní na *approved*, stav entity na *revoked* a stav ostatních probíhajících stížností na stejné entitě na *entity-modified*) nebo zamítnout (tímto se stav stížnosti změní na *rejected*, stav samotné entity se nezmění). Stejně jako u procesu schvalování se v historii entity objeví záznam o vzniku stížnosti a v případě schválení stížnosti i informace o změně stavu.

Výsledný proces je zobrazen na diagramu 2.4.

2.6 Návrh uživatelského rozhraní

Následující kapitola se věnuje návrhu uživatelského rozhraní nejdůležitějších obrazovek aplikace.



Obrázek 2.3: Založení a schválení/zamítnutí nové grafové entity

2.6.1 Horní menu

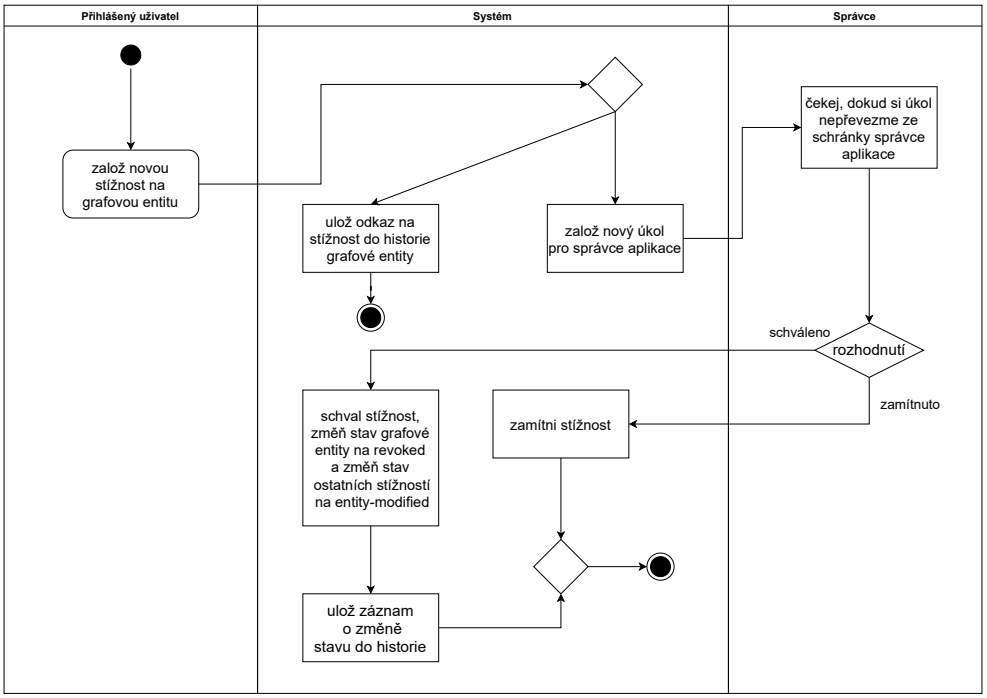
Horní menu bude zobrazeno na každé obrazovce a bude obsahovat základní elementy k navigaci v rámci celé aplikace, například odkazy na úvodní stránku, na stránky k přidání grafových entit nebo k přihlášení uživatele.

2.6.2 Úvodní obrazovka

Úvodní obrazovka aplikace bude uživatelům poskytovat možnost vyhledání osob nebo firem dle zadaného výrazu a jejich následné zobrazení. U každého záznamu budou uvedeny významné atributy, tj. jméno, příjmení a datum narození u osoby, resp. název, IČO a sídlo u firmy. Uživatel bude mít možnost přejít ke grafovému zobrazení či k detailu daného subjektu.

2.6.3 Obrazovka zobrazení grafu

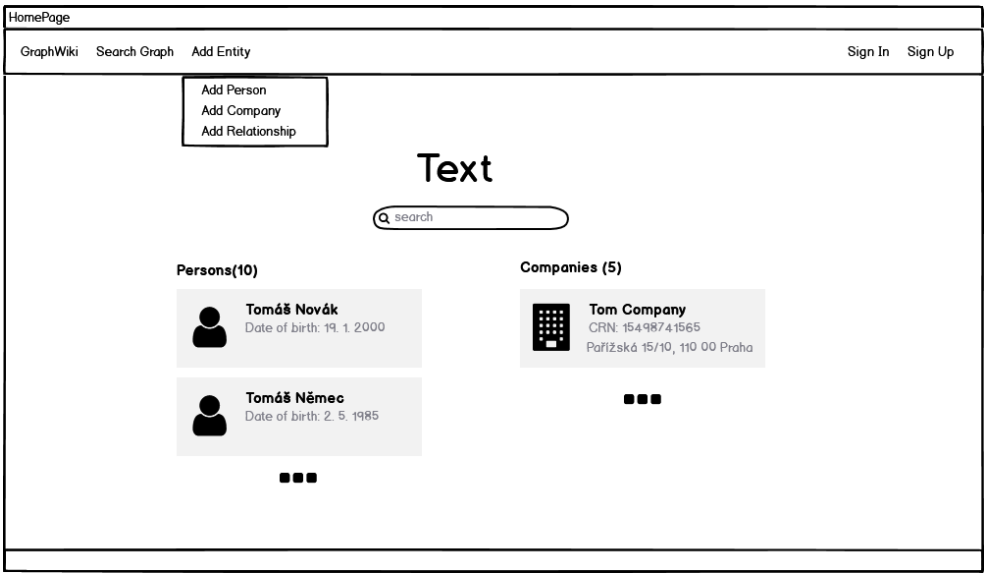
Tato obrazovka je rozdělena do dvou sekcí – samotné zobrazení grafu a přehled informací o hledaném subjektu. U grafového zobrazení budou uživateli dostupné navigační prvky umožňující procházení historie vyhledávání, změnu rozložení grafu nebo stáhnutí exportů.



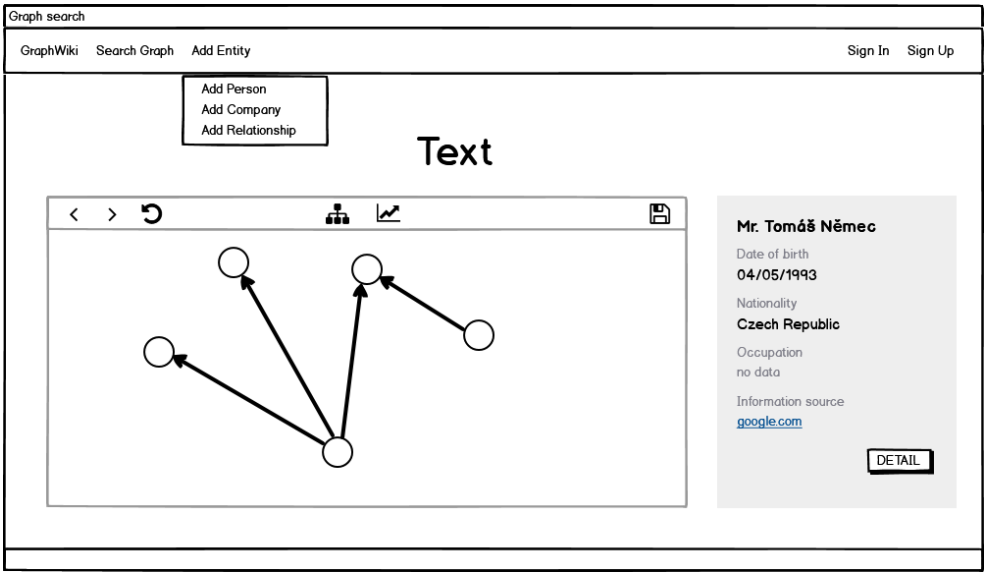
Obrázek 2.4: Stížnost na existující grafovou entitu

2.6.4 Obrazovky detailů entit, žádostí a stížností

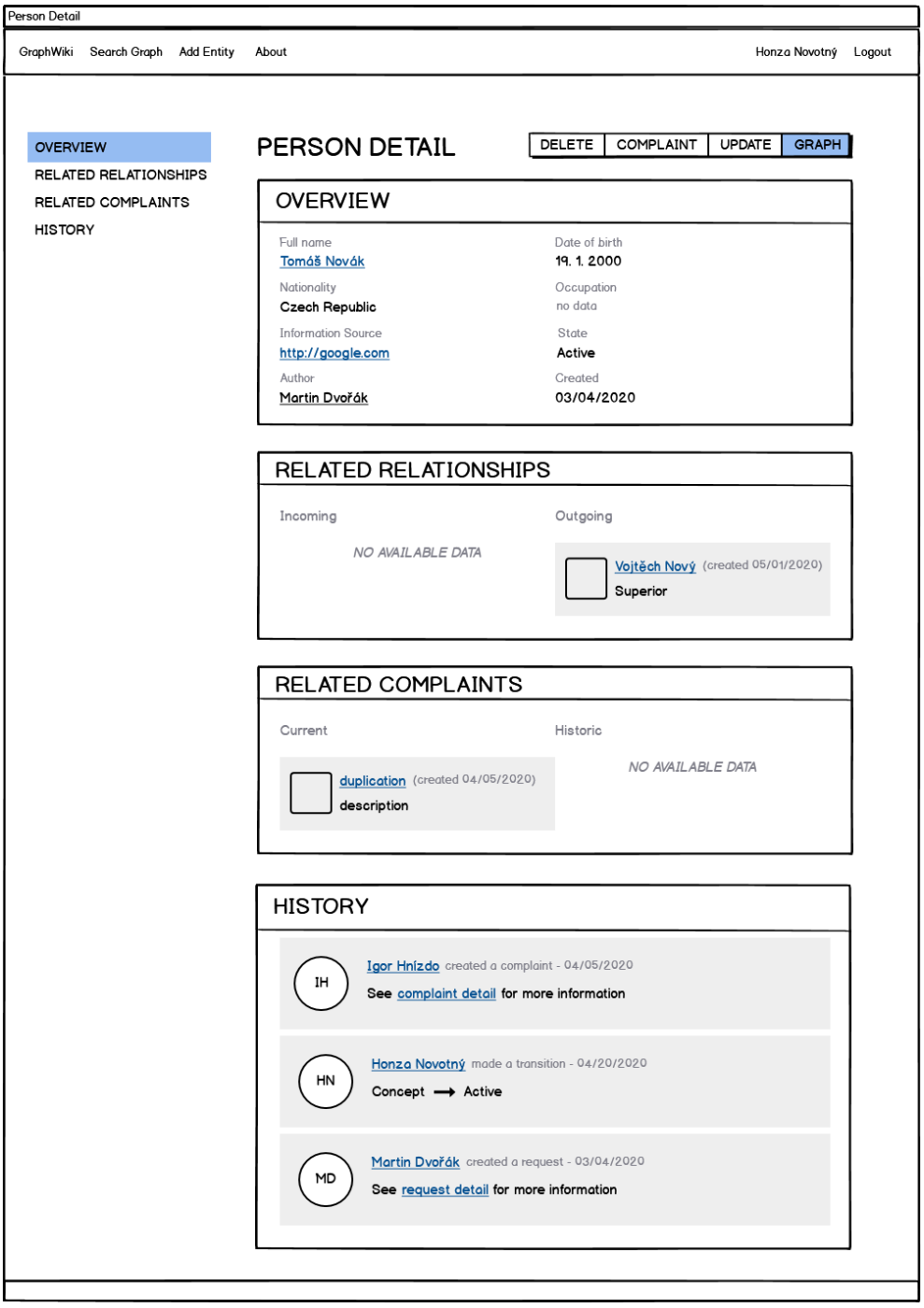
Každá obrazovka spadající do této kategorie bude standardně rozdělena do několika dílčích sekcí. Ke každé takové sekci bude příslušet záznam v interaktivním navigačním panelu, který bude uživatelům umožňovat přechod mezi těmito sekcemi. Návrh jednoho ze zástupců této kategorie, stránky detailu osoby, je zobrazen na obrázku 2.7.



Obrázek 2.5: Návrh úvodní obrazovky



Obrázek 2.6: Návrh obrazovky zobrazení grafu



Obrázek 2.7: Návrh obrazovky detailu osoby

Kapitola 3

Architektura

V této sekci budou analyzovány možné přístupy z pohledu architektury a bude navržena cílová alternativa.

Architekturu aplikace lze rozdělit do několika vrstev, které budou popsány v následujících kapitolách.

3.1 Databázová vrstva

Výběr správného databázového systému je kritickým rozhodnutím při tvorbě nové aplikace. Toto rozhodnutí může rozhodovat o úspěchu a neúspěchu celého projektu. Špatné rozhodnutí při výběru se nemusí projevit ihned, ale například až po ročním produkčním provozu, kdy už je jakákoliv změna velice náročná až nemožná.

3.1.1 Průzkum

Při volbě databázového systému máme možnost vybírat z klasických relačních databází (viz 3.1.1.2) nebo z databází nerelačních (viz 3.1.1.3).

Článek [13] uvádí, že jako podklad pro správné rozhodnutí je dobré si zodpovědět minimálně následující sadu otázek:

- jakou strukturu mají data,
- jaké je množství dat,
- k jakému účelu data slouží,
- jak často se data mění,
- jak často jsou data čtena,
- jak komplexní jsou dotazy,
- jak jsou data zásadní pro obchod,

- zda preferujeme vlastnost *consistency*, *availability* nebo *partition tolerance* (viz 3.1.1.1).

Dalším důležitým bodem, který je potřeba vzít v potaz, je škálovatelnost databázového systému. Například dotazy, které je potřeba zodpovědět, jsou v mnoha odvětvích stále komplexnější a tak nároky na výpočetní výkon rostou a rostou. V dřívějších dobách bylo standardním řešením použití výkonnější jednotky, tzv. vertikální škálování. Nicméně dnešní doba, kdy poptávka po distribuovaných systémech je stále častější, nabízí škálování horizontální, tj. zapojení více výpočetních jednotek do tzv. clusteru. Při zvolení distribuovaného řešení si je ale potřeba uvědomit omezení, která popisuje CAP teorém, který je rozebraný v kapitole 3.1.1.1.

3.1.1.1 CAP teorém

Dle článku [14] CAP teorém tvrdí, že distribuované databázové systémy mohou mít pouze dvě z těchto tří vlastností – *consistency*, *availability*, *partition tolerance*.

Consistency znamená, že všechny uzly mají v daný moment stejná data. Pokud v tomto případě provedeme zápis dat na konkrétní uzel, tak následný dotaz na libovolný uzel vrátí stejná data, jako by vrátil tento uzel. Napříč takovým systémem probíhají transakce, které převádí systém z jednoho konzistentního stavu do druhého.

Availability tvrdí, že dotaz na funkční uzel musí skončit odpovědí. Jinými slovy, funkční uzel nesmí odmítnout dotaz.

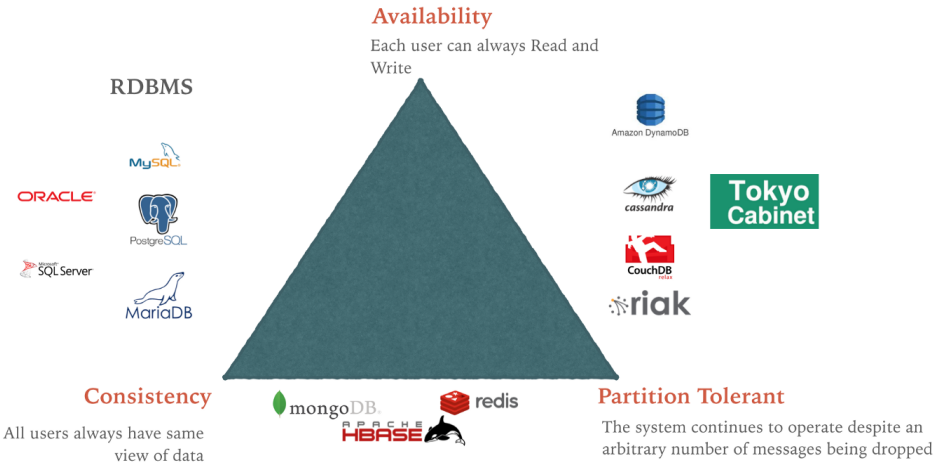
Partition tolerance uvádí, že systém je schopných pracovat i přes možné síťové výpadky. Takový systém je schopný udržet jakékoli selhání sítě, které nevede k selhání sítě celé. Datové záznamy jsou v tomto případě replikované v dostatečném množství napříč kombinací uzlů a sítí, aby udržovaly systém v provozu.

3.1.1.2 Relační databáze

Tradiční relační databáze nabízejí vlastnosti *consistency* a *availability* na úkor *partition tolerance* a jsou optimalizované pro zápis. [13]

Dle [16] jsou v následujících bodech shrnuté základní vlastnosti relačních databází:

- jsou založeny na relačním modelu,
- reprezentace dat v tabulkách,
- každý řádek v tabulce je záznamem s jedinečným klíčem,
- sloupce tabulky obsahují atributy dat a každý záznam má obvykle hodnotu pro každý atribut,



Obrázek 3.1: CAP teorém [15]

- normalizovaná data (není podmínkou).

3.1.1.3 Nerelační databáze

Nerelační databáze jsou naproti tomu optimalizované pro čtení a nabízejí jednotlivé vlastnosti dle požadavků uživatele. [13]

Dle [17] mezi základní charakteristiky nerelačních databází patří:

- nepoužívají relační model,
- možnost ukládání velkého množství dat,
- podpora strukturovaných, polostrukturovaných i nestrukturovaných dat,
- používají model úložiště optimalizovaný pro konkrétní požadavky typu uložených dat,
- nenormalizovaná data.

3.1.2 Požadavky

Hlavním požadavkem na výslednou aplikaci je, aby umožnila uchovávat vazby mezi subjekty. Dá se očekávat, že těchto dat bude velké množství, mohou být nestrukturovaná, jejich čtení bude hodně časté, dotazy mohou být komplexní, aktualizace dat bude výjimečná a vlastnost *consistency* můžeme potlačit na úkor vlastnosti *partion tolerance*.

Mimo to však bude nutné ukládat data ohledně uživatelských účtů a rolí nebo další podpůrné informace, například záznamy o úkolech. U těchto dat se

naopak očekává strukturovanost, dotazy na ně budou jednoduché a je u nich vyžadována vlastnost *consistency*.

3.1.3 Výběr

Následuje výběr databázových systémů pro oba typy dat, které byly zmíněné v předchozí kapitole.

3.1.3.1 JanusGraph

Pokud se podíváme na přirozenou strukturu dat vztahů mezi subjekty, vyjde nám z modelu graf, kde uzel je jeden ze subjektů a hrana je vztah mezi dvěma subjekty. Při použití grafové databáze můžeme data modelovat tak, jak vypadají, jako grafy. Odpadá nám tedy práce s vymýšlením různých myšlenkových modelů a samotné modelování je přímočaré.

K ukládání těchto dat bude ve výsledné aplikaci použita nerelační grafová databáze JanusGraph [10], jako datové úložiště pak Cassandra [12]. Volba těchto technologií vyplývá přímo ze zadání a z nefunkčních požadavků, viz kapitola 2.2. Grafová databáze JanusGraph bude více popsána v kapitole 4.1.

3.1.3.2 PostgreSQL

Pro ostatní data zmíněná v kapitole 3.1.2 bude použita klasická relační databáze – PostgreSQL [18].

PostgreSQL je open source, objektově-relační databáze, kterou lze řadit mezi nejlepší ve svém oboru. [19]

Volba padla na tuto databázi hlavně z důvodu spolehlivosti, široké komunitě, předchozí osobní zkušenosti a také z toho důvodu, že je dostupná jako open source.

3.2 Serverová vrstva

I na této vrstvě máme z čeho vybírat, sbírka možných technologií, jednotlivých architektur nebo návrhových vzorů je nepřeberná. Správné rozhodnutí je dalším kritickým bodem při tvorbě nové aplikace.

3.2.1 Průzkum

Následuje rozbor dvou standardních architektur – monolitické a microservice.

3.2.1.1 Monolitická architektura

První architekturou, která se v tomto případě nabízí, je architektura monolitická. Aplikace tohoto typu je charakteristická jedním zdrojovým kódem. Kód

může být rozdělen do jednotlivých modulů dle obchodních nebo technických požadavků, avšak celá aplikace se sestavuje a nasazuje jako jeden celek. [20]

Tento typ architektury přináší několik nevýhod, zvláště pak u rozsáhlých projektů. Mezi ně můžeme řadit:

- složité zaučení nových vývojářů,
- škálování – není možné škálovat pouze určitou část systému,
- doba sestavení,
- jakákoliv změna vyžaduje nasazení celé aplikace,
- změna použitých technologií je velmi složitá,
- jednotlivé moduly aplikace nelze rozdělit do nezávislých týmů,
- důsledkem selhání jednoho z modulů je selhání celého systému. [20]

3.2.1.2 Microservice architektura

Microservice architektura se skládá z několika nezávislých a soběstačných služeb. Každá z těchto služeb je charakteristická následujícími vlastnostmi:

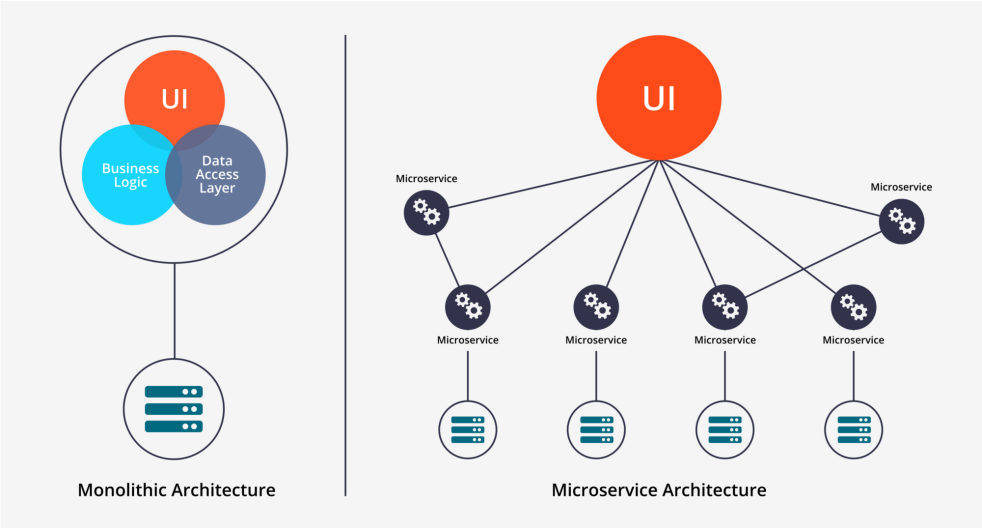
- je malá a nezávislá,
- plní určitý obchodní požadavek,
- službu může spravovat nezávislý tým,
- sestavení a nasazení může probíhat nezávisle na okolních službách,
- má svoje datové úložiště,
- možnost implementace v technologiích odlišných od okolních služeb,
- je škálovatelná nezávisle na okolních službách,
- při změně je nutné nasadit pouze danou službu, ne celý systém. [20]

Microservice architektura s sebou přináší společně se spoustou výhod také spoustu nevýhod, které se mohou projevit až při velkém množství služeb. Mezi takové nevýhody lze zařadit:

- větší nároky na infrastrukturu,
- komunikace mezi službami,
- problémy při nedostupnosti jednotlivých služeb,
- logování a trasování napříč službami,

- nasazování a verzování prostředí,
- transakce napříč službami,
- testování funkčností napříč službami. [20]

Komunikace v takové architektuře probíhá typicky přes RESTové služby nebo jako tzv. *messaging*. Oba typy komunikace jsou detailněji popsané v kapitole 3.4.1, resp. 3.4.2.



Obrázek 3.2: Monolitická vs. microservice architektura [21]

3.2.2 Výběr

Prvním poznatkem, který je třeba uvést, je, že výslednou aplikaci lze potencionálně rozdělit do několika nezávislých celků. V aplikaci určitě musí vzniknout modul, který se bude starat o uživatele a jejich role, modul pro správu úkolů a v neposlední řadě jádro celého systému, modul starající se o grafová data. Poslední jmenovaný modul by šlo považovat za tzv. úzké hrdlo aplikace, jelikož se zde dá očekávat největší zatížení ze všech modulů, a to jak počtem uživatelských přístupů, tak i objemem dat, který bude potřeba zpracovat. Vzhledem k tomuto faktu je možné, že v budoucnu bude požadované škálovat právě tento modul.

Vzhledem k předchozím poznatkům bude výsledná aplikace obsahovat několik nezávislých služeb, které budou založeny na klasické třívrstvé architektuře, tj. každá služba bude obsahovat datovou, aplikační a prezentační vrstvu. Služby budou implementovány v jazyce Kotlin [22] za použití frameworku Spring Boot [23].

Pro zjednodušení stávajícího sestavování a nasazování aplikace, budou služby zabaleny do tzv. *multimodule* projektu.

3.2.3 Popis jednotlivých služeb

Následuje seznam jednotlivých služeb společně s jejich popisem.

3.2.3.1 Comment service

Mezi úkoly této služby bude patřit veškerá funkcionalita, která se týká komentářů, tj. jejich přidávání, úprava a odebírání.

3.2.3.2 Graph service

Tato služba bude zajišťovat veškeré funkce ohledně správy grafových dat, tj:

- fulltextové vyhledávání,
- poskytování grafových dat klientské části,
- správa grafových dat,
- žádosti,
- stížnosti,
- exporty grafových dat.

3.2.3.3 Task service

Tato služba bude mít na starosti správu úkolů, jejich přiřazování a rozdělování nově vzniklých do soukromých nebo týmových schránek.

3.2.3.4 User service

Tato služba bude zajišťovat správu uživatelských účtů a k nim náležících rolí. Dále bude o jednotlivých uživatelích poskytovat základní informace, například v případě potřeby zobrazení autora grafové entity nebo komentáře.

3.3 Webová vrstva

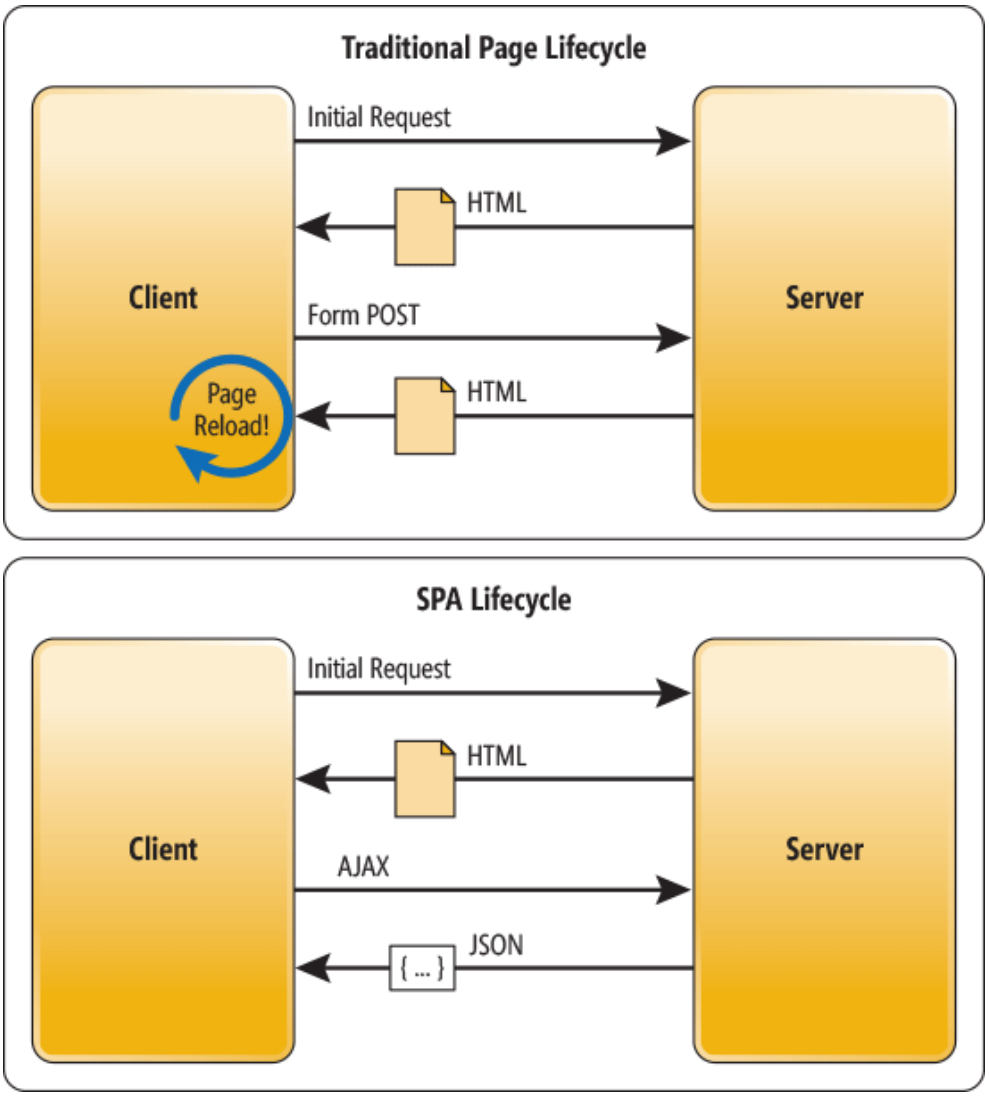
Nyní je potřeba zodpovědět otázku, jakým způsobem bude uživatel se serverovou částí komunikovat.

3.3.1 Průzkum

Při tvorbě webové části aplikace lze rozeznat dva základní návrhové vzory:

- multi-page application (MPA),
- single-page application (SPA).

Porovnání životních cyklů těchto dvou návrhových vzorů je zobrazeno na obrázku 3.3.



Obrázek 3.3: Životní cyklus MPA vs. SPA [24]

3.3.1.1 Multi-page application (MPA)

MPA fungují tradičním způsobem, tj. každý požadavek znamená vygenerování nové stránky na serveru. Objem dat, který je potřeba přenášet během jednotlivých požadavků, je typicky větší než u SPA.

3.3.1.2 Single-page application (SPA)

Jak už jméno napovídá, aplikace používající SPA vzor se skládají pouze z jedné webové stránky, která si s sebou nese množství JavaScript kódu. SPA dovoluje načítat data nezávisle na vykreslování obrazovky a při komunikaci se serverem posílat pouze samotná data (ne celou HTML stránku). [25]

V následujících bodech jsou shrnuty výhody SPA:

- většina zdrojů (HTML, CSS, skripty) je načtena pouze jednou v rámci životního cyklu aplikace,
- se serverovou částí probíhá pouze datová komunikace,
- asynchronní komunikace. [25]

V následujících bodech jsou shrnuty nevýhody SPA:

- komplikovaná SEO optimalizace,
- velké počáteční množství dat, které je potřeba načíst do prohlížeče,
- pro správnou funkci je potřeba JavaScript,
- větší zranitelnost proti XSS útoku. [25]

SPA jsou v dnešní době populární i z důvodu dostupnosti frameworků vyvíjených největšími IT společnostmi či obsáhlé komunitě. Mezi nejznámější zástupce těchto frameworků lze zařadit React.js (Facebook), Angular (Google) či Vue.js. [26]

3.3.2 Výběr

Standardně SPA přinášejí lepší uživatelský zážitek než MPA, jelikož vykreslování změn probíhá plynuleji.

Z tohoto důvodu a z důvodu osobní preference bude klientská část aplikace implementována jako SPA, konkrétně ve frameworku Angular [27].

3.4 Komunikace

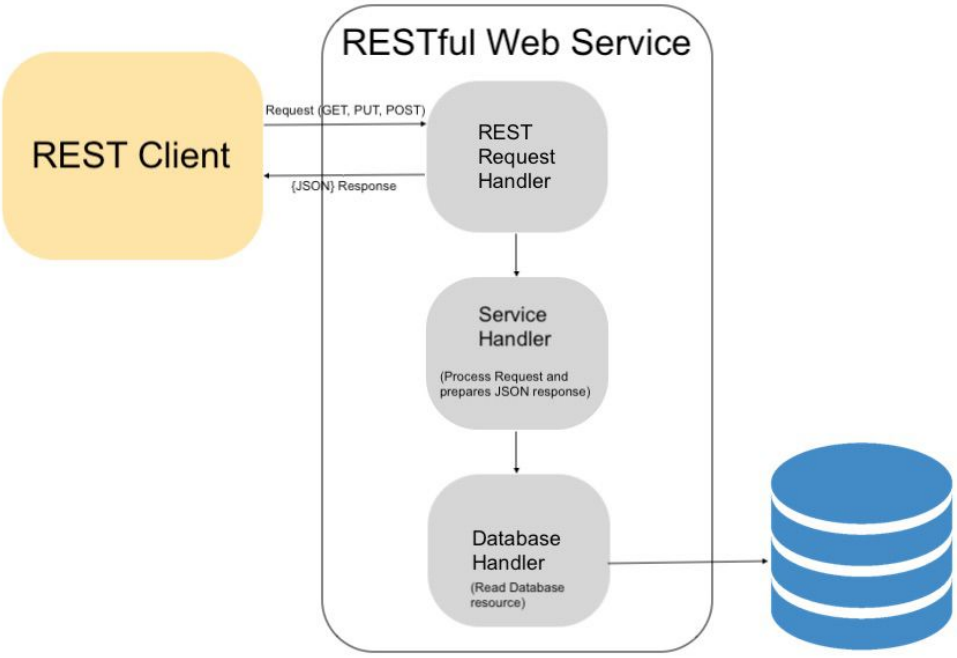
Nyní se nacházíme v situaci, kdy máme dostupných několik nezávislých serverových služeb poskytujících logiku aplikace a klientskou část ve formě Angular projektu. Nyní je třeba zvolit vhodný způsob komunikace mezi těmito komponentami.

3.4.1 REST

„*REST, neboli REpresentational State Transfer, je architektonický styl poskytující standardy pro komunikaci mezi systémy. REST systémy, často zvané RESTful, jsou charakteristické svou bezstavovostí a tím, že oddělují zájmy klientské části a části serverové.*“[28]. Schéma klasického RESTful systému je zobrazeno na obrázku 3.4.

Pro získání přehledu, co to REST vlastně je, je nutné uvést následující principy (viz [29]):

- **klient a server** -– Možnost přepoužití uživatelských rozhraní, zjednodušení serverových komponent a další.
- **bezstavovost** – Požadavky z klienta na server musí obsahovat všechny informace potřebné ke zpracovávání (server neukládá žádný kontext pro jednotlivé požadavky).
- **cachování** – Odpovědi ze serveru obsahují informaci, zda má klient právo přepoužít data pro pozdější ekvivalentní požadavky.
- **uniform interface**
  - **identification of resources** – Datové zdroje jsou identifikovány pomocí URI (Uniform Resource Identifier).
  - **manipulation of resources through representations** – Pokud má klient k dispozici reprezentaci zdroje (včetně metadat), má dostatek informací k manipulaci s tímto zdrojem (například úprava nebo smazání).
  - **self-descriptive messages** – Každá zpráva obsahuje dostatek informací ke svému zpracování (například typ média nebo nastavení cachování).
  - **HATEOAS** neboli Hypermedia As The Engine Of Application State – Klient používá přechody v aplikaci pouze skrz akce, které jsou dynamicky identifikované serverem (hyperlinky) s výjimkou fixně daných vstupních bodů do aplikace.
- **layered system** – Systém lze členit do vrstev, jednotlivé vrstvy ví pouze o vrstvách, s kterými přímo interagují.
- **code on demand (optional)** – REST umožňuje rozšířit klientskou funkcionalitu o applety či různé skripty.



Obrázek 3.4: Architektura RESTful služby [30]

3.4.2 Messaging

Messaging je typem asynchronní komunikace, která se hojně využívá v microservice architekturách. Používá se v případě potřeby komunikace mezi jednotlivými službami. Při použití synchronní komunikace jsou zainteresované služby těsně propojené a během požadavku musí být dostupné po celou dobu jeho trvání. Zmíněný problém asynchronní komunikace svými vlastnostmi řeší. [31]

Tento typ komunikace s sebou ovšem přináší i mnoho problémů, jako namátkou lze uvést následující (předpokládejme implementovaný database per service pattern [32]):

- konzistence dat napříč službami (viz saga pattern [33]),
- dotazy, které potřebují data z více služeb (viz cqrs pattern [34]),
- atomická aktualizace databáze a odeslání zprávy (viz transactional outbox pattern [35]).

U asynchronní komunikace, oproti té synchronní, odesilatel zprávy nemusí znát jejího příjemce. To nám dovoluje jednoduše reagovat na událost v systému ve více službách současně a nezávisle.

3.4.2.1 Motivace

Microservice architektura říká, že každá služba je implementována jako atomická a soběstačná. Tato architektura však vyžaduje i možnost komunikace mezi těmito částmi softwaru. Prvním způsobem komunikace mezi službami je komunikace synchronní, ta avšak, jak bylo zmíněno výše, způsobuje těsné propojení mezi jednotlivými službami, což je ve většině případů nežádoucí. Ovšem neznamená to, že pro synchronní komunikaci zde není místo, vždy je třeba určit pro a proti. [36]

3.4.2.2 Průzkum

V následující kapitole budou dle [36] rozebrány základní typy asynchronní komunikace založené na událostech, tzv. event-driven komunikace.

- **Message queuing** – V tomto modelu jsou zprávy ukládány do fronty. Zprávy může z fronty vyčítat jeden nebo více konzumentů (pokud je zpráva vyčtena, mizí z fronty). Pokud není dostupný žádný konzument, zpráva zůstává ve frontě do té doby, dokud jí některý z nich nevyčte. V tomto případě musí být zpráva vyčtena právě jednou.
- **Publish subscribe** – Tento model ukládá zprávy do tzv. topiků. Konzument může zprávy odebírat z jednoho nebo více topiků. Zdroj zpráv je nazýván jako publisher, spotřebitel jako subscriber, odtud název tohoto modelu. Na rozdíl od modelu message queuing může být zpráva vyčtena více konzumenty.

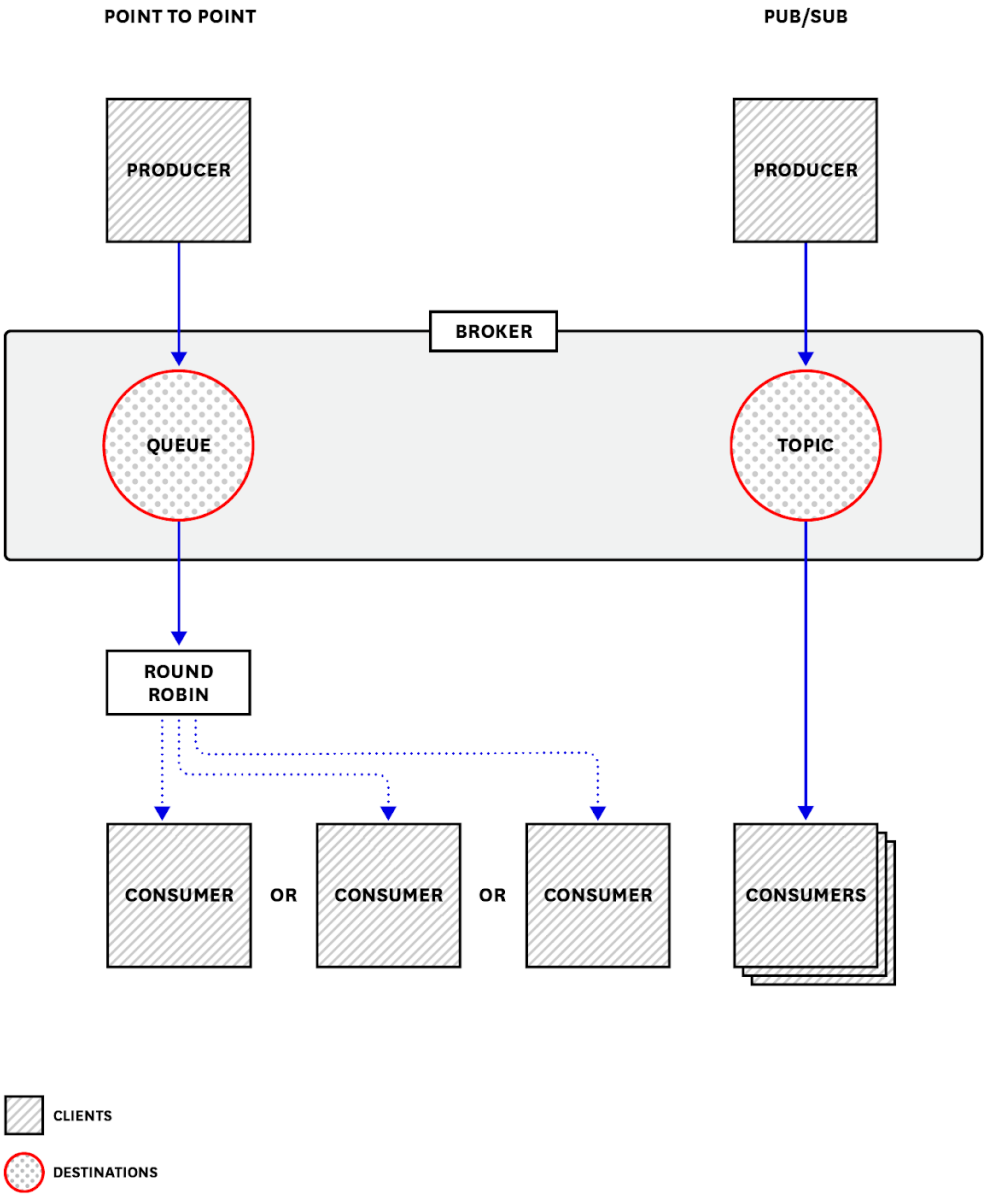
Dnešní trh nabízí několik technologií pro asynchronní komunikaci jako jsou RabbitMQ (Pivotal), ActiveMq (Apache), Google Pub/Sub (Google) nebo Kafka (Apache).

3.4.2.3 Výběr

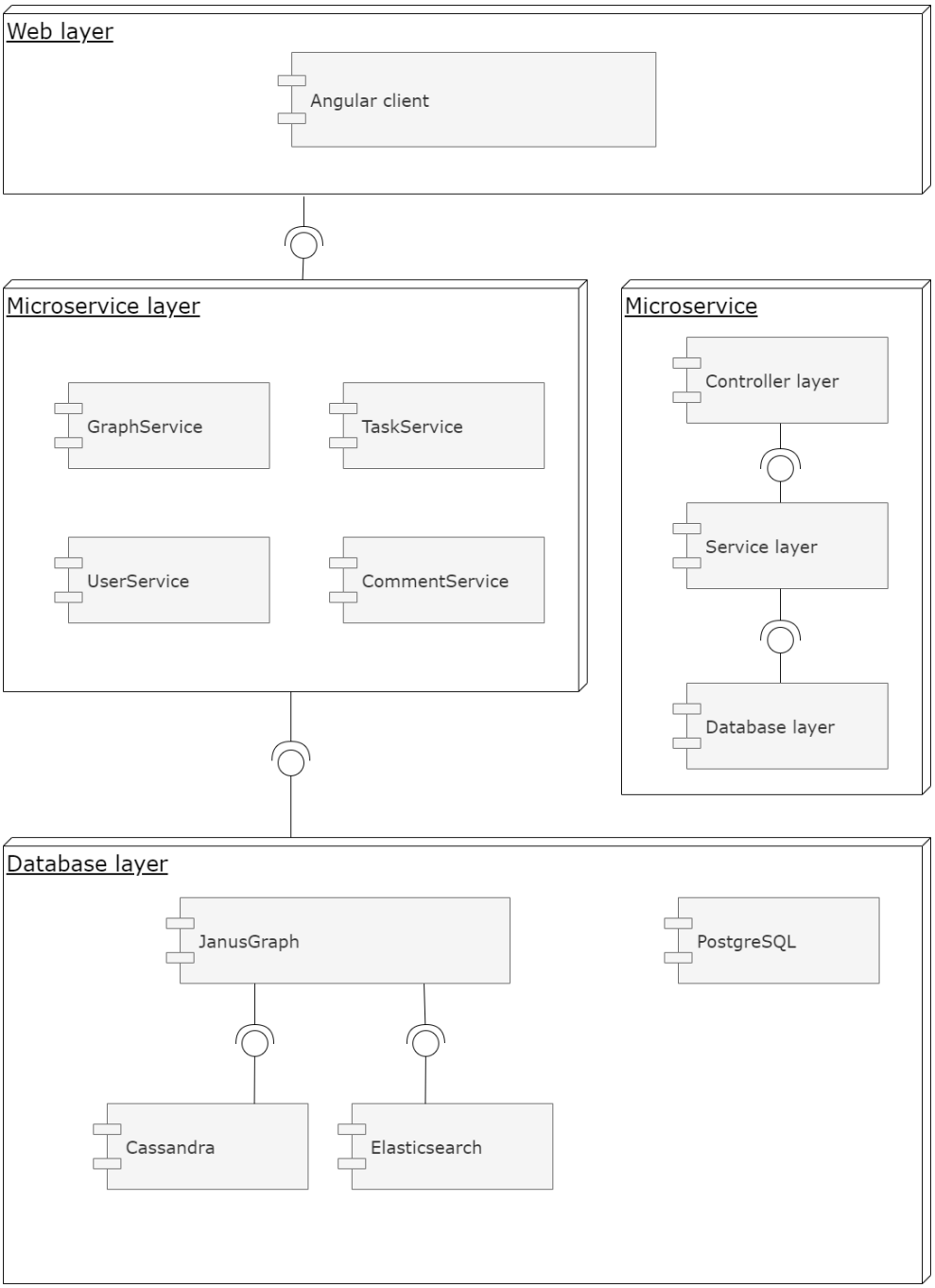
Pro oba typy komunikace, tj. mezi klientem a serverem i mezi jednotlivými službami, bude použita komunikace přes RESTové služby. Mezi klientskou částí a serverem je tento typ komunikace standardním. Co se týká komunikace mezi službami, zde byl tento typ zvolen z důvodu malého výskytu možných asynchronních komunikací a z důvodu dostupné infrastruktury.

3.5 Výsledná architektura

Po navržení výše uvedených částí je možné sestavit diagram komponent, který zobrazuje výslednou architekturu aplikace, viz 3.6.



Obrázek 3.5: Modely asynchronní komunikace [36]



Obrázek 3.6: Architektura aplikace

Kapitola 4

Technologie

V následujících kapitolách budou popsány některé z vybraných technologií použitých při implementaci.

4.1 JanusGraph

JanusGraph je škálovatelná grafová databáze optimalizovaná pro ukládání a dotazování nad rozsáhlými daty. Umožňuje provádět výpočetní operace na několika vzájemně propojených strojích, tzv. cluster. Jedním z primárních benefitů této technologie je škálovatelnost zpracování grafových operací nebo analytických dotazů. [10]

4.1.1 Architektura

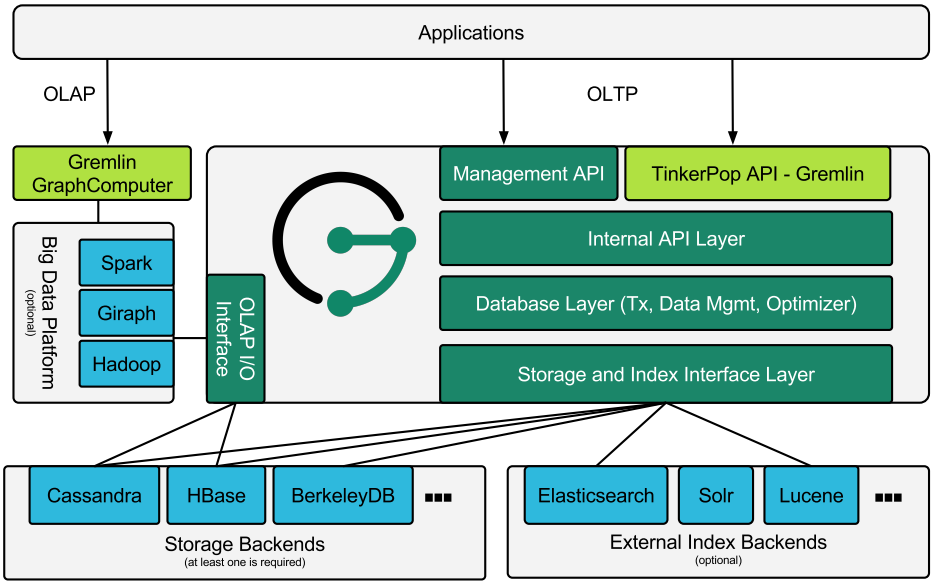
Architektura této databáze je založena na robustních rozhraních pro ukládání dat, indexování dat a pro klientský přístup. Modulární architektura umožňuje propojení s velkým množstvím technologií nebo možnost rozšíření o nové.

JanusGraph lze považovat za prostředníka mezi aplikací, úložištěm dat a indexovacím nástrojem. Ve standardní verzi jsou podporovány tyto nástroje:

- Datové úložiště:
  - Apache Cassandra,
  - Apache HBase,
  - Oracle Berkeley DB Java Edition,
  - *in-memory.*
- Indexovací nástroje:
  - Elasticsearch,
  - Apache Solr,

– Apache Lucene.

Detailnější informace lze získat v samotné dokumentaci, viz [37].



Obrázek 4.1: JanusGraph – architektura [37]

4.1.2 Podporované technologie

JanusGraph přichází s nativní podporou grafového modelu, který definuje Apache TinkerPop (viz 4.1.2.1). Komunikace aplikací s JanusGraph databází probíhá typicky pomocí jazyka Gremlin (viz 4.1.2.3), který je dalším standardem Apache TinkerPop.

4.1.2.1 Apache TinkerPop

Apache TinkerPop je framework pro grafové výpočty dostupný nejen pro grafové databáze (OLTP), ale i pro analytické systémy (OLAP) postavené nad grafovými daty. Základním stavebním kamenem tohoto frameworku je tzv. Directed Property Graph. [38]

4.1.2.2 Directed property graph

Tento typ grafu je dle [38] založen na třech základních blocích:

- **Vertex**, neboli uzel, je element, který odkazuje na konkrétní objekt (například osoba nebo firma). Každý uzel má svůj jedinečný identifikátor (standardně poskytnutý databází) a popisek definovaný uživatelem. Volitelně může obsahovat vlastnosti (properties).
- **Edge**, neboli hrana, reprezentuje vazbu mezi dvěma uzly. Hrana je v tomto případě orientovaná, tj. záleží na směru. Hrana má svůj jedinečný identifikátor, vstupní a výstupní uzel, případně opět volitelné vlastnosti.
- **Properties**, neboli vlastnosti, jsou dodatečné informace, které umožňují popisovat jak uzel, tak hranu. Tyto vlastnosti jsou standardně uváděny ve formátu klíč-hodnota.

4.1.2.3 Gremlin

Gremlin je jazyk pro provádění operací nad grafovými daty. Tyto operace se nazývají *traversals* a každá taková operace se skládá z posloupnosti dílčích kroků. Dílčí krok může být jedním z těchto typů:

- map – transformace dat,
- filter – filtrování dat,
- sideEffect – dovoluje provádět akce bez změny dat.

Gremlin je postaven na filozofii „write once, run anywhere“, tj. Gremlin výraz lze provést nejen na každém systému, který podporuje TinkerPop framework, ale také může být vyhodnocen buď jako databázový, nebo jako analytický dotaz. Tento přístup přináší výhodu, kdy uživateli stačí jeden jazyk pro účely grafových i analytických dotazů.

Použití tohoto jazyku eliminuje situaci zvanou vendor-lock, jelikož tento jazyk podporuje většina grafových databází. Aplikace tudíž může být snadno zmigrována na databázi jinou. [39]

4.1.3 Nasazení

Komunikace JanusGraph databáze s konkrétní aplikací může probíhat ve dvou rozdílných módech, které jsou blíže specifikovány v kapitole 4.1.3.1, resp. 4.1.3.2.

4.1.3.1 JanusGraph Server

JanusGraph Server je pojmenování pro kombinaci technologií JanusGraph a Gremlin Server. JanusGraph databáze v tomto módu běží jako nezávislý server, který odpovídá na Gremlin dotazy jednotlivých aplikací. Jako výčet výhod tohoto módu lze uvést tyto vlastnosti:

- všechny dotazy jsou centralizované,
- globální správa schémat, indexů a jiných objektů,
- zpracování každého Gremlin dotazu v samostatné transakci,
- škálovatelnost.

Na druhou stranu, toto použití znemožňuje používat JanusGraph API přímo v aplikacích, tj. například vytváření schématu nebo manuální správa transakcí není přímo z aplikace možná. [40]

4.1.3.2 Embedded JanusGraph

Aplikace postavené na JVM jazyku (Java, Kotlin či Scala) mohou JanusGraph používat jako knihovnu. V tomto případě aplikace s JanusGraph databází komunikuje na přímo a nemusí volat nezávislou službu. Tento přístup eliminuje infrastrukturní režii, na druhou stranu však znemožňuje škálovat instanci databáze separátně od dané aplikace. Umožňuje však používat JanusGraph API zmíněné v kapitole 4.1.3.1. [40]

4.1.4 Hromadný import

JanusGraph databáze podporuje mimo klasické transakční zpracování i hromadné načítání dat. Před samotným hromadným načítáním je vhodné provést několik konfigurací.

První z nich je zapnutí konfigurační proměnné „storage.batch-loading“. Na základě toho JanusGraph zakáže kontrolu konzistence dat a zamykání, jinými slovy to znamená, že považuje načítaná data jako by byla v souladu se schématem grafu a kvůli zvýšení výkonu zakáže vlastní kontroly. Při zapnutí této proměnné je také doporučeno vypnout automatické vytváření schématu pomocí proměnné „schema.default“.

Další možnou optimalizací je zvětšení bloku pro načítané identifikátory, které jsou přiřazovány ke každému přidanému uzlu nebo hraně. JanusGraph instance získává tyto identifikátory po blocích, což je drahá operace, jelikož se musí zajistit jedinečnost identifikátorů napříč instancemi. Při hromadném načítání se dá očekávat mnoho vložení do databáze, proto je vhodné zvětšit velikost bloku, aby se zmenšil počet jeho načítání.

Samotný hromadný import dat lze provést například pomocí nástroje Data importer [41] od společnosti IBM nebo jako skript přes Gremlin konzoli [42].

4.2 Elasticsearch

„*Elasticsearch je distribuovaný, open source vyhledávací a analytický nástroj pro všechny typy dat, včetně textových, numerických, geoprostorových, strukturovaných a nestrukturovaných. Elasticsearch je postaven na Apache Lucene*

a byl poprvé uveden na trh v roce 2010 společností Elasticsearch N.V. (nyní známá jako Elastic). Elasticsearch je typický svým jednoduchým REST API, distribuovanou povahou, rychlostí a škálovatelností. Tvoří nedílnou součást tzv. Elastic Stack (ELK) – Elasticsearch, Logstash a Kibana.“ [11]

Mezi hlavní přednosti této technologie patří zejména rychlost vyhledávání i samotného indexování, distribuovatelnost, škálovatelnost a možnost integrace s nástroji Logstash a Kibana.

4.2.1 Použití

Elasticsearch umožňuje zpracovávat data z různých zdrojů, což z něj dělá velmi univerzální nástroj. Mezi vzorové příklady použití patří:

- fulltextové vyhledávání,
- analýza logů,
- analýza metrik (výkonnost, dostupnost a další),
- analýza a vizualizace geoprostorových dat.

4.2.2 Index

Elasticsearch index je kolekce spolu souvisejících dokumentů, kde jsou uloženy data ve formátu JSON. Elasticsearch používá datovou strukturu zvanou invertovaný index, která je navržena tak, aby umožňovala rychlé fulltextové vyhledávání. Invertovaný index uchovává ke každému unikátnímu slovu, které se vyskytuje v libovolném dokumentu, seznam všech dokumentů, kde se dané slovo vyskytuje. Během procesu indexování Elasticsearch nejen že ukládá samotné dokumenty, ale vytváří i invertovaný index, aby následně bylo možné data v dokumentech prohledávat téměř v reálném čase. [11]

4.2.3 Logstash

Nástroj Logstash se používá k agregaci dat, zpracování dat a jejich následnému odeslání do Elasticsearch, tj. umožňuje přijímat data z více zdrojů současně a provádět nad nimi transformace ještě před samotným indexováním v Elasticsearch. [11]

4.2.4 Kibana

Kibana je nástroj pro vizualizaci a správu dat pro Elasticsearch, který poskytuje real-time grafy různých typů (histogramy, spojnicové grafy, výsečové grafy nebo mapy).

Obrázek 4.2: Elastic Stack [43]

4.3 Spring Boot

Spring framework poskytuje komplexní programovací a konfigurační model pro tvorbu enterprise aplikací založených na jazyku Java. Obsahuje širokou podporu infrastruktury pro vývoj samotné aplikace. Vývojáři se tak mohou soustředit primárně na obchodní logiku celého řešení. [44]

Spring Boot z frameworku Spring vychází a navíc s sebou přináší možnost rychlého a snadného založení funkční kostry aplikace. Framework poskytuje několik modulů, které lze přímo použít, a to bez nutnosti rozsáhlé konfigurace. Mezi základní poskytované moduly patří:

- spring-boot-starter-data-jpa,
- spring-boot-starter-security,
- spring-boot-starter-web,
- spring-boot-starter-test. [23]

4.4 Angular

Angular je framework k tvorbě SPA klientských aplikací v jazyce TypeScript.

Základním stavebním blokem každé Angular aplikace je tzv. NgModule. Každá aplikace má minimálně jeden tento modul (kořenový), který umožňuje spuštění celého systému, a obvykle i několik modulů funkčních. [45]

4.4.1 NgModule

Ve světě Angular frameworku lze význam slova modul definovat jako mechanismus, jak propojit elementy typu *component*, *directive* a *pipe*. Pomocí jednotlivých modulů lze pak složit celou aplikaci. Další vlastností modulů je

to, že s jejich pomocí je možné stavět bloky, které zapouzdřují implementační detaily a pro ostatní moduly vystavují pouze API. [46]

Na následujícím příkladu lze zmiňovanou vlastnost zapouzdření demonstrovat. Uvažujme definici modulu dle výpisu kódu 1, který má na starosti komentáře. Modul vnitřně používá tři komponenty. První komponenta má na starosti vykreslení jednoho komentáře, druhá formulář pro přidání nového komentáře a třetí předchozí komponenty slučuje dohromady a poskytuje plnohodnotnou sekci pro komentáře. Tato komponenta je jako jediná v deklaraci modulu uvedená v sekci *exports*. Ostatním modulům je tedy dostupná pouze tato komponenta, zbylé dvě jsou do tohoto modulu zapouzdřené.

```
@NgModule({
    declarations: [
        CommentsComponent,
        CommentItemComponent,
        CommentInputComponent,
        ...
    ],
    exports: [
        CommentsComponent
        ...
    ],
    imports: [
        CommonModule,
        ...
    ]
})
export class CommentModule {
}
```

Zdrojový kód 1: Angular NgModule

4.4.2 Component

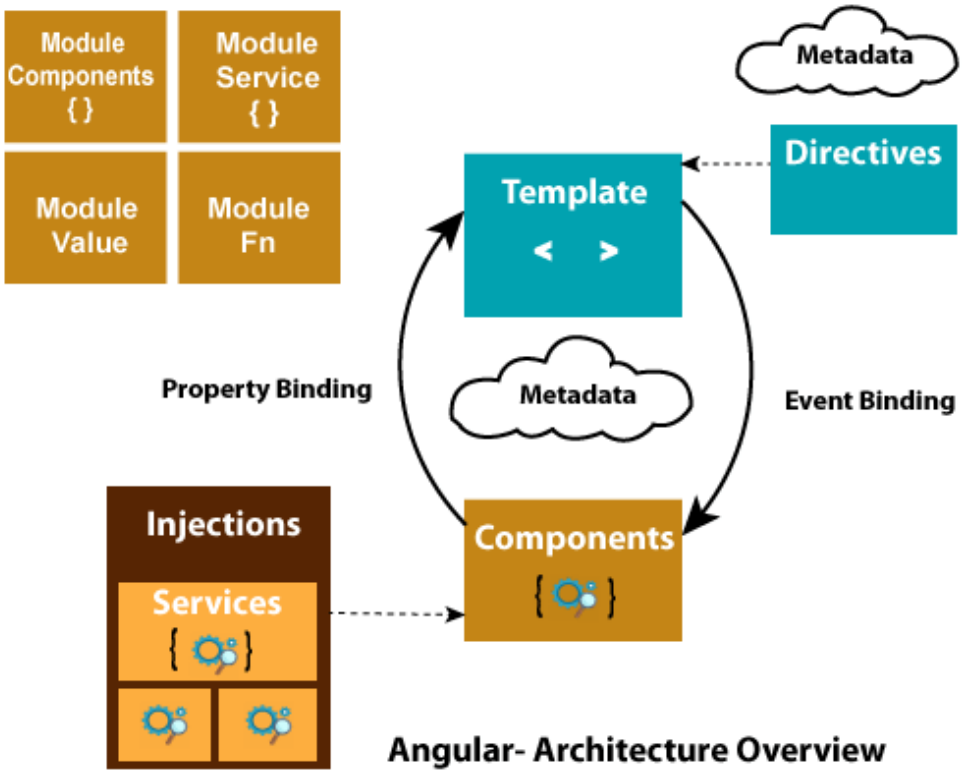
Element typu *component* definuje třídu, která obsahuje aplikační data a logiku pro konkrétní HTML šablonu. Jednotlivé stránky výsledné aplikace jsou pak typicky nějakou hierarchií těchto elementů. [47]

4.4.3 Directive

Directive element je rozdělen na dva typy. První je atributový, který mění vzhled nebo chování konkrétní položky v HTML DOM [48]. Druhý pak strukturální, který je zodpovědný za rozložení HTML, přidávání, odebírání nebo manipulaci s prvky [48].

4.4.4 Service

Třetím základním stavebním blokem Angular aplikace je element *service*. Tyto elementy se používají pro aplikační logiku, která není spojená s konkrétní HTML šablonou. V aplikaci jsou standardně dostupné přes *dependency injection*. [49]



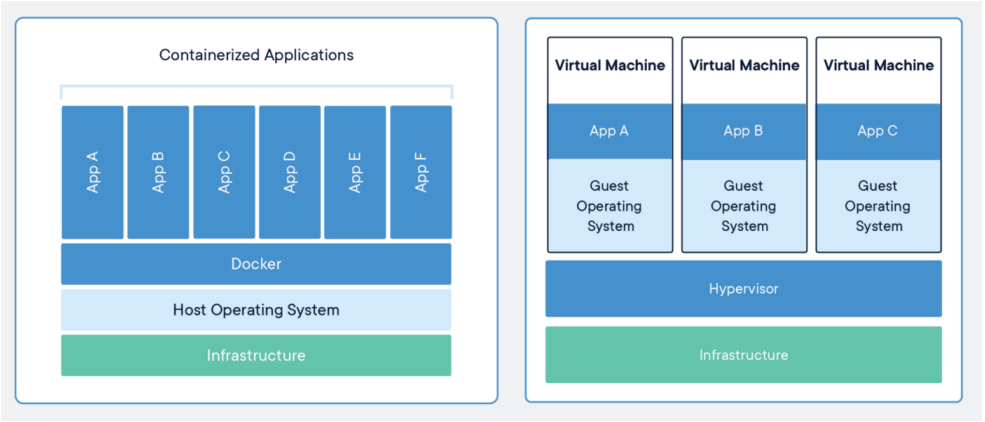
Obrázek 4.3: Angular architektura [50]

4.5 Docker

Docker je nástroj pro virtualizaci na úrovni operačního systému, známou také pod pojmem kontejnerizace. Následující definice zcela vystihuje koncept kontejnerů:

„A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.“ [51]

Zásadní rozdíl mezi klasickou virtualizací a Docker kontejnery je sdílení hostitelského operačního systému. V případě virtualizace každý virtuální stroj vyžaduje samostatný operační systém, který je následně schopen komunikovat s hostitelským operačním systémem pomocí hypervisoru. Docker kontejnery naproti tomu využívají přímo funkce hostitelského operačního systému. [52]



Obrázek 4.4: Docker kontejner vs. virtuální stroj [51]

Jednou z hlavních výhod této technologie je možnost sdílení prostředků, což se ve velké míře uplatňuje ve světě cloud technologií. Více informací o této technologie lze najít na oficiálních stránkách, viz [53].

Kapitola 5

Implementace

Výsledná aplikace je rozdělena do projektů:

1. **API** – definice API mezi klientskou a serverovou částí,
2. **commons-lib** – společné funkcionality pro služby projektu services,
3. **services** – serverové služby,
4. **UI** – klientská část.

V následujících kapitolách jsou popsány vybrané technické detaily implementace.

5.1 Zabezpečení

Tato kapitola se věnuje popisu implementace typů zabezpečení, které jsou přímo vyžadovány v nefunkčních požadavcích (viz 2.2) nebo jsou nutné kvůli technologiím, které byly při implementaci použity.

5.1.1 Autentizace a autorizace

Autentizace a autorizace jsou základní technikou k zabezpečení aplikace. Autentizace má za úkol zjistit, zda uživatel je opravdu ten, za koho se vydává, na rozdíl od autorizace, která má za úkol ověřovat, zda daný uživatel disponuje právem přistupovat na určité zdroje. K tomuto typu zabezpečení byl použit JWT (Json Web Token) a to hlavně z důvodu bezstavovosti serverové části, soběstačnosti (token může obsahovat libovolné informace o přihlášeném uživateli) a kompaktnosti (token lze posílat v HTTP hlavičce).

Pro získání přihlašovacího tokenu je nutné, aby uživatel zadal validní přihlašovací údaje, na základě nich server uživatele ověří a vygeneruje token, který pošle zpět klientské části aplikace. Klientská část si ho uloží a následně tento token přidává ke každému odchozímu HTTP požadavku ve formě HTTP

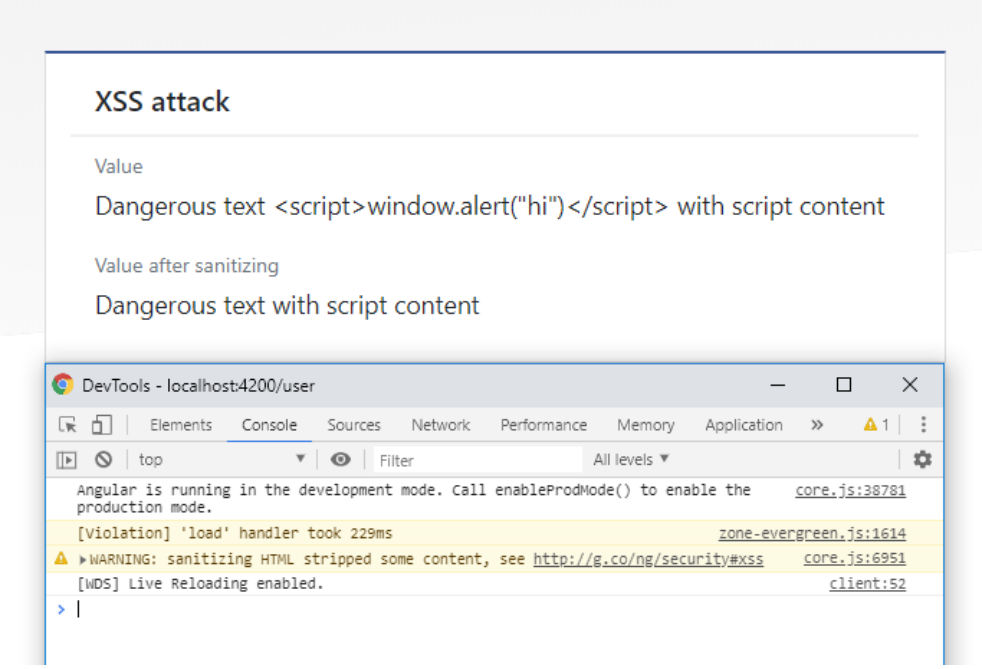
hlavičky Authorization. Pro vkládání tokenu do hlavičky je použit návrhový vzor interceptor.

Na serverové části jsou všechny zdroje ve výchozím stavu pro uživatele nedostupné. K povolování jednotlivých zdrojů pro určité uživatelské role se používá princip zvaný *whitelisting*. Tento přístup zamezuje potencionálnímu zapomenutí zabezpečit některý ze zdrojů a tím zanést do aplikace bezpečnostní díru.

5.1.2 Cross site scripting

Cross site scripting (XSS) je typ útoku, který umožňuje útočníkovi vložit škodlivý kód na webové stránky. Pomocí tohoto kódu pak může útočník získat různá uživatelská data, například přihlašovací údaje, nebo provádět akce pod identitou daného uživatele.

Angular pro mitigaci tohoto útoku používá mechanismus, kdy všechny hodnoty, které mají být vloženy do HTML DOM, jsou nejdříve zkontrolovány a případně změněny tak, aby byly bezpečné. Na obrázku 5.1 je vidět příklad takového chování, v konzoli prohlížeče lze najít záznam, který upozorňuje na to, že hodnota musela být před vložením do HTML DOM změněna. [54]



Obrázek 5.1: Angular – cross site scripting

5.1.3 SQL injection

Tento typ útoku je založen na vložení SQL dotazu jakožto vstupních dat od koncového uživatele. Pokud aplikace tento útok umožňuje, útočník je schopen číst citlivá data, upravovat data nebo provádět jiné nežádoucí akce. Na výpisu kódu 2 je demonstrována zranitelnost aplikace při jejím nedostatečném zabezpečení. Útočník v tomto případě dokáže získat přístup k datům všech uživatelů.

```
fun findById(id: String): List<User> {
    val sql = "select * from user where id = '$id'"
    return datasource.executeQuery(sql)
}

// find user with id XYZ
loadData("XYZ")

// find all users
loadData("XYZ' or '1' = '1")
```

Zdrojový kód 2: SQL injection

K eliminaci tohoto útoku bylo docíleno použitím knihovny Spring Data JPA [55], která umožňuje používat API, které vývojáře odstiňuje od používání ručně skládaných SQL dotazů. Namísto toho používá princip dotazů parametrizovaných. Na výpisu kódu 3 lze vidět vzorový SQL dotaz, který knihovna vyprodukuje. Místo zástupky „?“ se doplní uživatelem vyplněný výraz už jako prostá hodnota, nikoliv jako část dotazu.

```
// SQL query from Spring Data JPA
select * from user where id=?
```

Zdrojový kód 3: Spring data a SQL injection

5.1.4 Gremlin query injection

Stejně tak jako v předchozím případě, i u dotazovacího jazyka Gremlin musíme vyřešit možné bezpečností riziko podobného rázu – *query injection*.

5.1.4.1 Typy Gremlin dotazů

Dle [56] umožňuje Gremlin Server, který byl při implementaci použit jako součást JanusGraph serveru, provádění dotazů, které jsou založené buď na textové (skripty), nebo na bytecode reprezentaci. Preferovaným způsobem jsou

dotazy založené na bytecode reprezentaci, jelikož jsou psány ve stejném programovacím jazyce jako zbytek aplikace, tedy je možné provádět jejich kontrolu při sestavování aplikace nebo využívat podporu vývojového prostředí, a umožňují cachování na straně serveru. Na rozdíl od skriptů, které jsou prostými řetězci. Na výpisu kódu 4 je zobrazené použití obou typů dotazů.

```
// string based
val cluster: Cluster = Cluster.open()
val client: Client = cluster.connect()

    // with possible query injection
    client.submit("g.V().has('name','$name')").all().get()

    // with parameters
    val params = mapOf("name" to name)
    client.submit("g.V().has('name',name)", params).all().get()


// bytecode based
val g: GraphTraversalSource = traversal().withRemote(...)
g.V().has("name", name).toList()
```

Zdrojový kód 4: Gremlin Server – typy dotazů

5.1.4.2 Mitigace

Z výše zmíněných důvodů byla použita komunikace založená na bytecode dotazech. Tato volba rovnou eliminovala i možné *query injection* – „*Gremlin injection should not be possible with bytecode based traversals, because bytecode traversals will treat all arguments as literal values.*“ [57]

5.2 API projekt

Vzhledem k tomu, že klientská část aplikace je implementována separátně od té serverové a zároveň jsou použité rozdílné technologie, je nutné vyřešit problém, jak mít synchronizované definice objektů, se kterými bude probíhat vzájemná komunikace (tzv. *transfer object*). Naivním řešením by mohlo být udržování definic odděleně jak na serverové, tak na klientské části. Tento přístup však není dlouhodobě udržitelný a je jen otázkou času, kdy se definice začnou na jednotlivých částech lišit. V tomto okamžiku nezbývá nic jiného než použít nástroj, který je schopný tyto definice udržovat v kompatibilní verzi pro obě technologie současně.

5.2.1 OpenAPI specifikace

OpenAPI specifikace, dříve známá jako Swagger, je formát popisu REST API. Umožňuje popisovat jak *endpointy* rozhraní, včetně jejich různých parametrů, tak i definice objektů, které se v tomto rozhraní používají. Rozhraní lze popisovat ve formátu YAML nebo JSON, tudíž výsledný popis API lze dále strojově zpracovávat – například generování dokumentace nebo kódu. [58]

5.2.2 OpenAPI Generator

K vyřešení výše zmíněného problému byl použit nástroj OpenAPI Generator [59]. Tento nástroj dokáže z předpisu ve formátu OpenAPI (viz 5.2.1) vygenerovat definice objektů v různých programovacích jazycích. V tomto případě byl použit generátor pro jazyk Kotlin, kdy je vytvořen JAR soubor, a generátor pro Typescript, kdy je vytvořen npm (Node Package Manager) balík, který lze použít v Angular aplikaci. Pokud jsou importovány v klientské i serverové části aplikace stejné verze artefaktů, je konzistence definic objektů zajištěna.

Mimo to je nástroj schopen generovat i rozhraní pro RESTové služby. Primárně z důvodu tvorby dokumentace RESTového rozhraní celé aplikace (OpenAPI předpis je sám o sobě dokumentací) byl tento přístup v implementační fázi realizován.

Nástroj taktéž umožňuje k libovolnému parametru nebo atributu generovat různá omezení (délka řetězce, formát a jiné). Každý požadavek, který na serverovou část přijde, je proti těmto omezením validován.

5.2.3 Struktura API definice

K podpoře přehlednosti byl vytvořen vlastní nástroj, který umožňuje rozdělit celou definici API do separátních souborů. Nástroj před generováním nahradí zástupky obsahem jednotlivých souborů a tím vytvoří definici platnou dle OpenAPI specifikace, kterou následně předloží generátoru. Struktura dílčích definic má následující podobu. Základem je soubor *index.yaml* (viz výpis zdrojového kódu 5), který obsahuje odkazy na soubory s jednotlivými dílčími definicemi. Definice v těchto separátních souborech, jejichž příklad je uveden na výpisu kódu 6 (*transfer object*) a 7 (*endpoint*), se pak mohou odkazovat na jiné způsobem, který umožňuje OpenAPI specifikace pro definice, které se nacházejí ve stejném souboru.

5.3 HATEOAS

Tato kapitola se věnuje přínosu použití principu HATEOAS. Tento princip byl definován již dříve v kapitole 3.4.1.

```yaml
openapi: 3.0.0
info: ...
servers: ...

paths:
  /search:
    $ref: "paths/search/fulltextSearch.yaml"

components:
  parameters:
    $ref: "params/urlParams.yaml"
  requestBodies:
    $ref: "params/requestBody.yaml"
  schemas:
    SearchResult:
      $ref: "definitions/search/SearchResult.yaml"
    SearchPersonRecord:
      $ref: "definitions/search/SearchPersonRecord.yaml"
```

Zdrojový kód 5: OpenAPI – index.yaml

```yaml
type: "object"
required:
  - persons
  - links
properties:
  persons:
    type: "array"
    items:
      $ref: "#/components/schemas/SearchPersonRecord"
  links:
    type: "object"
    properties:
      nextPage:
        $ref: "#/components/schemas/LinkTO"
```

Zdrojový kód 6: OpenAPI – *transfer object*

5.3.1 Motivace

Při budování webové aplikace je nutné dobře promyslet, jakým způsobem budeme řídit navigaci v rámci aplikace, dostupnost prvků k provedení různých uživatelských akcí a podobně.

Pokud se vydáme cestou, kdy server poskytuje pouze data, musí se o na-

```
get:
  summary: "Fulltext search"
  operationId: fulltextSearch
  parameters:
    - in: query
      name: query
      required: true
      schema:
        type: string
        ...
      description: "search query"
  responses:
    '200':
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/SearchResult"
      description: "Records matching query"
```

Zdrojový kód 7: OpenAPI – *endpoint*

vigaci v aplikaci a dostupnost prvků starat klientská část. Tímto přístupem však tuto část zanášíme obchodní logikou, kterou primárně směřujeme na část serverovou. Další problém může přinést tvorba nové klientské části, všechna logika by musela být zduplikována do nového řešení.

Druhým možným řešením je situace, kdy dostupné akce řídí přímo server. V tento okamžik nastupuje na scénu princip HATEOAS. Při použití tohoto principu je standardní odpověď na REST volání (viz výpis zdrojového kódu 8) rozšířena o dodatečné informace ve formě dostupných akcí. Vzorová odpověď s tímto rozšířením je zobrazena na výpisu kódu 9. Na základě těchto dodatečných informací zbývá na klientské části implementovat pouze jednoduché rozhodování – pokud je akce v odpovědi dostupná, zpřístupni jí uživateli.

Přínosem principu HATEOAS je také *loose coupling*. Pokud chceme komunikovat s nějakou RESTful službou, musíme znát adresy, které můžeme provolat. Pokud nepoužijeme tento princip, je výsledná klientská aplikace úzce svázaná se strukturou těchto adres. [60]

5.3.2 Realizace

Ve výsledné aplikaci bylo docíleno situace, kdy klientská část zná pouze jednu adresu na každou službu. Tato situace by se dala ještě zlepšit vystavením proxy, která by všechny služby zastřešovala, v této situaci by musela znát adresu právě jednu. Klientská část aplikace před samotným startem provolá jednotlivé zaregistrované služby a získá od nich seznam počátečních dostupných

```
{
    "id": 1,
    "givenName": "XYZ",
    "familyName": "XYZ"
}
```

Zdrojový kód 8: Standardní REST odpověď

```
{
    "id": 1,
    "givenName": "XYZ",
    "familyName": "XYZ",
    "links": {
        "self": {
            "allow": "GET",
            "href": "http://localhost:8080/user/1"
        },
        "update": {
            "allow": "PUT",
            "href": "http://localhost:8080/user/1"
        },
        "data": {
            "allow": "GET",
            "href": "http://localhost:8080/user/1/data"
        }
    }
}
```

Zdrojový kód 9: REST odpověď při použití principu HATEOAS

akcí. Požadavky na tyto počáteční akce se mohou v průběhu používaní aplikace opakovat, například v případě změny kontextu přihlášeného uživatele. Ostatní uživatelské akce jsou zasílány jako součást poskytovaných dat.

5.4 Angular module lazy loading

Jak již bylo zmíněno v předchozích kapitolách, klientská část je implementována jako SPA s použitím frameworku Angular. Samotná Angular aplikace je rozdělena do následujících modulů:

- comment,
- core,
- graph-display,

- graph-entity,
- history,
- layout,
- search,
- task,
- user.

Význam slova modul v kontextu Angular aplikace byl již definován v kapitole 4.4.1.

Zde je dobré si povšimnout, že za předpokladu, že běžní uživatelé budou pouze zobrazovat grafová data, nebudou pro ně moduly starající se například o komentáře nebo úkoly relevantní. Pokud by ovšem takový uživatel přišel do této aplikace, do prohlížeče by se mu načetly moduly všechny. Jejich podstatnou část by však uživatel za celou dobu používání aplikace nevyužil. V tomto případě tak zbytečný přenos dat o nepotřebných modulech může vést k prodlevám při načítání aplikace. Tato situace lze vyřešit tzv. *module lazy loading* přístupem. K demonstraci tohoto přístupu je nutné porozumět, jak funguje směrování v tomto frameworku.

Angular směrování je mechanismus, jak lze přecházet mezi jednotlivými obrazovkami aplikace. Na výpisu zdrojového kódu 10 je vidět základní konfigurace, která je intuitivní – na adresu */home* se zobrazí `HomeComponent`, na adresu */data* zase `DataComponent`.

Pokud upravíme definici směrování dle výpisu zdrojového kódu 11, data k modulu `DataModule` budou staženy až ve chvíli, kdy se uživatel pokusí přistoupit na adresu */data/***.

```
const appRoutes: Routes = [
  { path: 'home', component: HomePageComponent },
  { path: 'data', component: DataComponent }
];

NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
    ...
  ],
  ...
})
export class AppModule { }
```

Zdrojový kód 10: Angular směrování

```
const appRoutes: Routes = [
    { path: 'home', component: HomePageComponent },
    {
        path: 'data',
        loadChildren: () => import('./modules/data/data.module')
                            .then(m => m.DataModule)
    }
];
```

Zdrojový kód 11: Angular lazy loading

5.5 Komunikace s JanusGraph databází

Při implementaci bylo potřeba vyřešit mapování mezi objekty a odpověďmi z databáze JanusGraph.

Na trhu existuje několik knihoven, které tento problém řeší, avšak buď nepodporují samotnou databázi JanusGraph, nebo jejich požadavkem je použití této databáze v módu *embedded* (viz 4.1.3.2). Příkladem těchto knihoven jsou Spring Data Gremlin [61], Kotlin Gremlin Ogm [62] nebo Ferma [63].

Z těchto důvodů byl implementován vlastní algoritmus mapování.

5.5.1 VertexRepository

Tento typ repository tříd se stará o databázové operace nad objekty, které se do databáze ukládají jako uzel nebo množina uzlů. Příkladem, kdy se objekt uloží do více uzlů je firma, ta mimo jiné obsahuje i vnořený objekt typu adresa, který reprezentuje sídlo firmy. Tento vnořený objekt se do databáze ukládá jako samostatný uzel.

Implementace je založená na reflexi. Při startu aplikace se ke každé třídě, která se má ukládat jako uzel/množina uzlů zjistí všechny atributy typu, jež má být v databázi reprezentován jako samostatný uzel. Ke všem takovým atributům se mimo jiné uloží reference na getter metodu, typ uzlu a název vazby, kterou mají být uzly spojené. Při databázových operacích se pak těchto uložených informací využívá, dle typu uzlu se najde příslušná repository třída a ta poskytne Gremlin dotaz pro zpracování odpovídajícího vnořeného objektu. Všechny tyto dotazy se následně pro zajištění transakčního zpracování (JanusGraph Server provádí každý Gremlin dotaz v samostatné transakci) spojí do jednoho. Vzorová definice třídy uzlu je zobrazena na výpisu kódu 12, definice repository třídy pak na výpisu kódu 13.

5.6 Fulltextové vyhledávání

Fulltextové vyhledávání využívá nástroj Elasticsearch a v aplikaci jej zajišťuje třída `FulltextSearchProvider`.

```
@Vertex(type = VertexType.COMPANY)
data class Company(
        val id: String,
        @FulltextSearchProperty
        val officialName: String,
        @NestedVertex(name = "headquarters_address")
        val headquarters: Address
) : BaseVertex
```

Zdrojový kód 12: Definice třídy uzlu

```
abstract class CompanyRepository :
            AbstractVertexRepository<Company>(Company::class)
```

Zdrojový kód 13: Definice repository třídy

Třída `FulltextSearchProvider` při startu aplikace zjistí pomocí reflexe, přes které atributy má vyhledávání probíhat. Označení takových atributů je realizováno pomocí anotace. Vzorový příklad použití je zobrazen na výpisu kódu 14. Při samotném vyhledávání se pak už jen vytvoří Gremlin dotaz pro fulltextové vyhledávání právě přes označené atributy.

```
data class Person(
        val id: String,
        @FulltextSearchProperty
        val givenName: String,
        @FulltextSearchProperty
        val familyName: String
)
```

Zdrojový kód 14: Anotace pro fulltextové vyhledávání

Ke správnému fungování je třeba vytvořit v databázi JanusGraph index, který bude obsahovat názvy všech anotací označených atributů. Příkaz k vytvoření takového indexu je zobrazen na výpisu kódu 15.

5.7 Poskytování relevantních dat

Jak již bylo zmíněno v kapitole 2.4, grafová entita se může nacházet v několika stavech. Běžným uživatelům se standardně zobrazují pouze entity, které jsou aktivní, na rozdíl od správců aplikace, kterým se zobrazují entity ve všech stavech. Vyhledávání grafových entit je realizováno pomocí Gremlin dotazu. Problém zobrazování relevantních dat lze řešit několika způsoby.

```
mgmt = graph.openManagement();

givenName = mgmt.makePropertyKey('givenName')
    .dataType(String.class)
    .make();

familyName = mgmt.makePropertyKey('familyName')
    .dataType(String.class)
    .make();

mgmt.buildIndex('search', Vertex.class)
    .addKey(givenName, Mapping.TEXTSTRING.asParameter())
    .addKey(familyName, Mapping.TEXTSTRING.asParameter())
    .buildMixedIndex('search');

mgmt.commit();
```

Zdrojový kód 15: JanusGraph index pro fulltextové vyhledávání

Prvním způsobem je přidání dodatečné podmínky do každého dotazu. Tento přístup není příliš vhodný, jelikož duplikujeme kód a logika přístupu k datům není centralizovaná.

Druhým způsobem je vytvoření funkce/metody, která se bude o tuto logiku starat. V tomto případě by se čitelnost Gremlin dotazů zhoršila, zejména v případě, kdy by byl vyžadován dotaz komplexní.

Třetí možností, která je v aplikaci implementována, je použití Kotlin extensions. Kotlin extensions umožňují rozšířit funkcionalitu třídy bez nutnosti použití dědičnosti nebo návrhových vzorů jako je například decorator [64]. Kotlin extensions nám umožňují definovat vlastní metodu na třídě starající se o Gremlin dotazy a tím docílit požadovaného chování. Použití tohoto přístupu je zobrazeno na výpisu kódu 16. S tímto přístupem zůstává dotaz čitelný a logika toho, co je relevantní uzel/hrana pro daného uživatele, zůstává na jednom místě.

5.8 Integrace s nástrojem ClueMaker

Integrace s nástrojem ClueMaker vychází ze vzorového archivu, který je možné do tohoto nástroje nahrát. V tomto vzorovém archivu byla nahrazena konfigurace entit těmi, které se vyskytují v aplikaci. Při generování exportu jsou do archivu přibalena požadovaná data a celý archiv je odeslán uživateli ke stažení. Příklad vygenerovaného archivu otevřeného v nástroji ClueMaker je zobrazen na obrázku 5.2.

```
// kotlin extension
fun <T : Element, U : Element> GraphTraversal<T, U>
                .onlyValidElement(): GraphTraversal<T, U> {
    if (UserAccessor.hasRoleAny(*DATA_MASTER_ROLE)) {
        return this
    } else {
        return this.has(
                GraphEntity::state.name,
                GraphEntityState.ACTIVE.name
        )
    }
}

// gremlin query
datasource.findValidVertexById(id)
    .repeat(
        bothE().hasLabel(EdgeType.RELATIONSHIP.name)
                .onlyValidElement()
                .where(inV().onlyValidElement())
                .where(outV().onlyValidElement())
                .subgraph("subgraph")
                .bothV().simplePath()
    ).times(depth)
    .cap<TinkerGraph>("subgraph")
    .tryNext()
```

Zdrojový kód 16: Poskytování relevantních dat pomocí Kotlin extensions

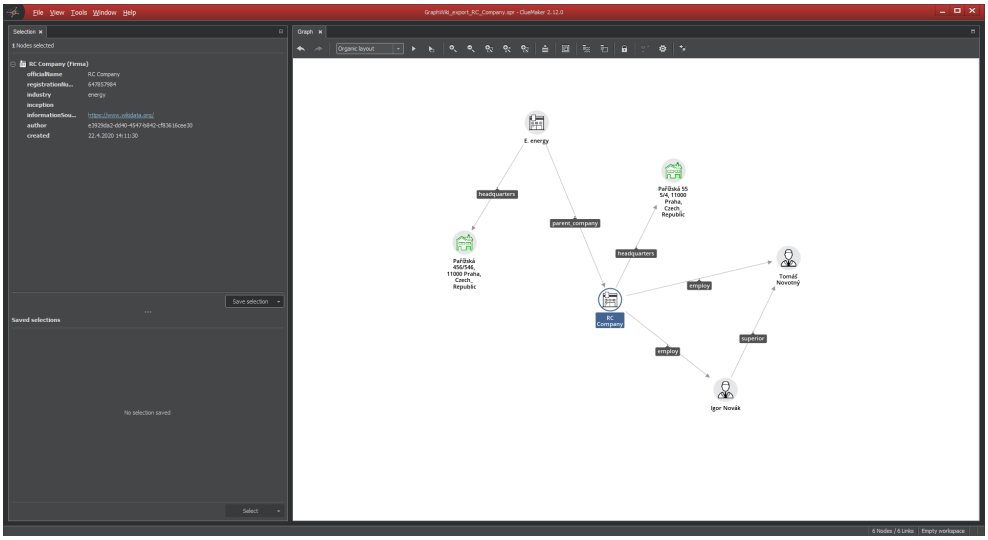
5.9 Dokumentace

Jako součást práce byly vytvořeny dva typy dokumentace – RESTového rozhraní a zdrojového kódu.

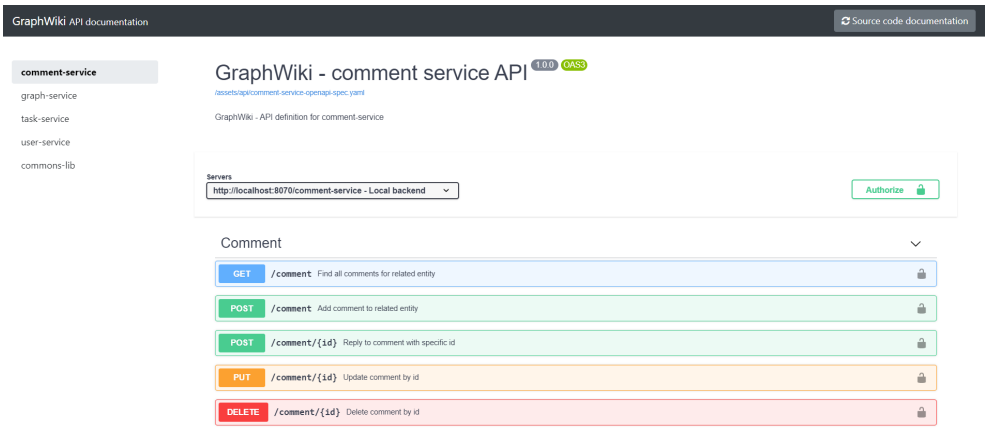
5.9.1 Dokumentace RESTového rozhraní

Jak již bylo zmíněno v kapitole 5.2, OpenAPI definice, která byla při implementaci použita pro generování RESTového rozhraní, je sama o sobě dokumentací. Avšak z uživatelského hlediska soubor ve formátu YAML není zcela dostačující. Z tohoto důvodu byla implementována samostatná aplikace, která obsahuje dokumentaci v uživatelsky přívětivém formátu. K samotné vizualizaci dokumentace byl použit nástroj Swagger UI [65].

Dokumentace obsahuje nejen popis jednotlivých *endpointů* (včetně jejich parametrů, návratových hodnot a dalších informací), ale disponuje i možností jednotlivé *endpointy* provolávat. Stačí pouze zvolit adresu požadovaného ser-



Obrázek 5.2: Integrace s nástrojem ClueMaker

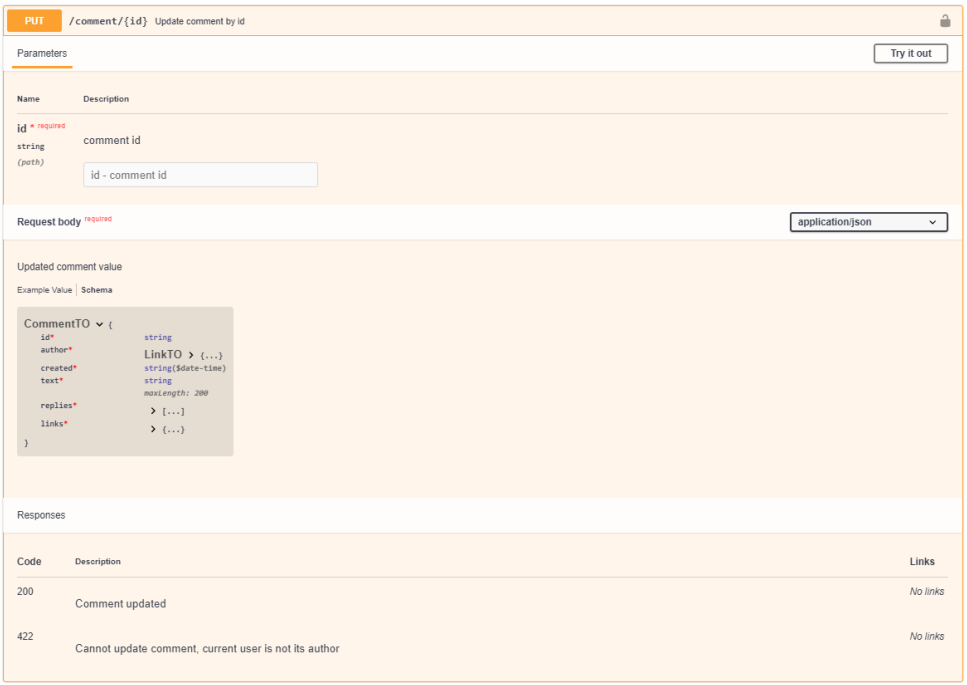


Obrázek 5.3: Dokumentace RESTového rozhraní

veru, následně se vůči němu autorizovat a pak už jen zvolit *endpoint*, vyplnit potřebné parametry a odeslat požadavek. Odpověď se zobrazí v rámci sekce daného *endpointu*.

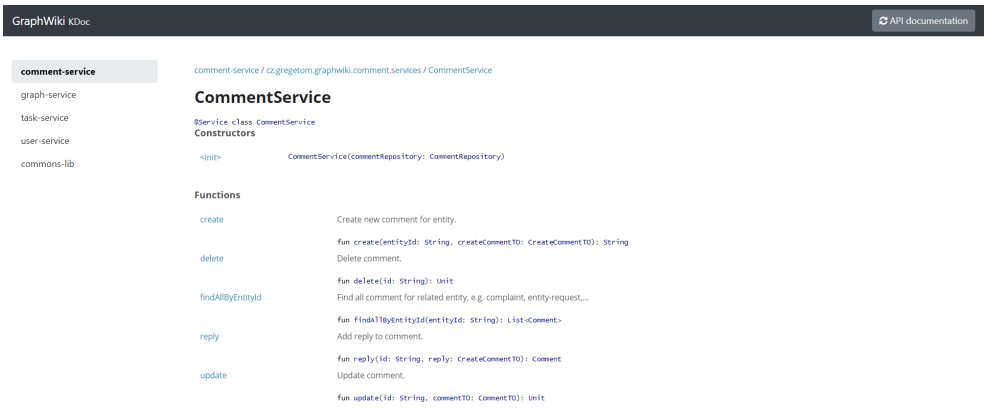
5.9.2 Dokumentace zdrojového kódu

K dokumentaci zdrojového kódu byl použit jazyk Kdoc, který je ekvivalentem z prostředí Javy známému jazyku JavaDoc. Samotná dokumentace je pak vygenerována pomocí nástroje Dokka. [66]



Obrázek 5.4: Dokumentace RESTového *endpointu*

Vygenerovaná dokumentace zdrojového kódu byla pro větší klientský komfort přidána do, v předchozí kapitole zmiňované, aplikace s dokumentací.



Obrázek 5.5: Dokumentace zdrojového kódu

Kapitola 6

Testování

Ke každému softwarovému dílu neodmyslitelně patří fáze testování. Jejím primárním účelem je ověřit správné fungování softwarového produktu. Testy lze dělit do několika kategorií, ať už podle komponent, které mají za úkol testovat (jednotkové, integrační, systémové), nebo podle specifických funkcionalit, například testy výkonnosti nebo zabezpečení.

Základním typem testů, které se nabízejí pro implementovanou aplikaci použít, jsou testy jednotkové. Ty mají za úkol otestovat ty nejmenší komponenty systému, tj. v případě programovacího jazyku Kotlin jednotlivé třídy. Jednotkové testy jsou vhodné, pokud systém obsahuje nějaký komplexnější algoritmus, který lze testovat už právě na úrovni jednotek. Co se týká implementované aplikace, zde žádné komplexní algoritmy na úrovni jednotek nejsou, veškeré procesy, které v aplikaci probíhají, vyžadují kooperaci několika částí aplikace. Z tohoto důvodu nebyly jednotkové testy realizovány, ale byly nahrazeny testy integračními.

Integrační testy slouží primárně k testování několika jednotek ve vzájemné součinnosti. Z výše uvedených důvodů proběhlo otestování implementované aplikace právě pomocí integračních testů, které byly prováděny u každé služby zvlášť.

6.1 Konfigurace integračních testů

Před každým integračním testem je potřeba spustit kontext aplikace a inicializovat všechny potřebné komponenty. Pro tento účel byla vytvořena třída `AbstractIntegrationTest` (viz výpis zdrojového kódu 17), ze které následně vychází každý integrační test. Tato třída také způsobí načtení proměnných ze souboru `application-test.yaml`, ve kterém jsou dostupné konfigurační údaje, například ty potřebné k připojení k databázi. Tyto konfigurační soubory je vhodné použít právě kvůli jednoduché změně konfigurace aplikace a to bez zbytečných zásahů do samotného zdrojového kódu.

```
@ActiveProfiles("test")
@RunWith(SpringRunner::class)
@SpringBootTest(
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    classes = [Application::class, TestConfig::class]
)
@AutoConfigureMockMvc
@Transactional
abstract class AbstractIntegrationTest {
    ...
}
```

Zdrojový kód 17: Třída `AbstractIntegrationTest`

6.1.1 MockMvc

Každý integrační test by v tomto případě měl začínat RESTovým voláním, které aplikace zpracuje a vrátí výsledek, tedy přesně tak, jak funguje v reálném provozu, kdy zdrojem RESTového volání je klientská část. Pro tyto případy poskytuje framework Spring třídu `MockMvc`, která RESTové volání dokáže simulovat. Na výpisu kódu 18 je zobrazeno standardní použití této třídy k provolání RESTové služby.

```
mockMvc.perform(get(uri))
        .andDo(print())
        .andExpect(status().isOk)
        .andReturn()
        .response
```

Zdrojový kód 18: Použití třídy `MockMvc`

6.1.2 MockRestServiceServer

Při testování služby, která ke své správné funkčnosti potřebuje služby jiné, je potřeba zajistit náhradní implementaci těchto služeb, tzv. mock. Pro tento účel byla použita třída `MockRestServiceServer`, která umožňuje nakonfigurovat komponentu, která bude reagovat na určité RESTové požadavky předem definovaným chováním.

6.1.3 EasyRandom

Motivací k použití této knihovny byl fakt, že pokud provádíme test se vzorovými daty, testujeme stále dokola jen chování aplikace za použití těchto

```
val mockServer = MockRestServiceServer.createServer(restTemplate)
mockServer.expect(ExpectedCount.once(),
        requestTo(crossServiceLinkFactory.taskCreate()))
        .andExpect(method(HttpMethod.POST))
        .andRespond(withStatus(HttpStatus.CREATED)
                .location(URI("http://test"))
        )
```

Zdrojový kód 19: Použití třídy `MockRestServiceServer`

vzorových dat. V reálném provozu však mají data různou podobu. K tomuto stavu se chceme během testů co nejvíce přiblížit.

EasyRandom je knihovna pro generování náhodných objektů. Umožňuje generovat nejen náhodné řetězce nebo čísla různých formátů, ale i komplexní objekty. Knihovna dovoluje rozsáhlou počáteční konfiguraci. Mezi zásadní konfigurační parametry lze zařadit:

- **seed** – Při použití stejné hodnoty tohoto parametru budou generovány každý běh testu stejné hodnoty. V testech je naopak požadovaný co největší rozsah hodnot, parametr tedy musí mít při každém běhu jinou hodnotu.
- **collectionSizeRange** – Tento parametr specifikuje, kolik položek bude generováno do proměnných typu kolekce.
- **stringLengthRange** – Tento parametr určuje, v jakém rozsahu se bude pohybovat délka generovaných řetězců.

Ke generování náhodných objektů byla vytvořena v projektu commons-lib třída `RandomGenerator`. Definice této třídy je zobrazena na výpisu zdrojového kódu 20.

6.1.4 Vzorový test

Na výpisu kódu 21 je zobrazen vzorový test, který kombinuje výše zmíněné přístupy. V tomto případě samotný test komunikuje pouze pomocí simulovaných RESTových volání. Pro zjednodušení vytváření těchto volání byly implementovány pomocné třídy vystavující API, jehož použití je stručnější než použití API samotné třídy `MockMvc` .

6.2 Statistiky testů

První statistikou, kterou je vhodné uvést, je počet provedených testů, počty k jednotlivým službám jsou zobrazeny v tabulce 6.1.

```
object RandomGenerator {

    val instance = EasyRandom(
            EasyRandomParameters()
                    .seed(Random.nextLong())
                    .stringLengthRange(1, 25)
                    .collectionSizeRange(0, 10)
    )

    fun randomString(length: Int): String {
        require(length > 0) { "Length must be greater than 0!" }
        return RandomStringUtils.randomAlphanumeric(length)!!
    }
}
```

Zdrojový kód 20: Třída `RandomGenerator`

Tabulka 6.1: Počet provedených testů

Testovaná služba	Počet provedených testů	Počet úspěšných testů
comment-service	3	3
graph-service	60	60
task-service	10	10
user-service	8	8

Počty provedených testů nemusí mít vždy úplnou vypovídající hodnotu. Další zajímavou statistikou je rozsah kódu, který testy pokryjí. Přesné procento pokrytí kódu, kterého by měly testy dosáhnout, neexistuje, vždy závisí na konkrétním projektu. V případě implementované aplikace lze za rozumnou mez pokrytí považovat hodnotu mezi 80-100 %. Dosažená procenta jsou zobrazená v tabulce 6.2. Jako v předchozím případě jsou hodnoty uvedeny pro každou službu zvlášť.

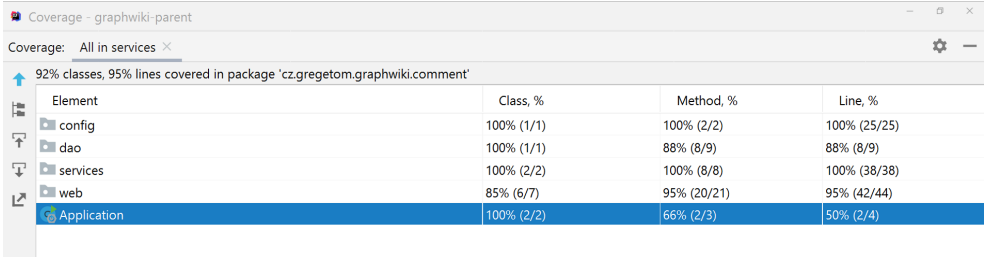
Tabulka 6.2: Procenta pokrytí kódu testy

Testovaná služba	Pokryto tříd	Pokryto řádků kódu
comment-service	92 %	95 %
graph-service	98 %	98 %
task-service	93 %	97 %
user-service	94 %	92 %

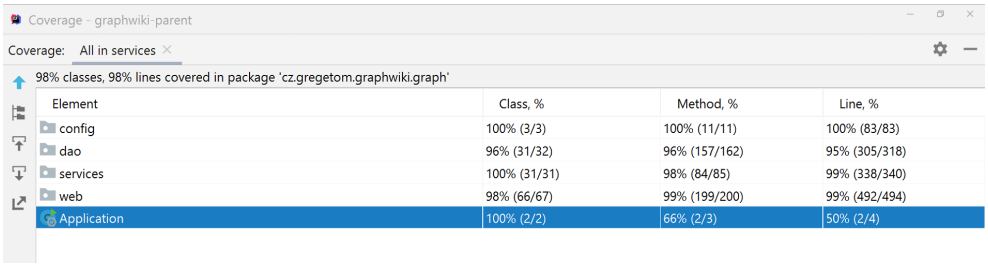
Následují přílohy, které obsahují podrobnější reporty o pokrytí kódu testy z vývojového prostředí.

```
class CreateTest : AbstractIntegrationTest() {

    ...

    @Test
    fun createCompanyTest() {
        // create
        val createCompanyTO = companyDataSupport
                .randomCreateCompanyTO()
        // configure MockRestServiceServer
        this.expectTaskCreating()
        val location = httpPost.doPost(
                LinkFactory.Company.create().toUri(),
                createCompanyTO
        )

        // get
        val entityRequestTO = httpGet.doGet(
                EntityRequestTO::class,
                location
        )
        val companyTO = httpGet.doGet(
                CompanyTO::class,
                entityRequestTO.links.entity.href
        )

        // check
        assertThat(createCompanyTO)
            .isEqualToComparingOnlyGivenFields(
                companyTO, "officialName", "industry", ...)
        assertThat(companyTO.id).isNotNull()
        ...
    }
```

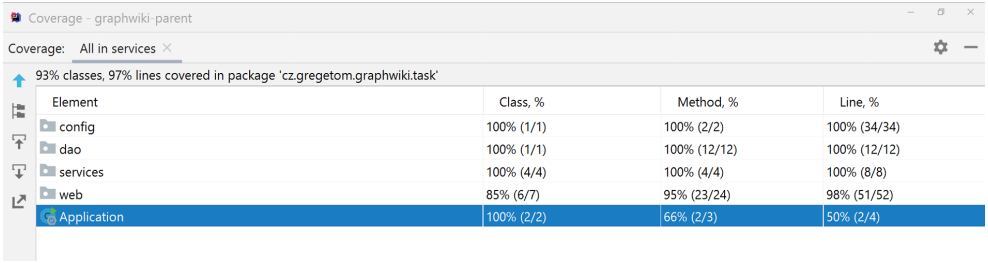
Zdrojový kód 21: Vzorový test – založení firmy



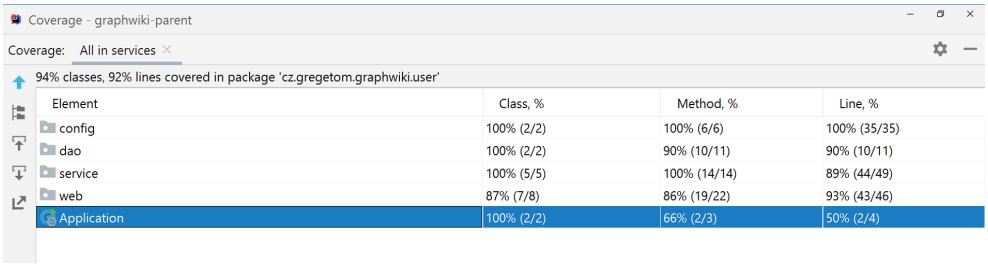
Obrázek 6.1: Procenta pokrytí kódu testy – comment-service



Obrázek 6.2: Procenta pokrytí kódu testy – graph-service



Obrázek 6.3: Procenta pokrytí kódu testy – task-service



Obrázek 6.4: Procenta pokrytí kódu testy – user-service

6.3 Testování uživatelského rozhraní

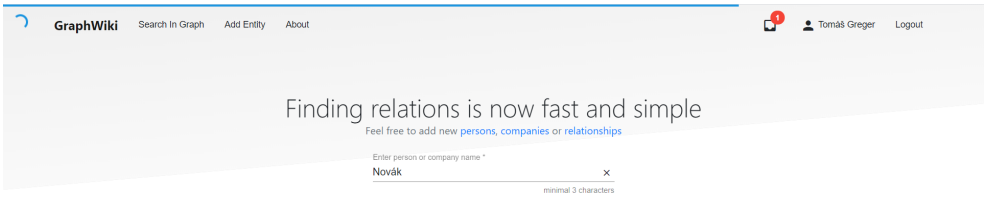
K otestování uživatelského rozhraní byla použita heuristika dle Jakoba Nielsena. Tato heuristika zahrnuje deset hlavních bodů, které by mělo uživatelské rozhraní splňovat, aby bylo pro uživatele přívětivé a snadno použitelné. [67]

6.3.1 Viditelnost stavu systému

Systém by měl uživatele informovat o tom, v jaké části systému se nachází a co se právě děje.

Většina obrazovek obsahuje zřetelný nadpis, podle kterého lze snadno zjistit, v jaké části aplikace se uživatel nachází. Při zakládání většiny entit je uživatel informován o výsledků buď odkázáním na detail založené entity, nebo informační hláškou.

Napříč celou aplikací je zaveden princip, kdy pokud požadavek na serverovou část trvá déle než je definovaná doba, objeví se v horní části obrazovky indikátor procesu načítání. Indikátor nezablokuje celou obrazovku z toho důvodu, že ta se typicky skládá z několika komponent, které posílají požadavky nezávisle na sobě, a tak zpoždění požadavku jedné komponenty neznamená zamrznutí celé aplikace.



Obrázek 6.5: Indikátor procesu načítání

6.3.2 Shoda mezi systémem a realitou

Systém by měl mluvit jazykem uživatele a používat fráze a koncepty, které jsou uživateli dobře známy.

Aplikace využívá pro komunikaci s uživatelem jednoduchou angličtinu. Dále jsou použity doplňující ikony (označení pro osoby/firmy, exporty a jiné), které jsou standardními a v souladu s jinými aplikacemi. Jako rozšíření se nabízí jazyková lokalizace.

6.3.3 Uživatelská kontrola a svoboda

Uživatelé mohou provádět funkce v systému omylem. Potřebují tedy jednoduchou cestu, jak se vrátit zpět.

Webová aplikace z povahy věci umožňuje krok zpět pomocí nativní podpory prohlížeče. Mimo to je uživateli dostupné navigační menu. Všechny akce mazání jsou podmíněné potvrzením dialogu, který lze snadno ukončit.

6.3.4 Konzistence a standardizace

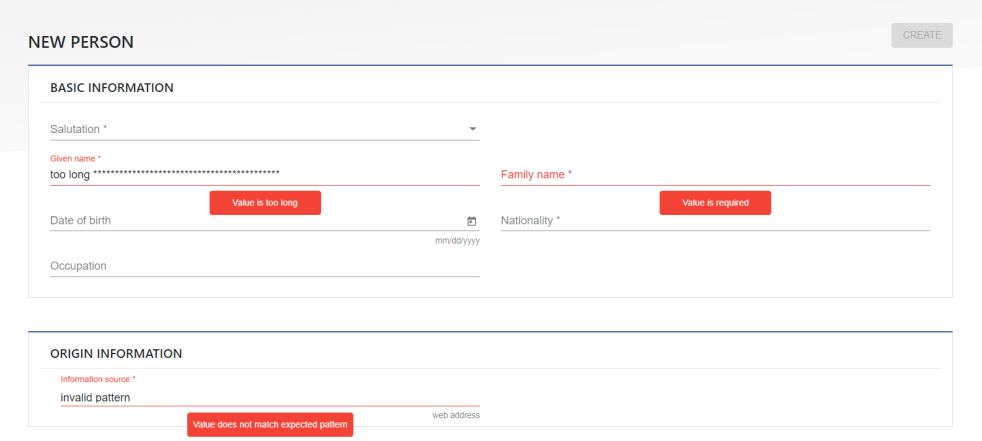
Uživatelé by neměli přemýšlet nad tím, zda různá slova, situace nebo činy znamenají totéž.

Napříč aplikací je použita stejná sémantika názvů/popisků a ikon, které jsou konzistentní s ostatními aplikacemi. Z důvodu komponentního designu je rozložení obrazovek významově podobného typu vždy stejné.

6.3.5 Prevence chyb

Lepší než dobré chybové hlášky je návrh, který primárně výskytu chyb zabraňuje.

Při vyplňování údajů do formuláře jsou uživateli ihned zobrazeny nevalidní hodnoty. Kontrola těchto polí probíhá buď jako kontrola na formát vložené hodnoty (povinné pole, email), nebo jako logická kontrola (unikátnost uživatelského jména), kdy je k validaci použit dotaz na serverovou část. Formulář nelze odeslat do chvíle, kdy jsou všechny hodnoty validní. Dostupnost uživatelských akcí je řízena přímo serverovou částí, tudíž je zamezeno zobrazení ovládacího prvku pro akci, která pro daného uživatele není dostupná.



Obrázek 6.6: Formulářové validace

6.3.6 Rozpoznání místo vzpomínání

Uživatel by neměl být nucen si pamatovat, jak v systému provádět různé operace. Instrukce by měly být v systému vždy viditelně umístěny.

Aplikace neposkytuje uživateli žádné složité procesy. Uživatel si typicky vystačí s řízením se dle navigačních tlačítek a nadpisů obrazovek/sekcí. Dostupné ovládací prvky pro různé akce jsou mu zobrazeny ihned (případně je

při nesplnění určité podmínky zamezeno jejich použití), uživatel tedy přesně ví, jaké akce může v dané situaci provést.

6.3.7 Flexibilní a efektivní použití

Pokročilým uživatelům by měla být dostupná možnost urychlení provádění častých akcí.

Vzhledem k poskytovaným funkcím žádné takové možnosti pro pokročilé uživatele nejsou.

6.3.8 Estetický a minimalistický design

Obrazovky by neměly obsahovat irelevantní nebo zřídka potřebné údaje, které by zastiňovaly údaje relevantní.

V aplikaci je typicky použit princip zobrazování dat v rozdílných detailech. Příkladem mohou být detaily osob/firem. Při vyhledávání jsou dostupné pouze základní informace (jméno, příjmení a datum narození u osoby, resp. název a IČO u firmy), při zobrazení v grafovém zobrazení pouze informace dostupné přímo na této entitě a na obrazovce detailu pak další informace, jako jsou navázané vazby, stížnosti nebo historie.

6.3.9 Pomoc uživatelům pochopit a vzpamatovat se z chyb

O výskytu chyby by měl být uživatel informován pomocí prostého textu s označením problému, případně s návrhem konstruktivního řešení.

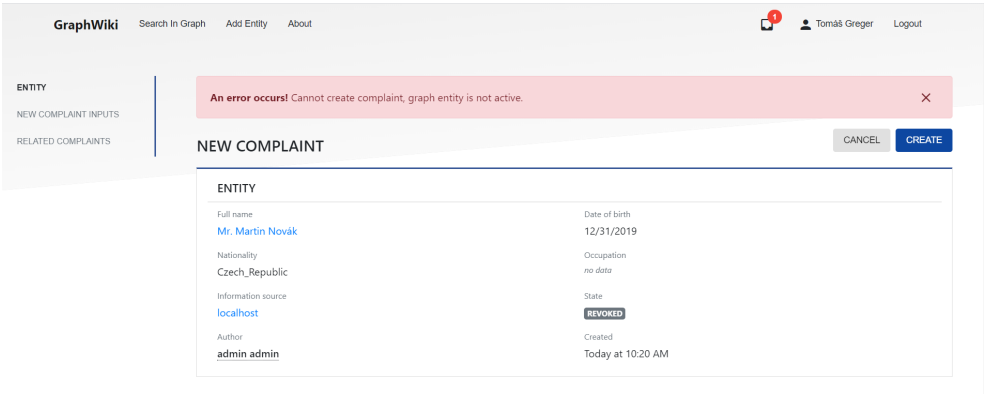
Pokud se uživatel snaží přistoupit na obrazovku klientské části, která neexistuje, je přesměrován na speciální obrazovku, která ho o tomto faktu informuje.

Pokud chybu vrátí serverová část, dle jejího typu je uživateli zobrazena relevantní obrazovka (nedostatečná práva, stránka nenalezena, obecná chyba). Pokud se však jedná o chybu z důvodu neúspěšné validace požadavku, je uživatel informován pomocí zprávy přímo na obrazovce, na které se právě nachází.

6.3.10 Nápověda a dokumentace

Přestože je lepší, pokud lze systém používat bez dokumentace, její existence může být v některých případech nutná.

Nápověda či dokumentace pro uživatele není v tuto chvíli dostupná.



Obrázek 6.7: Zobrazení chyby při neúspěšné validaci požadavku

Kapitola 7

Možné rozšíření

Vizí projektu GraphWiki je vytvořit plnohodnotnou platformu postavenou na myšlence grafové Wikipedie. Tato práce měla za úkol položit základ tohoto projektu. Nyní, kdy implementovaná aplikace splňuje definované požadavky, je potřeba vytvořit dostatečnou infrastrukturu pro provoz této aplikace v produkčním módu. Během doby kolem spuštění se dají očekávat potencionální výskyty chyb a nedostatků, které vyplynou až z používání aplikace reálnými uživateli.

Při návrhu a implementaci byl kladen důraz na možnou rozšiřitelnost celého řešení. Po fázi úspěšného spuštění aplikace v nynější verzi je zamýšleno její rozšiřování o nové funkcionality a primárně o další typy entit. Mezi možné rozšíření o nové funkcionality lze zařadit: přihlášení uživatele přes externího zprostředkovatele (Facebook, Google), zvětšení počtu dostupných grafových rozložení pro zobrazování dat nebo vytvoření nové služby pro emailové notifikace. V oblasti testů se nabízí zavést end-2-end testování, které by testovalo celý produkt jako celek. Nutností bude také rozšíření portfolia podporovaných prohlížečů.

V budoucnosti se dá počítat i s integrací na jiné nástroje. Počítá se s užším napojením na portál Wikidata, může být rozšířena spolupráce s nástrojem ClueMaker nebo implementována integrace nová, například za použití standardu GraphML.

Závěr

Cílem práce bylo analyzovat, navrhnout, implementovat a otestovat webovou aplikaci, která bude umožňovat v nynější verzi uchovávat vazby mezi osobami a firmami, a tím položit základ projektu GraphWiki. Zadání bylo splněno jak z hlediska funkčních a nefunkčních požadavků, tak z hlediska použitých technologií. Kvůli plánovanému rozvoji byl během implementační fáze kladen důraz primárně na rozšiřitelnost a škálovatelnost celého řešení.

Přínosem této práce, a projektu GraphWiki jako takového, je zpřístupnění velkého množství dat z různých oborů a jejich následné zobrazení ve vizuální podobě široké veřejnosti. Za jednu z výhod lze považovat kompatibilitu s již existujícím nástrojem ClueMaker.

Výsledná práce, včetně podpůrného materiálu použitého v tomto dokumentu a zdrojových kódů, je nahrána na přiloženém CD.

Literatura

[1] Wikimedia Foundation, Inc: *WikiMedia Foundation [online]*. [cit. 2020-04-02]. Dostupné z: https://wikimediafoundation.org/

[2] AliaWeb.cz, a.s.: *podnikanicz [online]*. [cit. 2020-04-02]. Dostupné z: https://www.podnikani.cz/

[3] Rousek, L.: Prověřte si svého obchodního partnera. S FIRMO.CZ je to snadné! *Webitech [online]*, srpen 2015, [cit. 2020-04-02]. Dostupné z: http://webitech.cz/proverte-si-sveho-obchodniho-partnera-s-firmo-cz-je-to-snadne/

[4] Fordesk s.r.o.: *FIRMO [online]*. [cit. 2020-04-02]. Dostupné z: https://www.firmo.cz/

[5] Imper CZ s.r.o.: *Merk [online]*. [cit. 2020-04-18]. Dostupné z: https://www.merk.cz/

[6] Bisnode Česká republika, a.s.: *Bisnode BIZguard [online]*. [cit. 2020-04-18]. Dostupné z: https://www.bisnode.cz/produkty/bizguard/

[7] QRA: Functional vs Non-Functional Requirements: The Definitive Guide. *QRA [online]*, září 2019, [cit. 2020-05-03]. Dostupné z: https://qracorp.com/functional-vs-non-functional-requirements/

[8] GraphML Team: *The GraphML File Format [online]*. [cit. 2020-04-05]. Dostupné z: http://graphml.graphdrawing.org/

[9] Profinit EU, s.r.o.: *ClueMaker [online]*. [cit. 2020-04-05]. Dostupné z: https://cluemaker.com/

[10] JanusGraph Authors: *JanusGraph [online]*. [cit. 2020-04-05]. Dostupné z: https://janusgraph.org/

[11] Elastic NV: *What is Elasticsearch? [online]*. [cit. 2020-04-05]. Dostupné z: https://www.elastic.co/what-is/elasticsearch

[12] Apache Software Foundation: *Cassandra [online]*. [cit. 2020-04-05]. Dostupné z: https://cassandra.apache.org/

[13] Wu, J.: Choosing The Right Database. *Towards Data Science [online]*, červen 2019, [cit. 2020-04-10]. Dostupné z: https://towardsdatascience.com/choosing-the-right-database-c45cd3a28f77

[14] Nazrul, S. S.: CAP Theorem and Distributed Database Management Systems. *Towards Data Science [online]*, duben 2018, [cit. 2020-04-10]. Dostupné z: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e

[15] Elicherla, R.: CAP Theorem simplified. *Medium [online]*, srpen 2017, [cit. 2020-04-30]. Dostupné z: https://medium.com/@ravindraprasad/cap-theorem-simplified-28499a67eab4

[16] Oracle Corporation: *Co je relační databáze [online]*. [cit. 2020-04-11]. Dostupné z: https://www.oracle.com/cz/database/what-is-a-relational-database/

[17] Microsoft Corporation: *Nerelační data a NoSQL [online]*. [cit. 2020-04-11]. Dostupné z: https://docs.microsoft.com/cs-cz/azure/architecture/data-guide/big-data/non-relational-data

[18] PostgreSQL Global Development Group: *PostgreSQL [online]*. [cit. 2020-04-12]. Dostupné z: https://www.postgresql.org/

[19] Chiessi, L.: Why should I use PostgreSQL as Database in my Startup/Company. *Medium [online]*, září 2018, [cit. 2020-04-12]. Dostupné z: https://medium.com/we-build-state-of-the-art-software-creating/why-should-i-use-postgresql-as-database-in-my-startup-company-96de2fd375a9

[20] Raycad: Monolithic vs Microservice Architecture. *Medium [online]*, říjen 2018, [cit. 2020-04-13]. Dostupné z: https://medium.com/@raycad.seedotech/monolithic-vs-microservice-architecture-e74bd951fc14

[21] Malav, B.: Microservices vs Monolithic architecture. *Medium [online]*, prosinec 2017, [cit. 2020-05-03]. Dostupné z: https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4

[22] JetBrains s.r.o.: *Kotlin programming language [online]*. [cit. 2020-04-04]. Dostupné z: https://kotlinlang.org/

[23] VMware, Inc: *Spring Boot [online]*. [cit. 2020-04-04]. Dostupné z: https://spring.io/projects/spring-boot

[24] Wasson, M.: Single-Page Applications. *Microsoft Docs [online]*, listopad 2013, [cit. 2020-04-07]. Dostupné z: https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/november/asp-net-single-page-applications-build-modern-responsive-web-apps-with-asp-net

[25] Neoteric: Single-page application vs. multiple-page application. *Medium [online]*, prosinec 2016, [cit. 2020-04-07]. Dostupné z: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

[26] Duomly: The best front-end framework to learn in 2019. *DEV Community [online]*, říjen 2019, [cit. 2020-04-07]. Dostupné z: https://dev.to/duomly/the-best-front-end-framework-to-learn-in-2019-dn7

[27] Google: *Angular [online]*. [cit. 2020-04-07]. Dostupné z: https://angular.io/

[28] Codecademy: *What is REST? [online]*. [cit. 2020-04-09]. Dostupné z: https://www.codecademy.com/articles/what-is-rest

[29] RESTfulAPI.net: *What is REST [online]*. [cit. 2020-04-24]. Dostupné z: https://restfulapi.net/

[30] Phppot: *PHP RESTful Web Service API [online]*. [cit. 2020-04-10]. Dostupné z: https://phppot.com/php/php-restful-web-service/

[31] Chris Richardson: *Pattern: Messaging [online]*. [cit. 2020-04-13]. Dostupné z: https://microservices.io/patterns/communication-style/messaging.html

[32] Chris Richardson: *Pattern: Database per service [online]*. [cit. 2020-04-13]. Dostupné z: https://microservices.io/patterns/data/database-per-service.html

[33] Chris Richardson: *Pattern: Saga [online]*. [cit. 2020-04-13]. Dostupné z: https://microservices.io/patterns/data/saga.html

[34] Chris Richardson: *Pattern: Command Query Responsibility Segregation (CQRS) [online]*. [cit. 2020-04-13]. Dostupné z: https://microservices.io/patterns/data/cqrs.html

[35] Chris Richardson: *Pattern: Transactional outbox [online]*. [cit. 2020-04-13]. Dostupné z: https://microservices.io/patterns/data/transactional-outbox.html

[36] Mohammed, A.: Asynchronous communication in Microservices. *Medium [online]*, květen 2019, [cit. 2020-04-14]. Dostupné z: https://medium.com/@aamermail/asynchronous-communication-in-microservices-14d301b9016

[37] JanusGraph Authors: *Architectural Overview [online]*. [cit. 2020-04-16]. Dostupné z: https://docs.janusgraph.org/getting-started/architecture/

[38] Apache Software Foundation: *Apache TinkerPop [online]*. [cit. 2020-04-16]. Dostupné z: https://tinkerpop.apache.org/

[39] Apache Software Foundation: *The Gremlin Graph Traversal Machine and Language [online]*. [cit. 2020-04-17]. Dostupné z: https://tinkerpop.apache.org/gremlin.html

[40] JanusGraph Authors: *Deployment Scenarios [online]*. [cit. 2020-04-17]. Dostupné z: https://docs.janusgraph.org/basics/deployment/

[41] IBM: *Import CSV file to JanusGraph [online]*. [cit. 2020-04-30]. Dostupné z: https://github.com/IBM/janusgraph-utils/blob/master/doc/users_guide.md#import-csv-file-to-janusgraph

[42] Apache Software Foundation: *The Gremlin Console [online]*. [cit. 2020-04-30]. Dostupné z: http://tinkerpop.apache.org/docs/3.4.6/tutorials/the-gremlin-console/

[43] Abhsk: Set up ELK (Elastic, Logstash, Kibana) stack on windows. *Abhishek [online]*, únor 2019, [cit. 2020-04-20]. Dostupné z: https://abhishek.co.uk/?p=94

[44] VMware, Inc: *Spring Framework [online]*. [cit. 2020-04-24]. Dostupné z: https://spring.io/projects/spring-framework

[45] Google: *Introduction to Angular concepts [online]*. [cit. 2020-04-19]. Dostupné z: https://angular.io/guide/architecture

[46] Google: *Introduction to modules [online]*. [cit. 2020-04-19]. Dostupné z: https://angular.io/guide/architecture-modules

[47] Google: *Introduction to components and templates [online]*. [cit. 2020-04-19]. Dostupné z: https://angular.io/guide/architecture-components

[48] Google: *Structural directives [online]*. [cit. 2020-04-19]. Dostupné z: https://angular.io/guide/structural-directives

[49] Google: *Introduction to services and dependency injection [online]*. [cit. 2020-04-19]. Dostupné z: https://angular.io/guide/architecture-services

[50] JavaTpoint: *Angular 7 Architecture [online]*. [cit. 2020-04-30]. Dostupné z: https://www.javatpoint.com/angular-7-architecture

[51] Docker Inc.: *What is a Container? [online]*. [cit. 2020-05-18]. Dostupné z: https://www.docker.com/resources/what-container

[52] do Prado, K. S.: Docker Introduction. *Medium [online]*, říjen 2018, [cit. 2020-05-18]. Dostupné z: https://medium.com/@kelvin_sp/docker-introduction-what-you-need-to-know-to-start-creating-containers-8ffaf064930a

[53] Docker Inc.: *Docker documentation? [online]*. [cit. 2020-05-18]. Dostupné z: https://docs.docker.com/

[54] Google: *Preventing cross-site scripting (XSS) [online]*. [cit. 2020-04-21]. Dostupné z: https://angular.io/guide/security#xss

[55] Oliver Gierke, Thomas Darimont, Christoph Strobl, Mark Paluch, Jay Bryant: *Spring Data JPA - Reference Documentation [online]*. [cit. 2020-04-22]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference

[56] Apache Software Foundation: *Gremlin Server [online]*. [cit. 2020-04-22]. Dostupné z: http://tinkerpop.apache.org/docs/current/reference/#gremlin-server

[57] Apache Software Foundation: *Parameterized Scripts [online]*. [cit. 2020-04-22]. Dostupné z: http://tinkerpop.apache.org/docs/current/reference/#parameterized-scripts

[58] SmartBear Software: *What Is OpenAPI? [online]*. [cit. 2020-04-23]. Dostupné z: https://swagger.io/docs/specification/about/

[59] OpenAPI-Generator Contributors: *OpenAPI Generator [online]*. [cit. 2020-04-10]. Dostupné z: https://openapi-generator.tech/

[60] Karanam, R.: REST API — What Is HATEOAS? *DZone [online]*, listopad 2019, [cit. 2020-04-24]. Dostupné z: https://dzone.com/articles/rest-api-what-is-hateoas

[61] Microsoft Corporation: *Spring Data Gremlin [online]*. [cit. 2020-04-26]. Dostupné z: https://github.com/microsoft/spring-data-gremlin

[62] Peter Meyers: *The Object Graph Mapping Library for Kotlin and Gremlin [online]*. [cit. 2020-04-26]. Dostupné z: `https://github.com/pm-dev/kotlin-gremlin-ogm`

[63] Syncleus, Inc.: *Ferma [online]*. [cit. 2020-04-26]. Dostupné z: `http://syncleus.com/Ferma/`

[64] JetBrains s.r.o.: *Extensions [online]*. [cit. 2020-04-25]. Dostupné z: `https://kotlinlang.org/docs/reference/extensions.html`

[65] SmartBear Software: *Swagger UI [online]*. [cit. 2020-05-09]. Dostupné z: `https://swagger.io/tools/swagger-ui/`

[66] JetBrains s.r.o.: *Documenting Kotlin Code [online]*. [cit. 2020-05-09]. Dostupné z: `https://kotlinlang.org/docs/reference/kotlin-doc.html`

[67] Ibanez, L.: User Experience: Jakob Nielsen's 10 general principles for interaction design. *Medium [online]*, červen 2019, [cit. 2020-05-10]. Dostupné z: `https://medium.com/theagilemanager/user-experience-jakob-nielsens-10-general-principles-for-interaction-design-2593b0b53ddc`

[68] PostgreSQL Global Development Group: *Linux downloads (Red Hat family) [online]*. [cit. 2020-05-02]. Dostupné z: `https://www.postgresql.org/download/linux/redhat/`

[69] PostgreSQL Global Development Group: *Authentication Methods [online]*. [cit. 2020-05-02]. Dostupné z: `https://www.postgresql.org/docs/9.1/auth-methods.html`

[70] Biliya, N.: Deploying angular application on Tomcat server. *Medium [online]*, duben 2019, [cit. 2020-05-02]. Dostupné z: `https://medium.com/@nithin.biliya/deploying-angular-application-on-tomcat-server-fixing-deep-linking-issue-577565fe303d`

Příloha **A**

Seznam použitých zkratek

IT Information technology

SEO Search engine optimization

SPA Single page application

XSS Cross-site scripting

API Application interface

JVM Java Virtual Machine

JSON JavaScript Object Notation

YAML YAML Ain't Markup Language

HTML Hypertext Markup Language

DOM Document Object Model

SQL Structured Query Language

JWT Json Web Token

HTTP Hypertext Transfer Protocol

JPA Java Persistence API

URL Uniform Resource Locator

NPM Node package manager

REST Representational State Transfer

Příloha B

Uživatelská příručka

V této kapitole jsou obsaženy potřebné kroky k úspěšnému spuštění aplikace. Ta, jak už bylo zmíněno, se skládá ze čtyř projektů – UI, API, services a commons-lib. Mimo to tato kapitola obsahuje návod na spuštění aplikace s dokumentací (projekt docs).

Jednotlivé nástroje použité při vývoji byly provozovány na virtuálním serveru s operačním systémem Centos 8. Uživatelská příručka tuto skutečnost reflektuje.

Nutno poznamenat, že během implementace nebyla dostupná úložiště pro npm ani Maven artefakty. Jednotlivé závislosti mezi projekty jsou tedy řešeny lokálně.

B.1 Nástroje

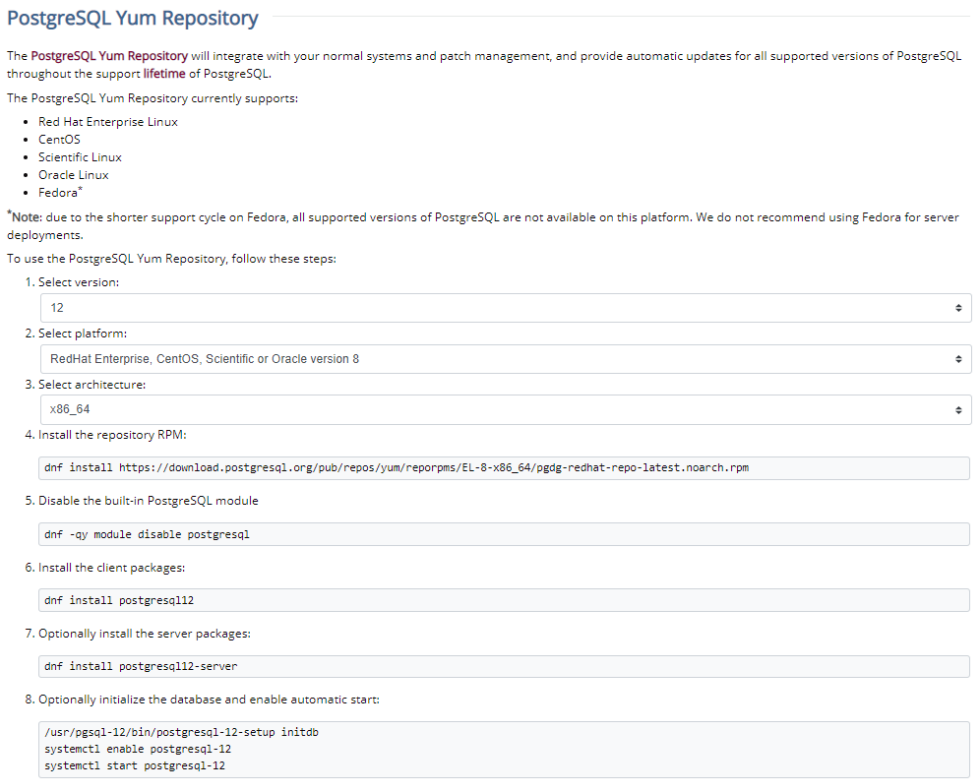
Prvním krokem k úspěšnému spuštění aplikace je instalace nástrojů dle tabulky B.1.

Tabulka B.1: Potřebné nástroje ke spuštění aplikace

Nástroj	Verze	Návod
Java	1.8	–
Node.js	12.14.1	–
npm	6.13.4	–
PostgreSQL	12.2	viz kapitola B.2
JanusGraph	0.4.1	viz kapitola B.3
Docker	19.03.8	–
Jenkins	2.204.5	–

B.2 Nastavení databáze PostgreSQL

Pro instalaci této databáze lze použít návod, který je dostupný přímo na oficiálních stránkách, viz [68]. Stačí vyplnit informace o operačním systému a použít doporučené příkazy k instalaci.



Obrázek B.1: Instalace PostgreSQL databáze [68]

Po úspěšné instalaci je potřeba definovat uživatele, přes kterého bude aplikace s databází komunikovat. Při použití výchozího uživatele *postgres* je nutné pouze změnit heslo tohoto uživatele, viz výpis zdrojového kódu 22. Zvolené přístupové údaje je následně nutné promítnout do konfiguračních souborů jednotlivých služeb v projektu services, více informací o konfiguraci služeb je uvedeno v kapitole B.4.1. Při použití přístupu, kdy databáze běží na separátním serveru, je nutné nastavit dle [69] možnost přihlášení přes dvojici údajů username/password.

```
sudo -u postgres psql postgres
\password postgres
```

Zdrojový kód 22: PostgreSQL – změna hesla výchozího uživatele

B.3 Nastavení databáze JanusGraph

Dokumentace pro instalaci, konfiguraci a práci s JanusGraph databází je součástí zdrojového kódu. Rozcestníkem celé dokumentace je soubor umístěný v adresáři `graph-service/JanusGraph/JanusGraph.md` v projektu services (viz obrázek B.2). Součástí dokumentace je mimo jiné i vytvoření indexu pro fulltextové vyhledávání. Při jakékoliv změně přístupových údajů k databázi je tuto změnu opět nutné reflektovat v konfiguračních souborech jednotlivých služeb.

Nastavení této databáze je určeno pouze pro vývojové účely. Produkční nastavení nebylo po domluvě s vedoucím práce realizováno.

B.4 Serverová část

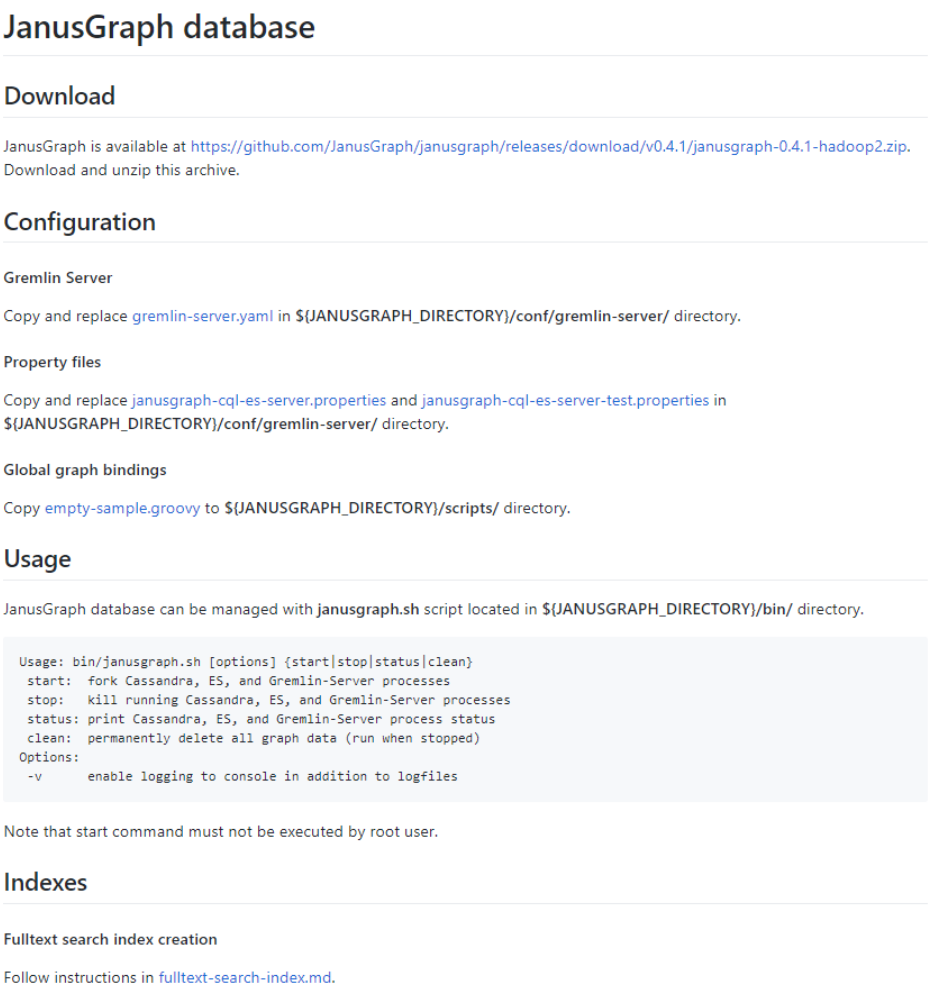
Pro spuštění serverové části aplikace je potřeba splnit několik dílčích kroků, které jsou popsány v následujících kapitolách.

B.4.1 Konfigurace služeb

Každá služba v projektu services vyžaduje ke svému spuštění konfiguraci. Jednotlivé konfigurace jsou umístěny v souborech `application-\$PROFILE.yml` ve složce `src/main/resources`. Pro každé prostředí typicky existuje unikátní dvojice profil a konfigurační soubor, tj. pro lokální prostředí existuje profil *local* a konfigurační soubor `application-local.yml`, pro ostatní prostředí analogicky. Vzorový příklad obsahu těchto souborů je zobrazen na výpisu kódu 23. V aplikaci jsou dostupné dva profily, prvním z nich je profil *local*, který slouží k vývoji, druhým pak *remote*, který je určen pro spuštění služeb přímo na virtuálním serveru.

B.4.2 Závislost na projektech API a commons-lib

Nejdříve je nutné publikovat artefakty z projektů API (z tohoto projektu vznikne pro každou službu nezávislý artefakt) a commons-lib do Maven repozitáře. K tomu stačí provést příkaz `mvn clean install` v kořenových adresářích obou projektů. Následně je potřeba přidat závislost na artefakty v odpovídajících verzích do jednotlivých služeb v projektu services. To lze provést přidáním Maven koordinátů publikovaných artefaktů do souboru `pom.xml`, který se nachází v kořenovém adresáři každé služby. Vzorové použití je zobrazeno na výpisu kódu 24.



JanusGraph database

Download

JanusGraph is available at https://github.com/JanusGraph/janusgraph/releases/download/v0.4.1/janusgraph-0.4.1-hadoop2.zip. Download and unzip this archive.

Configuration

Gremlin Server

Copy and replace gremlin-server.yaml in **${JANUSGRAPH_DIRECTORY}/conf/gremlin-server/** directory.

Property files

Copy and replace janusgraph-cql-es-server.properties and janusgraph-cql-es-server-test.properties in **${JANUSGRAPH_DIRECTORY}/conf/gremlin-server/** directory.

Global graph bindings

Copy empty-sample.groovy to **${JANUSGRAPH_DIRECTORY}/scripts/** directory.

Usage

JanusGraph database can be managed with **janusgraph.sh** script located in **${JANUSGRAPH_DIRECTORY}/bin/** directory.

```
Usage: bin/janusgraph.sh [options] {start|stop|status|clean}
 start:  fork Cassandra, ES, and Gremlin-Server processes
 stop:   kill running Cassandra, ES, and Gremlin-Server processes
 status: print Cassandra, ES, and Gremlin-Server process status
 clean:  permanently delete all graph data (run when stopped)
Options:
 -v      enable logging to console in addition to logfiles
```

Note that start command must not be executed by root user.

Indexes

Fulltext search index creation

Follow instructions in fulltext-search-index.md.

Obrázek B.2: Nastavení JanusGraph databáze

B.4.3 Sestavení a spuštění

Pro sestavení všech služeb je potřeba provést příkaz `mvn clean package` v kořenovém adresáři projektu services. Po úspěšném dokončení tohoto příkazu vznikne pro každou službu v adresáři `services/*-service/target` spustitelný JAR soubor. Každý takový soubor lze spustit s odpovídajícím profilem, příkaz ke spuštění je dostupný na výpisu kódu 25.

```
server:
  port: 8080

spring:
  # PostgreSQL datasource
  datasource:
    username: *****
    password: *****

graphwiki:
  janusgraph:
    # JanusGraph datasource
    datasource:
      host: localhost
      port: 8182
      traversal-source-name: g

  web:
    cors:
      allowed-origins: http://localhost:4200

    ...
```

Zdrojový kód 23: Vzorový konfigurační soubor

B.5 Klientská část

B.5.1 Závislost na projektu API

Obdobně jako u serverové části je nutné přidat závislost na projektu API. Vygenerování potřebných knihoven z tohoto projektu lze provést příkazem `mvn clean package` v kořenovém adresáři tohoto projektu. Poté je nutné vygenerované knihovny připojit k projektu UI, toho se docílí přidáním závislostí v souboru `package.json`, kde je nutné přidat konfiguraci zobrazenou na výpisu zdrojového kódu 26.

B.5.2 Sestavení a spuštění

Po úspěšném připojení knihoven je klientská část připravena ke spuštění. Do příkazové řádky stačí zadat nejdříve příkaz `npm install`, který nainstaluje potřebné závislosti, a poté příkaz `npm start` pro spuštění s konfigurací pro vývojové prostředí, resp. `npm run start:production` pro spuštění s konfigurací produkční. Aplikace bude po úspěšném spuštění dostupná na portu 4200. Vystavení klientské části do produkčního prostředí probíhá typicky jejím nasazením na webový server, k tomuto účelu je dostupný příkaz `npm run build`,

```
<dependency>
    <groupId>cz.gregetom.graphwiki</groupId>
    <artifactId>commons-lib</artifactId>
    <version>${version}</version>
</dependency>

<dependency>
    <groupId>cz.gregetom.graphwiki.api</groupId>
    <artifactId>graph</artifactId>
    <version>${version}</version>
</dependency>
```

Zdrojový kód 24: Přidání Maven závislostí do projektu

```
java -jar -Dspring.profiles.active=${ACTIVE_PROFILE} ${JAR_FILE}
```

Zdrojový kód 25: Spuštění JAR souboru s příslušným profilem

```
{
    "dependencies": {
        "@graphwiki/comment-service-api":
        "file:$API_PROJECT_DIR/comment/target/npm-package/dist",
        "@graphwiki/graph-service-api":
        "file:$API_PROJECT_DIR/graph/target/npm-package/dist",
        "@graphwiki/task-service-api":
        "file:$API_PROJECT_DIR/task/target/npm-package/dist",
        "@graphwiki/user-service-api":
        "file:$API_PROJECT_DIR/user/target/npm-package/dist",
         "..."
    },
    "..."
}
```

Zdrojový kód 26: Projekt UI – definice závislostí na projektu API

který sestaví aplikaci s produkční konfigurací, samotné nasazení lze poté realizovat například dle [70].

B.6 Dokumentace

Obdobně jako klientská část je aplikace pro dokumentaci implementována pomocí frameworku Angular. Pro sestavení a spuštění aplikace jsou dostupné stejné příkazy, jako u klientské části. Příkaz `npm install` slouží k instalaci potřebných závislostí, `npm start` pro spuštění s konfigurací pro vývojové

prostředí, `npm run start:production` pro spuštění s konfigurací produkční. V tomto případě bude aplikace po úspěšném spuštění dostupná na portu 4201. Pro sestavení aplikace k nasazení na webový server je určen příkaz `npm run build`.

B.7 Nasazení pomocí nástrojů Jenkins a Docker

V každém projektu je dostupný soubor `Jenkinsfile`, který se používá k automatizaci procesů pomocí nástroje Jenkins. Pro projekty API a commons-lib tento soubor slouží k definici *pipeline*, která umožňuje instalaci artefaktů do repozitářů. V případě projektů UI, services a docs pak k sestavení, otestování a nasazení spustitelné formy projektu jako Docker kontejner. U těchto projektů je dostupný i soubor `Dockerfile`, který obsahuje konfiguraci obrazu daného kontejneru.

B.8 Poznámky

- Vytvoření správce aplikace probíhá manuálním vložením role k danému uživateli do databáze.

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `application` ..... adresář se spustitelnou formou implementace
  - `services` ..... serverová část
    - `comment-service.jar`
    - `graph-service.jar`
    - `task-service.jar`
    - `user-service.jar`
  - `ui` ..... klientská část
  - `docs` ..... dokumentace
- `materials` ..... ostatní materiály
- `src` ..... zdrojové kódy implementace
  - `api` ..... projekt API
  - `commons-lib` ..... projekt commons-lib
  - `docs` ..... projekt s dokumentací
  - `services` ..... serverová část
  - `ui` ..... klientská část
  - `thesis.zip` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF