Masarykova univerzita
Fakulta informatiky

Implementace fulltextového vyhledávání v systému správy požadavků

Bakalářská práce

Jiří Holuša

Brno, Jaro 2014

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Jiří Holuša

Vedoucí práce: Mgr. Filip Nguyen

Poděkování

Chtěl bych poděkovat Mgr. Filipovi Nguyenovi za odborné vedení práce a za cenné rady při psaní tohoto textu.

Rád bych poděkoval Lukáši Vlčkovi za ochotné zodpovězení jakýchkoli dotazů týkajících se technologie Elasticsearch a fulltextového vyhledávání obecně. Dále Ondrovi Žižkovi za sestavení velice zajímavého zadání bakalářské práce a poskytnutí nápadů při programování.

V neposlední řadě nesmím zapomenout poděkovat celému týmu *JBoss Data Grid* ve společnosti *Red Hat*, který mi vždy ochotně poradil a toleroval mou občasnou nepřítomnost v práci z důvodu psaní této práce.

Shrnutí

Cílem této práce je implementace fulltextového vyhledávání v systému pro správu požadavků eShoe za využití technologie Elasticsearch. První část popisuje teorii fulltextového vyhledávání a dostupné technologie pro jeho implementaci na platformě Java. Druhá část se věnuje návrhu a implementaci vyhledávání a importu dat do systému.

Klíčová slova

fulltextové vyhledávání, Elasticsearch, Apache Lucene, systém pro správu chyb, issue tracker, Elasticsearch-Annotations

Obsah

1 Úvod

Vývoj počítačového software je složitý proces. Při vývoji vzniká mnoho požadavků, které je potřeba evidovat, např. vzniklé chyby, žádosti o novou funkčnost apod. Pro usnadnění evidence těchto požadavků vznikly systémy označované jako systémy pro správu požadavků (*issue tracker*). Tyto systémy si kladou za cíl pomoci zajistit kvalitu produktu při vývoji, a to tak, že budou sdružovat veškeré informace o požadavcích na jednom místě a poskytnou prostředky pro jejich správu. Jedním z nich je systém eShoe.

eShoe (název odvozen od výslovnosti anglického slova „*issue*“) je pracovní název pro nově vznikající volně šiřitelný systém pro správu požadavků na platformě Java. Jeho kostra je výstupem diplomové práce Moniky Gottvaldové [6] a je hostován na serveru GitHub[1]. V současné době je eShoe ve fázi prototypu, jenž se neustále vyvíjí.

Čím je projekt větší, tím více při jeho vývoji požadavků vzniká a nastává problém, jak se ve velkém množství požadavků efektivně orientovat. Toho je většinou docíleno vyhledáváním, které umožní na základě zadané fráze najít relevantní výsledky.

1.1 Cíle práce

Cílem této práce je rozšířit existující implementaci systému pro správu požadavků eShoe o možnost fulltextového vyhledávání, které bude splňovat základní požadavky na použitelnost a umožní efektivní vyhledávání v požadavcích.

Hlavní cíl lze rozdělit na několik menších částí. První část spočívá v seznámení se s volně šiřitelnými technologiemi pro implementaci fulltextového vyhledávání na platformě Java. Je nutné pochopit, jak fulltextové vyhledávání funguje, jaké možnosti nabízí a jak lze jeho vlastností využít k co nejpoužitelnějšímu vyhledávání v systému.

Druhá část cíle obsahuje návrh a implementaci fulltextového vyhledávání v systému eShoe. Je zapotřebí vyřešit proces indexace, tedy zpřístupnění uživatelských dat pro potřeby fulltextového vyhledávání, což zahrnuje specifikace dotazů, na které je systém schopen odpovědět. Poté je nutné vytvořit prostředek pro pokládání dotazů uživatelem, aby mohl fulltextové vyhledávání v systému používat.

Posledním z cílů je vytvořit mechanismus pro importování reálných dat z existujícího systému pro správu požadavků Red Hat Bugzilla.

---

1. `https://github.com/MonikaGottvaldova/eShoe`

1.2 Struktura práce

Tato sekce stručně popisuje kapitoly, které se zabývají jednotlivými cíly této práce.

Kapitola *Vyhledávání* poskytuje úvod do vyhledávání pomocí SQL a uvádí problémy, které takto řešené vyhledávání může mít. Dále nabízí alternativní řešení v podobě fulltextového vyhledávání. Zaměřuje se na popis vlastností fulltextového vyhledávání a jeho základních principů.

Kapitola *Dostupné technologie* se věnuje třem dostupných technologiím pro implementaci fulltextového vyhledávání na platformě Java: *Apache Lucene*, *Hibernate Search* a *Elasticsearch*. Velký důraz klade na popis Apache Lucene, neboť ostatní zmíněné technologie na ní staví. Podrobněji rozebírá možnosti analýzy textu pomocí Apache Lucene a denormalizaci entit v Hibernate Search.

Kapitola *Analýza* je první kapitolou věnující se samotné realizaci vyhledávání v eShoe. Na základě zadání práce extrahuje požadavky na vyhledávání a import dat a vytváří z nich specifikaci. Na základě specifikace uvádí možná řešení jednotlivých bodů a navrhuje jedno z nich, které je vybráno k implementaci.

V rámci návrhu je rozhodnuto o naprogramování dedikovaného nástroje pro zvládnutí indexace nazvaného *Elasticsearch-Annotation*, který je podrobně popsán v kapitole *Elasticsearch-Annotations*. Jsou popsány všechny jeho aspekty, od architektury, přes popis vlastností, které nabízí, až po ukázku, jak jej použít.

Předposlední kapitola *Implementace* pojednává o implementaci návrhu fulltextového vyhledávání v eShoe. Rozebírá, jak je indexace spjatá se změnami v databázi a které informace jsou pro vyhledávání zpřístupněny. Dále se zabývá vyhledáváním v systému s využitím dotazovacího jazyka speciálně vytvořeného pro projekt eShoe. Rovněž představuje prototyp uživatelského rozhraní a popisuje, jak je fulltextové vyhledávání otestováno.

V kapitole *Závěr* jsou shrnuty výsledky této práce a navržena další vylepšení jak fulltextového vyhledávání v systému eShoe, tak projektu Elasticsearch-Annotations.

2 Vyhledávání

Tato kapitola stručně popisuje způsob vyhledávání skrze SQL v relačních databázích a uvádí jeho nedostatky. Poté se detailněji věnuje jedné z možností jejich řešení, a to fulltextovým vyhledáváním. Uvádí nezbytnou teorii k pochopení principů, jak fulltextové vyhledávání funguje.

2.1 Vyhledávání v textu pomocí SQL

Ve většině Java aplikací je vyhledávání implementováno pomocí technologií, které poskytuje datové úložiště. Protože jsou relační databáze obvykle datovým úložištěm, k implementaci vyhledávání se využívá jazyk SQL [2]. SQL nabízí pro vyhledávání v datech pouze dva způsoby: porovnání obsahu buňky a operátor `LIKE` [8].

Porovnání obsahu buňky funguje na principu úplné shody obsahu. Ukázka 2.1 uvádí příklad dotazu, který vybírá záznamy z tabulky `Lide`, které mají hodnotu atributu `jmeno` rovnou „Bruce Banner“.

```
SELECT * FROM Lide WHERE jmeno = 'Bruce Banner'
```

Ukázka 2.1: Jednoduché použití SQL pro vyhledávání pomocí úplné shody obsahu pole

Nejsou vybrány žádné jiné záznamy, přestože by obsah atributu `jmeno` byl např. „Bruce Banners“ či dokonce ani „Bruce Banner “ (přebytečná mezera na konci). Výhodou tohoto řešení je efektivita a jednoduchost – jediná nutná operace je pouze porovnání dvou řetězců, žádné dodatečné zpracování není potřeba.

```
SELECT * FROM Lide WHERE jmeno LIKE '%Banner'
```

Ukázka 2.2: Použití SQL operátoru LIKE

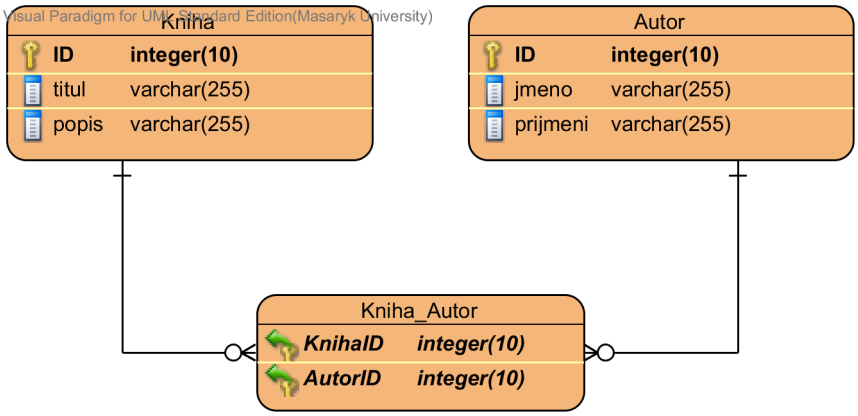
Trochu více sofistikovaným způsobem je operátor `LIKE`, který umožňuje (v omezené míře) používat vyhledávání pomocí vzoru (*pattern matching*). Podporovány jsou tzv. zástupné symboly (*wildcards*), jenž mohou mít v tomto kontextu jiný význam než jen právě daný znak, např. symbol % (procento) zastupuje libovolnou sekvenci znaků (třeba i žádnou) nebo znak _ (podtržítko) libovolný, ale právě jeden znak. Ukázka 2.2 uvádí příklad SQL dotazu, jenž vrací všechny záznamy z tabulky `Lide`, jejichž jméno končí na „Banner“.

S použitím operátoru `LIKE` je možné získat jak lidi se jménem „Bruce Banner“, tak i „Richard Banner“.

2.2 Problémy vyhledávání pomocí SQL

Předchozí kapitola představila základní způsoby vyhledávání pomocí SQL. Tato kapitola se věnuje problémům, na které může vyhledávání pomocí SQL narazit a nedokáže si s danou situací poradit buď vůbec, nebo pouze neefektivně.

Pro demonstraci problémů na příkladech uvažujme existenci jednoduché relační databáze s následujícím schématem (obrázek 2.1).



Obrázek 2.1: Datový model ukázkové databáze

Vyhledávání přes několik tabulek Uživatel zadal do vyhledávacího políčka nějaký řetězec, na jehož základě očekává odpovídající výsledky. Vyvstává otázka, kde má systém zadanou frázi hledat. V případě uvedené modelové databáze pravděpodobně v nadpisu, popisu, ve jméně a příjmení autora, všude tam se mohlou nacházet informace, které uživatel hledal.

SQL nyní musí prohledat všechny zadané sloupce, které se však mohou nacházet v různých tabulkách, což vede ke spojování tabulek. Možný příklad výsledného dotazu uvádí ukázka 2.3.

```
SELECT *
FROM Kniha kniha
LEFT JOIN kniha.autor autor
WHERE kniha.titul = ? OR kniha.popis = ? OR
autor.jmeno= ? OR autor.prijmeni= ?
```

Ukázka 2.3: SQL dotaz vyhledávající přes několik tabulek

I při relativně jednoduchém požadavku (vyhledávání probíhá pouze ve čtyřech sloupcích) je výsledný dotaz poměrně složitý. Pokud uživatel má mít možnost využívat komplexnější dotazy, je otázka generování odpovídajích SQL dotazů netriviální. Při složitějších dotazech je často nutné spojit více tabulek, což může vést k problémům s efektivitou [2, s. 9].

Vyhledávání jednotlivých slov Kapitola 2.1 ukázala, že SQL dokáže vyhledat v jednotlivých sloupcích přesně zadanou frázi. Je ovšem velice nepravděpodobné, že sloupce v databázi budou obsahovat přesně stejnou frázi, hledání jednotlivých slov by velice zvýšilo pravděpodobnost nálezu [2, s. 9]. SQL však žádnou takovou funkcionalitu na dělení vět neposkytuje, je tedy nutné si větu předpřipravit explicitně (tj. rozdělit na slova), a poté spouštět vyhledávácí dotaz pro každé slovo zvlášť. Následně výsledky nějakým způsobem sloučit. Takové řešení však nebude dostatečně efektivní [2, s. 10].

Filtrace šumu Některá slova ve větách nenesou vzhledem k vyhledávání žádnou informační hodnotu, např. spojky, předložky či ještě lepším příkladem mohou být anglické neurčité členy. Taková slova se nazývají šum (*noise*). Dále se pak některá slova v určitém kontextu šumem stávají, např. slovo „kniha“ v internetovém knihkupectví [2, s. 9]. Jelikož šum nenese žádnou informační hodnotu, měl by být při hledání ignorován. SQL opět neposkytuje žádný prostředek k řešení tohoto problému.

Vyhledávání příbuzných slov Je velice žádoucí, aby se uživatel při vyhledávání mohl zaměřit pouze na význam hledaného slova, nikoliv na jeho tvar. Nemělo by záležet na tom, zda je vyhledávanou frází „fulltextové hledání“ nebo „fulltextových vyhledávání“, význam těchto frázi je stejný. Jinak řečeno, vyhledávání by mělo brát v potaz i slova odvozená, se stejným kořenem. Ještě pokročilejším požadavkem by mohla být možnost zaměňovat slova s jejich synonymy, např. „upravit“ a „editovat“ [2, s. 10].

SQL nenabízí možnost k řešení těchto požadavků, klíčem by mohl být slovník příbuzných slov a synonym a pokusit se vyhledávat i podle něj.

Oprava překlepů Uživatel je člověk a jako člověk je omylný a dělá chyby. Vyhledávání by to mělo brát v potaz a snažit se tyto překlepy opravit či uhodnout, co měl uživatel na mysli. Když v internetovém knihkupectví uživatel hledá knihu „Fulltextové vyhledávání“ a omylem zadá do vyhledávacího pole „Fulltetové vyhledávání“, je žádoucí, aby i přes tento překlep knihu našel [2, s. 10].

Relevance Pravděpodobně největším problémem vyhledávání pomocí SQL je absence jakéhokoliv mechanismu pro určení míry shody (*relevance*) záznamu se zadaným dotazem [2, s. 10]. Předpokládejme, že v internetovém knihkupectví napsal autor „John Smith“ 100 knih, jednu o fulltextovém vyhledávání a zbytek naprosto nesouvisející s informatikou. Dále několik dalších autorů rovněž napsalo publikace na téma fulltextového vyhledávání.

Pokud uživatel ví, že je autorem John Smith a kniha je o fulltextovém vyhledávání, očekává, že na vyhledávací dotaz „John Smith fulltextové vyhledávání“ obdrží nejdříve právě chtěnou knihu, a poté teprve knihy ostatní od našeho autora či další knihy o fulltextovém vyhledávání, jelikož hledaná kniha „nejvíce“ odpovídala položenému dotazu.

2.3 Fulltextové vyhledávání

Předchozí kapitola demostrovala, jaké problémy má vyhledávání pomocí SQL. Nyní bude představeno možné řešení – fulltextové vyhledávání.

Fulltextové vyhledávání (někdy také *fulltext* nebo *full-text*) je speciální způsob vyhledávání informací v textu. Vyhledávání probíhá porovnáváním s každým slovem v hledaném textu. Jelikož počet slov v textu může teoreticky neomezený a jelikož je nutné, aby vyhledávání bylo co nejrychlejší, funguje fulltextové vyhledávání ve dvou fázích: *indexace* a *hledání* [2, s. 11].

2.3.1 Indexace

Indexace je hlavním krokem ve fulltextovém vyhledávání. Jedná se o proces předpřipravení vstupních dat, jejich přeměnu na co nejvíce efektivní datovou strukturu, aby se v ní dalo snadno a rychle vyhledávat. Této datové struktuře, která je výstupem indexace, se říká *index* [9, s. 11].

Index si lze představit jako datovou struktu umožňující přímý přístup ke slovům v něm obsažených. Základním úkolem je rozdělit text do slov a pomocí přímého přístupu umožnit velice efektivně zjistit, kde se dané slovo vyskytuje. Toho je typicky (např. v Apache Lucene) dosaženo *invertovaným indexem* [9, s. 35].

Pouhým rozdělením do slov však možnosti předpřipravení textu nekončí a může být zapojena složitá analýza. V praxi (např. v Apache Lucene [9, s. 35]) je celý text předáván analyzátoru, který může index libovolně budovat, a tím ho lépe připravit na nadcházející dotazování, a umožnit mu odpovídat na složitější dotazy. Typickým příkladem možné analýzy je úprava podstatných jmen do základního tvaru (např. z množného čísla na jednotné), přidání synonym do indexu či získávání statistiky o četnosti výskytu daného slova.

2.3.2 Hledání

Stejně jako indexace je i hledání proces rozdělený do několika kroků. Za předpokladu, že je fulltextovému vyhledávání předán dotaz, je první na řadě analýza tohoto dotazu podobně, jako je tomu v případě indexace (rozdělení do slov, převedení na kořen apod). Tato fáze je klíčová, neboť se na dotaz musí použít stejné operace jako při indexaci. Jak je uvedeno v předchozí kapitole, index zaznamenává všechny výskyty právě stejného slova. V případě, že se výsledná slova neshodují, slovo v indexu nebude nalezeno a výsledek dotazů tak nebude správný [2, str. 16].

Jakmile je dotaz analyzován, přichází na řadu získání informací v indexu o slovech, které souhlasí s dotazem, např. kde se všude vyskytuje, jeho četnost apod. Důležitým faktem přitom je, že ve fulltextovém vyhledávání nemusí být odpovídající záznam načten, aby se zjistilo, že dané slovo obsahuje, mechanismus to pozná z indexu. Proto je fulltextové vyhledávání tak efektivní.

Nakonec se na základě získaných informací výsledky filtrují a přiřazuje se jim míra relevance (*skóre*). Jak se skóre počítá, záleží již na konkrétní technologii, pro představu následují typické faktory [2, str. 16]:

- V dotaze s více slovy, čím blíže jsou v textu od sebe, tím vyšší skóre.
- V dotaze s více slovy, čím více jich je v textu nalezeno, tím vyšší skóre.
- Čím častěji se slovo v dokumentu vyskytuje, tím vyšší skóre.
- Čím méně se muselo slovo analýzou upravit, tím vyšší skóre.

Pomocí vhodně nastavené analýzy a vyhledávání lze řešit všechny nedostatky, které má vyhledávání pomocí SQL, viz. kapitola 2.2. Ukazuje se, fulltextové vyhledávání je správným nástrojem k vytváření sofistikovanějšího vyhledávání, které lépe splní požadavky uživatele [2, str. 16].

3 Dostupné technologie

Pro platformu Java existuje řada dostupných volně šiřitelných vyhledávacích technologií. Tato kapitola představuje tři z nich: *Apache Lucene*, *Hibernate Search* a *Elasticsearch*.

3.1 Apache Lucene

Apache Lucene je vysoce výkonná, škálovatelná, volně šiřitelná vyhledávací knihovna napsána v jazyce Java [9, s. 6]. Autorem projektu, který vznikl v roce 1997, je Doug Cutting. Zajímavostí je, že jméno Lucene bylo vybráno podle druhého jména manželky autora [9, s. 6]. V roce 2000 zveřejnil Lucene na stránkách serveru SourceForge.com a uvolnil ji tak zdarma pro komunitu. O rok později byla adoptována organizací *Apache Software Foundation*. Od té doby je knihovna neustále vyvíjela a v dubnu roku 2014 je aktuálně dostupná ve své nejnovější verzi 4.7.1 [9, s. 6].

Již několik let je Lucene nejpopulárnější vyhledávací technologií zdarma. Díky své popularitě se dočkala i přepsání do jiných jazyků než je Java jako například Perl, Python, Ruby, C/C++, PHP a C# (.NET) [9, s. 3]. Projekt je stále aktivně vyvíjen s širokou komunitní základnou.

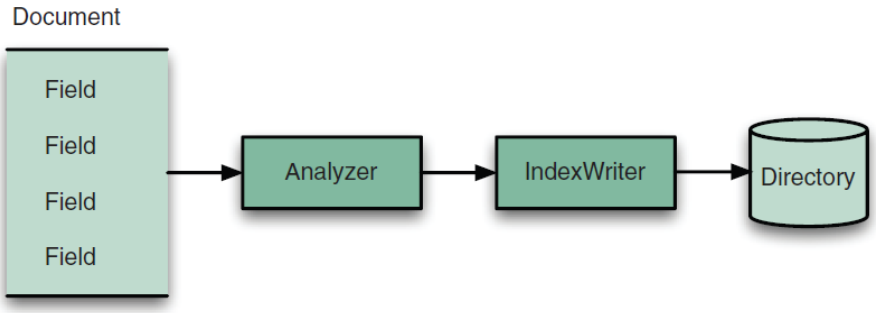
Apache Lucene není hotová vyhledávací aplikace, je to knihovna, nástroj, poskytující všechny potřebné prostředky, aby mohla být taková aplikace pro vyhledávání naprogramována. Nabízí rozhraní pro vytváření, úpravu indexu, zpracování dat před indexací a tvorbu, úpravu dotazů a mnoho dalšího. O zbytek úkonů se musí programátor postarat sám, z čehož vyplývají hlavní výhoda (robustnost, univerzálnost použití), ale také hlavní nevýhoda (složitost nasazení) [9, s. 7].

Používání Apache Lucene je poměrně náročné, což vychází z její univerzálnosti [5] – uživatel (programátor) má mnoho možností, jak výslednou vyhledávací aplikaci nakonfigurovat, a tím i vyladit. Kvůli této složitosti začaly vznikat další technologie, které staví na Apache Lucene, snaží se schovat podrobná, a tedy i méně často používaná, nastavení do pozadí a umožnit tak vývojáři se v technologii rychle zorientovat se zachováním původní síly Apache Lucene. Takových technologií existuje více (Apache Solr, Hibernate Search, Elasticsearch a další) a je dobré při jejich používání vědět, jak funguje Apache Lucene na nižší úrovni, neboť tyto technologie ji přímo využívají. Z toho důvodu je architektura Apache Lucene podrobněji představena v následujících kapitolách.

3.1.1 Architektura

Pro lepší pochopení, jak Apache Lucene funguje, následuje výčet základních tříd, které se podílejí na procesu indexace [9, s. 26]:

- `IndexWriter`
- `Directory`
- `Analyzer`
- `Document`
- `Field`



Obrázek 3.1: Architektura indexační části Apache Lucene, převzato z [9, s. 26]

Třída `IndexWriter` je vstupní bod indexace. Je zodpovědná za vytváření nového indexu a přidávání dokumentů do indexů existujících. Neslouží k vyhledávání ani modifikaci indexu. `IndexWriter` musí znát umístění, kam má svůj index uložit a k tomu slouží `Directory`.

`Directory` je abstraktní třída reprezentující fyzické umístění indexu.

Předtím než je text indexován, je předán analyzéru, implementaci abstraktní třídy `Analyzer`. Analyzér je zodpovědný za extrakci *tokenů* – jednotek, které následně budou uloženy do indexu [9, s. 116] – a eliminaci všeho ostatního. Analyzér je patrně nejdůležitější komponenta indexace, rozhoduje, které tokeny budou uloženy a dokáže je libovolně modifikovat. Apache Lucene obsahuje již některé praktické implementace třídy `Analyzer`, které jsou nejběžnější. Některé z nich se například zabývají odstraněním šumu z textu, další převedením všech písmen na malá apod. Proces analýzy je podrobněji rozebírán v další kapitole, neboť je to klíčová vlastnost Apache Lucene, kterou dědí i ostatní technologie na ní postavené.

`Document` je kolekcí polí (*fields*), tedy kontejnerem pro objekty `Field`, které nesou textová data.

`Field` je základní jednotka, která obsahuje vlastní indexovaný text.

Jak je uvedeno v kapitole 2.3, fulltextové vyhledávání je rozděleno na dvě části – indexaci a vyhledávání. Protože však technologie postavené na Apache Lucene poskytují své vlastní vyhledávací API, a tím skrývájí vyhledávání v Apache Lucene úplně, nebudou detaily architektury vyhledávání v Apache Lucene dále rozebírány.

3.1.2 Indexace

Předchozí kapitola stručně popisuje architekturu indexační části Apache Lucene. Následuje bližší vysvětlení, jak spolu jednotlivé části spolupracují.

Základní jednotkou indexu Apache Lucene jsou *dokumenty* a *pole* [9, s. 32]. Dokument je kolekcí polí, která pak obsahují indexovaný text. Každé pole má své jméno, textovou nebo binární hodnotu a seznam operací, které popisují, co má Apache Lucene dělat s hodnotou pole při vytváření indexu. Aby mohla být uživatelská data (položky z databáze, PDF dokumenty, HTML stránky apod.) indexována, je potřeba je převést do formátu Apache Lucene dokumentu. Při indexaci nástroj Apache Lucene nezohledňuje sémantiku obsahu. Převedením struktury uživatelského obsahu do struktury Lucene dokumentů, do dvojic klíč hodnota, se zabývá *denormalizace*.

Denormalizace Denormalizace je proces převedení libovolné struktury dat do jednoduchého formátu klíč hodnota [9, s. 34]. Například v databázi jsou jednotlivé záznamy spojovány cizími klíči mezi různými tabulkami, vzniká mezi nimi vztah, jednotlivé záznamy se na sebe odkazují. V dokumentech Apache Lucene však žádná možnost odkazu či spojení není, jediný akceptovaný formát je klíč hodnota. Programátor musí vyřešit problém, jak data, ve kterých chce vyhledávat, denormalizuje. Apache Lucene nechává tuto část zcela na programátorovi, na rozdíl od na ní postavených technologiích jako např. Hibernate Search (viz. kapitola 3.2).

Jednou z dalších důležitých věcí, které je potřeba vědět o Apache Lucene dokumentech, je absence jakéhokoliv pevného schématu jako např. u databází. Tato vlastnost se někdy označuje jako *flexibilní schéma* [9, s. 34]. Umožňuje například iterativně budovat index, protože nově nahraný index může být naprosto rozdílný, obsahovat jiná pole, od předchozího. Rovněž je možné do jednoho dokumentu uložit indexy reprezentující zcela jiné entity.

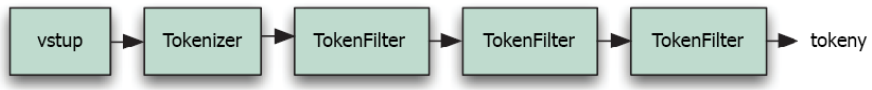
3.1.3 Analýza

V předchozích kapitolách jsou uvedeny základní principy, na kterých Apache Lucene staví indexy, v následujícím textu je podrobněji rozebrána nejdůležitější část indexačního procesu – analýza.

Nejdříve jsou vstupní data denormalizována do dokumentů, které jsou naplněny poli. Analýza v Apache Lucene je proces převedení textových polí do základní indexované podoby – do *termů* [9, s. 28]. Analyzérem nazýváme komponentu, která zajišťuje analýzu. Ukažme si několik typických příkladů, co analyzéry dělají [9, s. 110]:

- extrakce slov
- zahození interpunkce
- převod na malá písmena (*normalizace*)
- redukce šumu
- převod slova na jeho kořen (*stemming*)
- převod slova na základní tvar (*lemmatizace*) a další

Samozřejmě je možné naprogramovat vlastní analyzér, některé úkony jsou však natolik běžné (jako například výše uvedené), že Apache Lucene přichází s několika zabudovanými analyzéry. Analyzéry pro svou funkčnost využívají dva další typy komponent: *tokenizéry* (potomky třídy `Tokenizer`) a *filtry* (potomky třídy `TokenFilter`) [9, s. 115]. Obě dědí od abstraktní třídy `TokenStream`, zabývají se však rozdílnou částí zpracování vstupu. Tokenizér čte vstup a vytváří tokeny. Filtr bere jako vstup tokeny a na jejich základě vrátí nově vytvořený seznam tokenů. Tento seznam může vzniknout přidáním nových tokenů, úpravou existujících či odstraněním některých z nich.



Obrázek 3.2: Použití tokenizéru a filtrů, převzato z [9, s. 117]

Typické využití, kterého se drží i zabudované analyzéry, vypadá následovně. Analyzéru je předán vstup. Ten je rozdělen na tokeny pomocí jednoho tokenizéru. Následně jsou tokeny předány jednomu či více filtrům, čímž

vznikne finální kolekce tokenů, která je předána jako výsledek analýzy (obrázek 3.2).

Uveďme příklady zabudovaných tokenizérů [9, s. 118]:

- `WhitespaceTokenizer` - nový token je ohraničen bílými znaky
- `KeywordTokenizer` - předá celý vstup jako jeden token
- `LowerCaseTokenizer` - nový token je ohraničen jinými znaky než písmeny
- `StandardTokenizer` - pokročilý tokenizér založený na sofistikovaných gramatických pravidlech, dokáže rozpoznat např. e-mailové adresy a předat je jako jediný token

```
WhitespaceAnalyzer:
[The] [quick] [brown] [fox] [jumped] [over] [the]
[lazy] [dog]

SimpleAnalyzer:
[the] [quick] [brown] [fox] [jumped] [over] [the]
[lazy] [dog]

StopAnalyzer:
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]

StandardAnalyzer:
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]
```

Ukázka 3.1: Použití zabudovaných analyzérů pro větu *„The quick brown fox jumped over the lazy dog“*, převzato z [9, str. 111]

Představme rovněž i několik základních filtrů [9, s. 118]

- `LowerCaseFilter` - převede token na malá písmena
- `StopFilter` - odstraní tokeny, které se nacházejí v předaném seznamu
- `StandardFilter` - navržen pro spolupráci s tokenizérem `Standard Tokenizer`, odstraňuje tečky z akronymů a „'s“ (apostrof následovaný písmenem s)

Aby byl výčet kompletní, následuje přehled zabudovaných analyzérů. Zabudované analyzéry jsou v podstatě kombinací tokenizérů a filtrů, z čehož je následně jasná jejich funkce [9, s. 112].

- `WhitespaceAnalyzer` - dělí text na tokeny pomocí tokenizéru `WhitespaceTokenizer`
- `SimpleAnalyzer` - zpracovává vstup pomocí tokenizéru `LowerCaseTokenizer`
- `StopAnalyzer` - kombinace tokenizéru `LowerCaseTokenizer` a filtru `StopFilter`, kterému je předán seznam často se vyskytujících nevýznamových slov v angličtině (členy *a*, *an*, *the*, apod.)
- `StandardAnalyzer` - nejpropracovanější zabudovaný analyzér, využívá `LowerCaseTokenizer`, `StopFilter`, navíc však přidává i propracovanou logiku, která dokáže např. rozeznat e-mailové adresy, názvy společností atd.

```
WhitespaceAnalyzer:
[XY&Z] [Corporation] [-] [xyz@example.com]

SimpleAnalyzer:
[xy] [z] [corporation] [xyz] [example] [com]

StopAnalyzer:
[xy] [z] [corporation] [xyz] [example] [com]

StandardAnalyzer:
[xy&z] [corporation] [xyz@example.com]
```

Ukázka 3.2: Použití zabudovaných analyzérů pro větu „*XY&Z Corporation - xyz@example.com*“, převzato z [9, str. 112]

Popis analýzy je zakončen ukázkami, jaké tokeny jednotlivé zabudované analyzéry vytvoří ze dvou anglických vět (ukázky 3.1 a 3.2).

```
@Entity
@Indexed
public class Person {

        @Id     @GeneratedValue
        @DocumentId
        private Long id;

        @Field
        private String firstName;

        @Field
        private String lastName;
}
```

Ukázka 3.3: Zpřístupnění entity pro vyhledávání v Hibernate Search

3.2 Hibernate Search

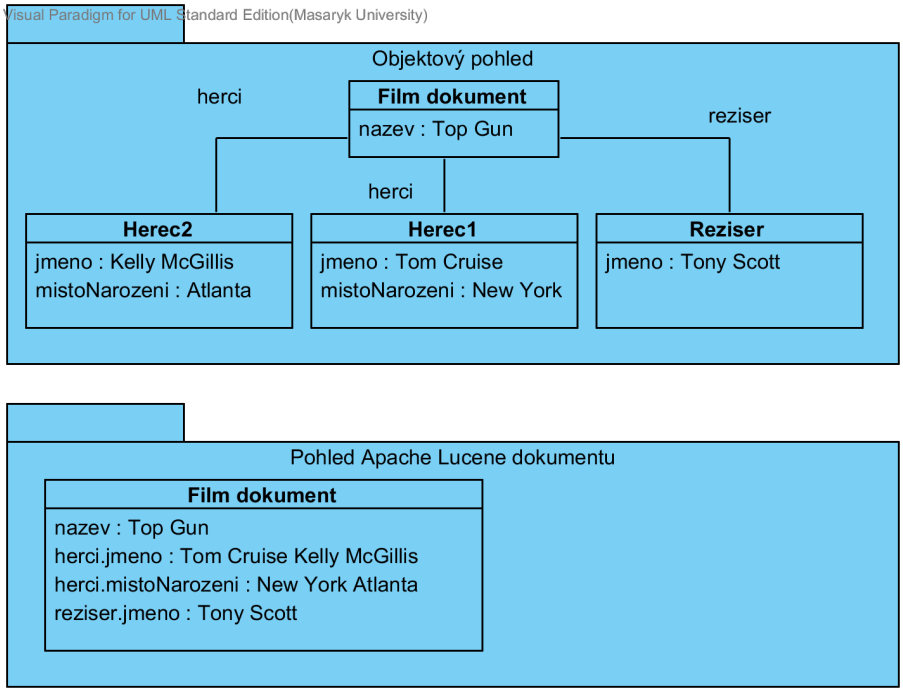
Po rozmachu technologie *objektově relačního mapování* (ORM, *Object-Relational Mapping*) na platformě Java a její nejznámější implementace Hibernate Core [2, s. 29] bylo nutné dát tomuto nástroji možnosti fulltextového vyhledávání. Hibernate Search je volně šiřitelná knihovna napsaná Emmanuelem Bernardem, která doplňuje Hibernate Core o možnosti fulltextového vyhledávání pomocí kombinace s Apache Lucene [2, s. 29]. Hibernate Search se snaží integrovat funkčnost Apache Lucene do Hibernate ORM. S minimálním úsilím řeší převod objektového datového modelu do podoby přijatelné pro Apache Lucene, čímž výrazně usnadňuje její použití.

Ukázka 3.3 demonstruje, jak snadno lze s využitím Hibernate Search zpřístupnit entitu pro fulltextové vyhledávání. Entita musí být označena anotací `@Indexed` [2, s. 38]. Dále je přidána anotace `@DocumentId` k primárnímu klíči, a poté označeny atributy, podle kterých bude vyhledáváno anotací `@Field` [2, s. 38]. V momentě uložení entity Hibernate Search vyřeší přidání uvedených atributů do indexu, tedy denormalizuje entitu. Jelikož je to však pod povrchem stále Apache Lucene, jsou k dispozici všechny možnosti, které nabízí, nyní v přístupnější formě.

Integrace s Hibernate Core elegantně řeší jeden podstatný problém, který vyvstává s použitím čistě Apache Lucene – synchronizaci fulltextového indexu a obsahu databáze. Jsou to v zásadě dvě zcela oddělená datová úložiště, která

spolu úzce souvisí. Pokud je použita přímo Apache Lucene, je nutné se po manipulaci s objektem v databázi explicitně postarat o úpravu příslušného indexu, což je pro programátora práce navíc. Oproti tomu Hibernate Search je navázán na události Hibernate Core, tudíž při úpravě objektu v databázi je automaticky spuštěn proces aktualizace indexu, aby spolu byla data v databázi a fulltextovém indexu synchronizována [2, s. 24].

Denormalizace v Hibernate Search Jak uvádí odstavec 3.1.2, při použití Apache Lucene je nezbytné strukturu Java objektů nějakým způsobem rozložit do jednoduchého formátu klíč hodnota. Protože Hibernate Search staví na Apache Lucene, je toto nutné i při jeho použití. Hibernate Search však nenechává denormalizaci na programátorovi, realizuje ji sám automaticky. Následující text uvádí, jaké problémy nastávají a jak je Hibernate Search řeší.



Obrázek 3.3: Hibernate Search denormalizuje vztahy, aby bylo možné podle nich vyhledávat.

Denormalizace atributů primitivních typů je triviální, hodnota atributu je přímo zavedena do indexu [2, str. 76]. V případě atributů neprimitivního

datového typu (uživatelsky definované typy, kolekce, mapy, atd.) je situace složitější. Tyto objekty mezi sebou vytvářejí vztah. Apache Lucene bere v úvahu pouze jediný dokument při hodnocení relevance vůči dotazu a nemá možnost jakýmkoliv způsobem vztahy mezi dokumenty vyjádřit [2, str. 105]. Aby bylo možné podle těchto objektů vyhledávat, je nutné všechny informace o odkazovaných objektech přiložit do stejného indexového dokumentu. Obrázek 3.3 ukazuje, jak denormalizaci řeší Hibernate Search.

Posledním problémem, který je potřeba vyřešit, je automatická úprava indexu asociovaných objektů. Pokud je např. změněno jméno herce (viz. obrázek 3.3), Hibernate Search musí poznat, ke kterým objektů je herec přiřazen a jejich index znovu vybudovat. Vztahy mezi objekty se dají rozdělit na dva typy – jeden objekt je vnořený (*embedded*) do druhého, nebo jsou spolu související (*associated)* [2, str. 107, 110].

Jednodušším ze vztahů je, když jeden objekt je vnořený do druhého. To znamená, že životní cyklus vnořené entity je naprosto závislý na odkazované. Bez odkazované entity nemá vnořená entita žádný smysl sama existovat a je k ní přistupováno pouze v souvislosti s „rodičovskou“ entitou. Příkladem může být existence entit `Movie` (znázorňující film v kině) a `Rating` (reprezentující hodnocení filmu od jednoho fanouška). Samostatné hodnocení nemá žádný smysl bez filmu, nemá smysl jej vyhledávat, tudíž životní cyklus je spjat s entitou filmu. Při editaci filmu je úprava indexu jednoduchá – Hibernate Search si poznačí, že musí znovu vytvořit index pro související film, v rámci něhož se aktualizuje i hodnocení [2, str. 108].

Druhým případem je, když jsou entity nezávislé a jedna dává i bez druhé smysl, např. herec a film. Uživatel může chtít vyhledávat film podle herců, teří v něm hrají, zároveň však může požadovat vyhledávání čistě mezi herci jen podle jejich atributů, např. roku narození. Film i herec tedy musí být uloženy v samostatných dokumentech a při úpravě herce se musí aktualizovat index všech filmů, ve kterých se herec objevil [2, str. 110].

Hibernate Search řeší výše uvedené problémy automaticky za programátora na základě anotací, a tím značně usnadňuje celý proces indexace.

3.3 Elasticsearch

Elasticsearch je distribuovaný vyhledávácí a analytický nástroj pracující v reálném čase [5]. Historie této technologie se začala psát v roce 2004, kdy Shay Banon vytvořil *Compass*. Postupným vývojem a změnou požadavků však dospěl k názoru, že aby se mohl Compass stát distribuovanou technologií, bylo by zapotřebí ho značnou část přepsat. Rozhodl se proto naprogramovat zcela nový nástroj, který měl být již od počátku distribuovaný. První verze Elasticsearch byla vydána v únoru 2010 [4].

Elasticsearch využívá Apache Lucene, nicméně architekturou je Elasticsearch middleware, který je při nasazení distribuován na množinu serverů. Každý tento server využívá Apache Lucene. Výsledkem je zapouzdření funkcionality Apache Lucene a poskytutí API v jednodušší formě. Velký důraz je kladen právě na distribuovanost celého systému, proto je Elasticsearch vysoce škálovatelný, schopný vytvořit klastr několika stovek serverů, a tím zajistit vysoký výkon i při několika petabajtech dat, což patří mezi hlavní přidanou hodnotu navrch k Apache Lucene [5].

Základním způsobem komunikace se serverem Elasticsearch je REST (*Representational State Transfer*) API posílající JSON (*JavaScript Object Notation*) objekty. Tím je zajištěna nezávislost na programovacím jazyku, komunikace může probíhat přímo i z příkazové řádky. Pro některé jazyky byly již vytvořeny knihovny umožňující komunikaci skrze proprietálního klienta. Jedná se například o jazyky: Java, PHP, Python.

Za zmínku stojí, kam Elasticsearch ukládá dokumenty. Struktura Elasticsearch se podobá modelu relační databáze, proto je pro lepší představu uvedena paralela. Elasticsearch klastr může obsahovat několik *indexů* (*index*, obdoba databáze), indexy obsahují *typy* (*type*, paralela s dabázovou tabulkou). Typy mohou držet několik *dokumentů* (*document*, podobnost s řádkem v tabulce), které mají několik *polí* (*field*, analogie ke sloupci tabulky). Pro přístup k hodnotám se pak využívá cesty $< index > / < typ > / < id\_dokumentu > / < pole >$ [5].

Stejně jako Hibernate Search je i Elasticsearch zaobalená knihovna Apache Lucene s několika přidanými hodnotami [5].

4 Analýza

Předchozí kapitoly představily fulltextové vyhledávání a dostupné technologie pro jeho implementaci na platformě Java. Následující text se věnuje návrhu implementace fulltextového vyhledávání v systému správy požadavků *eShoe*.

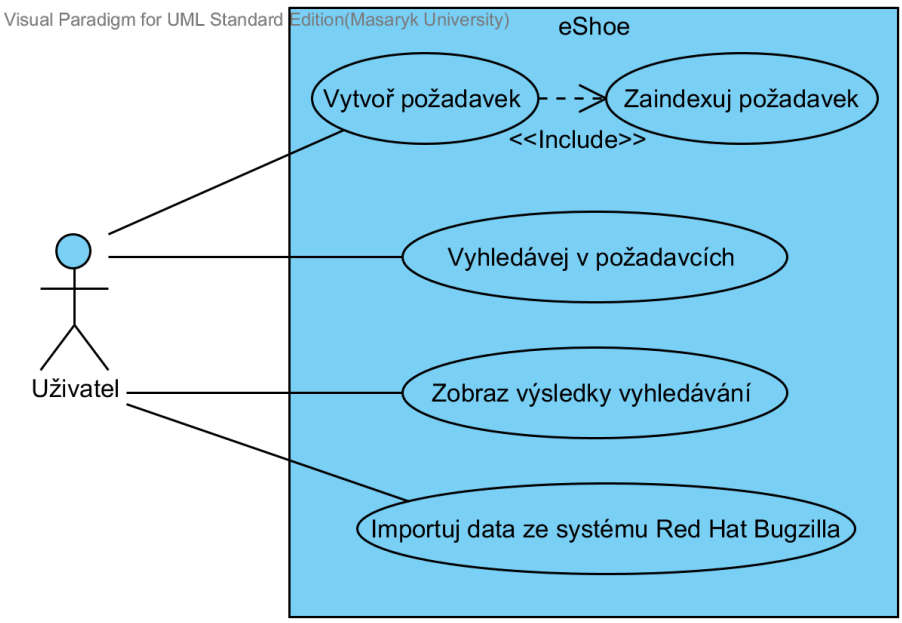
4.1 Specifikace požadavků

Hlavním úkolem je vytvořit funkční fulltextové vyhledávání v systému eShoe. Tento požadavek lze rozdělit do několika menších částí shrnutých v následujících bodech:

- **Provedení indexace:**
  Při modifikaci entity v databázi musí být entita rovněž vhodně uložena do indexu pro fulltextové vyhledávání, aby mohla být následně vyhledávána.
- **Vyhledávání nad indexovanými daty:**
  Na základě dotazu uživatele musí být index prohledán a vráceny relevantní výsledky.
- **Zobrazení výsledků:**
  Vytvořit jednoduché uživatelské rozhraní, které umožní pokládat dotazy a zároveň zobrazí výsledky.

Protože je nezbytné vybrat vhodné vlastnosti entit, které mají být indexovány, je nutné specifikovat dotazy, které uživatel může chtít položit a systém na ně umí vrátit požadovanou odpověď. Kromě obecného zadání fráze, na jejímž základě mají být vráceny relevantní požadavky, je žadoucí umožnit uživateli zadat vlastnosti, které musí požadavek splňovat a všechny nevyhovující odfiltrovat, například vyhledat všechny požadavky na dotaz „Failing unit tests“, ale pouze ty přiřazené k projektu „Infinispan“. Následuje výčet vlastností, které může uživatel explicitně zadat, a podle kterých systém umožní požadavky filtrovat:

- projekt, ke kterému je požadavek přiřazen
- status, v němž se požadavek nachází
- typ požadavku
- datum vytvoření požadavku



Obrázek 4.1: Diagram případů užití

- datum poslední modifikace požadavku
- jméno uživatele, který požadavek vytvořil
- jméno uživatele, kterému je přiřazeno řešení požadavku
- prioritu požadavku
- konkrétní ID požadavku

Dalším bodem ze zadání je vytvořit mechanismus pro import dat z již existujícího systému správy požadavků Red Hat Bugzilla do systému eShoe. Systém musí být schopen na základě předaného seznamu ID požadavků z onoho existujícícho systému nejprve získat všechny potřebné informace o požadavcích, a poté je namapovat na datový model systému eShoe, uložit je do databáze a fulltextového indexu.

4.2 Návrh

V předchozí sekci je uvedena specifikace, kterou musí implementace splňovat. Následující text popisuje návrh, jak budou jednotlivé body specifikace vyřešeny.

Pro implementaci fulltextového vyhledávání bude zvolena technologie Elasticsearch z následujících důvodů. Jedná se o technologii, která je postavena na osvědčené Apache Lucene, s aktivní komunitní základnou a stálým vývojem. Je používána např. serverem GitHub[1] [5], z čehož lze usuzovat, že poskytne i dostatečný výkon. Kromě toho mi Elasticsearch přišel subjektivně nejvíce elegantní a použití Elasticsearch bylo vyžadováno zadavatelem práce.

4.2.1 Indexace

První z problémů, který je potřeba vyřešit, je zvládnutí procesu indexace, tedy denormalizaci entit do formátu, který se dá přímo předat Elasticsearch serveru. Elasticsearch příjímá JSON objekty (viz. 3.3), entity je tedy potřeba převést právě do JSON formátu. Jedním z prvních možných řešení je prostá manuální tvorba indexu z entity pomocí *get* metod, to znamená pro každou třídu vytvořit mechanismus, který v předem daném pořadí předem dané atributy získá a vytvoří z nich JSON objekt.

Nevýhoda tohoto řešení je zjevná – nulová flexibilita. Při každé úpravě entity je nutné dopsat odpovídající mechanismus, který upravený atribut denormalizuje. Navíc je toto řešení udělané přesně na míru tomuto projektu, resp. přesně danému datovému modelu, tudíž není znovupoužitelné do budoucna. Výhoda je ovšem rovněž zřejmá – jednoduchost. K naprogramování takového mechanismu není potřeba víc než základní znalost jazyka Java. Z důvodu programování kódu, který by byl použitelný pouze v jednom projektu a je poměrně neelegantní, bylo toto řešení zavrhnuto a hledal jsem alternativní přístup.

Po prostudování dokumentace pro Elasticsearch jsem zjistil, že v současné době není pro jazyk Java naprogramován žádný nástroj, jenž by usnadnil denormalizaci objektů, jako je tomu např. v Hibernate Search (viz. 3.3). Bylo proto rozhodnuto, že podobný mechanismus naprogramuji první a poskytnu podobnou funkcionalitu i pro Elasticsearch.

Základní myšlenkou je použití anotací, které jsou zárukou vysoké elegance a jednoduchosti použití. Jakmile budou atributy entity označeny anotacemi, entita se předá správci indexu, který entitu denormalizuje, připojí se skrze zvoleného klienta k serveru Elasticsearch a uloží nově vytvořený dokument

1. `http://www.github.com`

do indexu opět na základě zadaných parametrů u anotací. Tento nově vzniklý projekt byl pojmenován *Elasticsearch-Annotations* (viz. 5) a je hostován na serveru GitHub[2].

Protože při vývoji projektu eShoe nebyla potřeba servisní vrstva aplikace, tak zcela chybí. Nyní je však nutné navázat operace změny indexu na změny v databázi a servisní vrstva je místem, kde by se to dalo realizovat. Proto musí být servisní vrstva nově vytvořena. Vyřešení indexace pak spočívá v označení entit anotacemi a zavolání správce indexu na servisní vrstvě.

Jelikož je proces denormalizace zcela vyčleněn do samostatného projektu Elasticsearch-Annotations a zakomponování indexačního mechanismu do datového modelu eShoe vyžaduje minimální úpravy, je toto řešení elegantní a vysoce znovupoužitelné. Bylo proto rozhodnuto se ubírat tímto směrem a tento návrh implementovat.

4.2.2 Vyhledávání

Jakmile jsou data zaindexována, lze přistoupit k vlastnímu vyhledávání. Jedná se hlavně o způsob tvorby dotazu pro Elasticsearch server. Kapitola 4.1 uvádí systémem podporované typy dotazů a patrně v budoucnu přibydou další. Typů dotazů je několik a poměrně různorodých, proto je potřeba vymyslet robustní způsob zadávání dotazů, který by se mohl dále rozšiřovat.

Základním způsobem je tvorba dotazu přes uživatelské rozhraní, na pozadí by se postupně budoval objekt reprezentující dotaz pro Elasticsearch. Je potřeba však brát v potaz uživatele, kteří by chtěli vyhledávání používat skrze nějaký automatizovaný mechanismus, nikoliv ručně klikáním na komponenty v GUI. Pro ty by automatizace tvorby dotazu nebyla jednoduchá.

Další možností je vytvořit vlastní dotazovací jazyk, jako má např. *Attlasian JIRA*[3]. Uživatel by dostal možnost vytvářet dotazy buď klikáním v GUI, nebo by rovnou mohl napsat dotaz v dotazovacím jazyce. V budoucnu pak není problém naprogramovat přístupový bod např. skze REST API, který umožní v systému vyhledávat pomocí těchto dotazů. Tím by se proces automatizace velice zjednodušil. Pro některé uživatele je dokonce pohodlnější napsat dotaz rovnou v dotazovacím jazyce, pokud je dostatečně jednoduchý. Pro poskytnutí maximálně flexibility se přikláním k tomuto řešení.

Vyhledávání bude probíhat na základě dotazu vytvořeném ve vlastním dotazovacím jazyce. Uživatelské rozhraní bude sloužit jako tvůrce oněch dotazů umožňující rovněž zadávání dotazu přímo. Dotazovací jazyk bude mít následující vlastnosti:

2. `https://github.com/Holmistr/elasticsearch-annotations`
3. `https://www.atlassian.com/software/jira`

- zadat text, který se má použít jako fráze pro fulltextového vyhledávání
- zadat filtr na vlastnost entity na přesnou shodu jedné položky
- určit filtr na vlastnost entity na shodu s některou ze seznamu předaných hodnot
- předchozí body libovolně kombinovat

Uvedený jazyk by měl být co nejjednodušší a mít intuitivní syntaxi. Ukázky 4.1, 4.2 a 4.3 uvádí příklady, jak bude výsledný dotazovací jazyk vypadat.

```
text ~ "Failing unit tests"
AND project = "Infinispan"
```

Ukázka 4.1: Vyhledání fráze „Failing unit tests“ pouze u projektu „Infinispan“

```
text ~ "Failing unit tests"
AND status IN ("Unresolved", "Open")
```

Ukázka 4.2: Vyhledání fráze „Failing unit tests“ u požadavků se statusem „Unresolved“ nebo „Open“

```
text ~ "Failing unit tests"
AND project = "Infinispan"
AND status IN ("Unresolved", "Open")
```

Ukázka 4.3: Kombinace dotazů 4.1 a 4.2

4.2.3 Uživatelské rozhraní

Přestože zadání práce uživatelské rozhraní nevyžaduje, rozhodl jsem se jej zahrnout, aby byla demostrace výsledků snazší a byl poskytnut prototyp pro další rozvíjení. Účelem tohoto prototypu není v žádném případě poskytnout plnohodnotné uživatelské rohraní, neklade se tedy důraz na grafické zpracování či poskytnutí grafických komponent pro zadání všech možných filtrů.

Součástí grafického rozhraní budou tři textová pole a tři pole pro vybrání ze seznamu. Textová pole budou použita pro zadání fráze pro vyhledání a ohraničení časového úseku, kdy byl požadavek vytvořen. Textové pole pro zadání fráze bude rovněž sloužit pro zadání dotazu čistě pomocí vytvořeného dotazovacího jazyka. Pole s předem daným seznamem prvků budou sloužit ke zvolení typů požadavků, jejich statusů a projektů, ke kterým mají být

požadavky přiřazeny. Tato pole musí podporovat výběr více možností najednou. GUI rovněž umožní jednoduchý výpis nalezených požadavků a přejití na jejich detaily.

Pro implementaci bude použita technologie Apache Wicket, jelikož zbytek grafického prostředí systému je rovněž napsán pomocí ní.

4.2.4 Import dat

Součástí specifikace je i import dat z již existujícího systému pro správu požadavků. Pro tento účel byl vybrán systém *Red Hat Bugzilla*[4]. Red Hat Bugzilla nabízí REST API pro komunikaci se systémem. Skrze tento přístupový bod bude systém schopen získat požadavky, jejichž ID bude předáno v konfiguračním souboru importovacího mechanismu.

Po získání všech potřebných informací o požadavku bude požadavek ze systému Red Hat Bugzilla převeden na odpovídající entity v datovém modelu eShoe. Importovat se musí ty entity a jejich vlastnosti, které jsou relevantní vzhledem k fulltextovému vyhledávání. Rovněž budou importovány další entity, pokud bezprostředně souvisejí s některou z importovaných entit a jejich nepřítomnost by měla za následek chybné chování systému, např. komponenty a verze projektu.

Importem se rozumí vytvoření odpovídající entity v datovém modelu eShoe, naplnění vybranými daty získanými ze systému Red Hat Bugzilla, případné propojení entit mezi sebou a uložení do databáze a fulltextového indexu. Uložení bude realizováno skrze servisní vrstvu, která již sama zajistí uložení do databáze a indexu.

4. `https://bugzilla.redhat.com/`

5 Elasticsearch-Annotations

Při návrhu (viz. kapitola 4.2.1) jsme se rozhodli naprogramovat nový mechanismus pro zajištění denormalizace entit a automatické úpravy indexu na základě anotací. Následující text podrobně probírá architekturu projektu Elasticsearch-Annotations.

5.1 Anotace

```
@Indexed(index = "people", type="person")
public class Person {

        @DocumentId
        private Long id;

        @Field(name = "lastName")
        @Analyzer(name = "personAnalyzer", tokenizer =
            "keyword")
        private String name;

        @IndexEmbedded
        private Address address;
}
```

Ukázka 5.1: Užití anotací Elasticsearch-Annotations

Základním stavebním kamenem projektu jsou anotace. Sada poskytovaných anotací je inspirována Hibernate Search a snaží se rovněž podobat i funkčností, aby byl případný přechod z Hibernate Search co možná nejjednodušší. Jejich dopad na výsledné chování je popsán v následující kapitole 5.3. Seznam nabízených anotací:

- `@Analyzer`
  Použitelná v kombinaci s anotací `@Field`. Umožňuje určit, které tokenizéry a filtry jsou využity při analýze tohoto atributu, tedy v podstatě definovat nový analyzér.

- `@ContainedIn`
  Použitelná na atributech entity neprimitivního datového typu. Označuje atribut jako vazbu na jinou entitu. Podrobněji vysvětleno v kapitole 5.3.

- `@Date`
  Použitelná v kombinaci s anotací `@Field`. Označuje datový typ `java.util.Date` a umožňuje stanovit, v jakém formátu je atribut uložen do indexu.

- `@DocumentId`
  Použitelná na atributy primitivního typu. Entita musí obsahovat právě jeden atribut označený touto anotací. Hodnota tohoto atributu udává ID záznamu v indexu Elasticsearch (viz. kapitola 3.3).

- `@Field`
  Použitelná na atributy primitivního datového typu a `java.util.Date`. Označuje atribut, jehož hodnota je výsledně skutečně zapsána do indexu. Anotaci můžeme specifikovat výsledný klíč v indexu parametrem `name`. V případě nepoužití parametru explicitně je klíč roven názvu atributu.

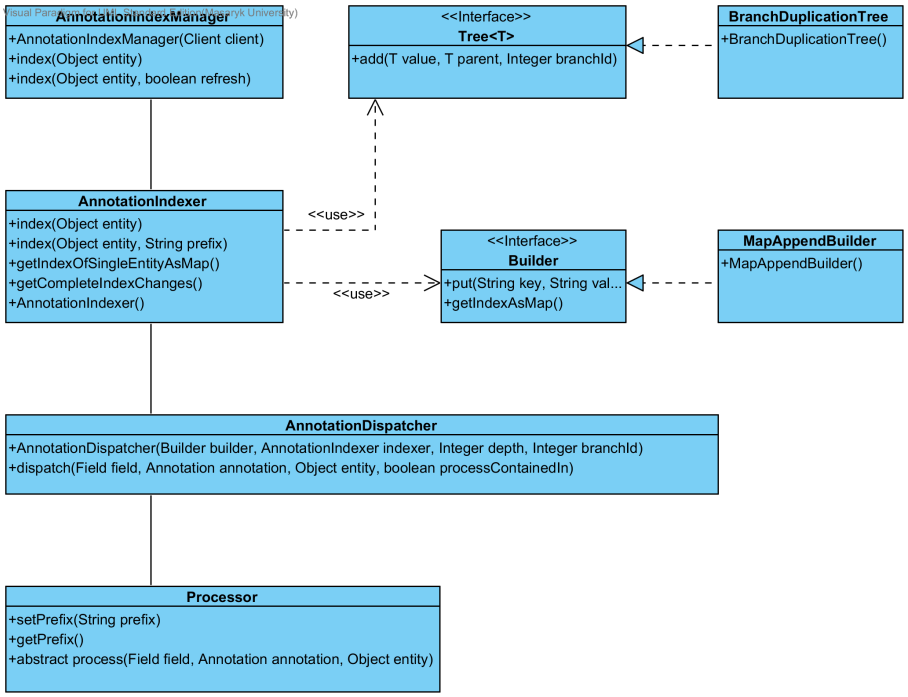
- `@Indexed`
  Použitelná na třídy. Signalizuje, že třída může být předána indexačnímu mechanismu Elasticsearch-Annotations. Přebírá dva parametry: `index` a `type`. Parametr `index` určuje jméno indexu, do kterého se výsledná entita uloží. Výchozí hodnotou je index „default“. `type` značí, do kterého typu v daném indexu. V případě nepoužití parametru je typ roven jménu třídy (viz. kapitola 3.3).

- `@IndexEmbedded`
  Použitelná pro atributy neprimitivního typu, především uživatelsky definované typy, kolekce (implementace rozhraní `java.util.Collection`, mapy (implementace rozhraní `java.util.Map`) a pole. Označuje relaci mezi entitami. Stejně jako anotace `@Field` umožňuje zadat parametr `name` se stejnou funkčností. Navíc může přebírat parametr `depth`, který omezuje počet úrovní indexace odkazované entity. Funkci této anotace podrobně rozebírá část 5.3.

Ukázka 5.1 demonstruje použití (s výjimkou `@ContainedIn` a `@Date`) všech anotací na jednoduché entitě.

5.2 Architektura indexační části

Text dále se věnuje architektuře indexace projektu Elasticsearch-Annotations. O tom, jak spolu jednotlivé třídy spolupracují a jak různá nastavení ovlivní výsledek, pojednává kapitola 5.3.



Obrázek 5.1: Architektura tříd řídících indexaci projektu Elasticsearch-Annotations

Kromě uvedených anotací (viz. 5.1) se dá indexační část projektu rozdělit do dvou částí: řídící třídy a procesory (implementace abstraktní třídy `Processor`). Diagram řídících tříd ukazuje obrázek 5.1.

Vstupním bodem indexace je třída `AnnotationIndexManager`. Při vytváření instance této třídy je nutné předat konstruktoru implementaci rozhraní `org.elasticsearch.client.Client`, která reprezentuje uživatelem nakonfigurovaného klienta, skrze kterého je komunikováno s Elasticsearch serverem. `AnnotationIndexManager` obsahuje metodu `index(Object entity)`, které je jako parametr předána anotacemi označená entita. Následně je vytvořena instance třídy `AnnotationIndexer`, která se postará o vlastní denormalizaci entity do formátu JSON dokumentu na základě přidružených anotací.

`AnnotationIndexer` postupně prochází atributy předané entity a u každého atributu zjišťuje, zda je označen některou z Elasticsearch-Annotations anotací. Za toto rozhodnutí je zodpovědná třída `AnnotationDispatcher`. `AnnotationDispatcher` projde všechny anotace u právě zpracovávaného atributu a v případě, že je přítomna jemu známá anotace, vybere podle ní příslušný procesor. Procesorem rozumíme implementaci abstraktní třídy

`Processor`, která odpovídá za „skutečné“ zpracování atributu a jeho hodnoty, případně předání řízení dál. Poznamenejme, že procesor je zvolen dle následující pravidel.

1. přítomna anotace `@Field` – zvolen `FieldAnnotationProcessor`
2. přítomna anotace `@Contained` – zvolen `ContainedInProcessor`
3. přítomna anotace `@IndexEmbedded`:
   (a) atribut je typu pole – zvolen `EmbeddedArrayProcessor`
   (b) atribut je typu implementujícího `java.util.Collection` – zvolen `EmbeddedCollectionProcessor`
   (c) typ atributu je implementací `java.util.Map` – zvolen `EmbeddedMapProcessor`
   (d) jinak zvolen `SimpleEmbeddedObjectProcessor`

Procesorům je skrze konstruktory předávána implementace rozhraní `Builder`. Toto rozhraní představuje datovou strukturu, která je použita pro průběžné budování indexu, v našem případě je jedinou implementací třída `MapAppendBuilder`. Funguje podobně jako mapa, ovšem při pokusu o uložení další hodnoty se stejným klíčem je hodnota přiřetězena k předchozí, nikoliv přepsána. Této vlastnosti je využito u indexace kolekcí, map a polí, aby všechny hodnoty byly k nalezení pod jedním klíčem v JSON dokumentu a daly se následně předat Elasticsearch serveru, který je mohl zaindexovat.

Po denormalizaci třídou `AnnotationIndexer`, `AnnotationIndexManager` pomocí metody `getCompleteIndexChanges()` získá seznam všech dokumentů v indexu, které je potřeba upravit a jejich nové hodnoty ve formátu JSON. Následně je vytvořen požadavek skrze předaného Elasticsearch klienta, jemuž je předán JSON dokument. Požadavek je poté odeslán na server, který se již postará o jeho zaindexování (viz. ukázka 5.2). Umístění dokumentu závisí na nastavených parametrech anotace `@Indexed` a hodnotě ID dokumentu označeného anotací `@DocumentId`.

```
IndexResponse response = client
        .prepareIndex(index, type, documentId)
        .setSource(jsonSource)
        .setRefresh(refresh)
        .execute()
        .actionGet();
```

Ukázka 5.2: Odeslání požadavku na vytvoření/úpravu indexu

5.3 Průběh indexace

Následující text se věnuje významu jednotlivých anotací a kdy kterou použít.

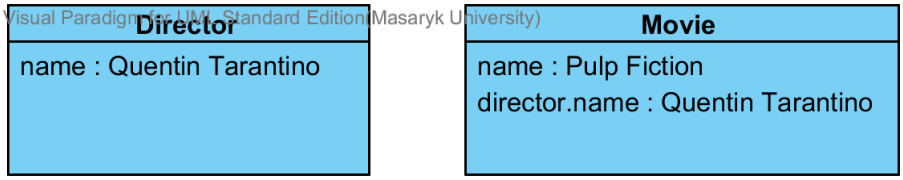
Použití anotace `@Field` je triviální. Do indexu je uložena přímo hodnota atributu, neboť je tato anotace použitelná na primitivní datové typy. Hodnota klíče je odvozena z předaného parametru `name`, případně ze jména atributu, jak bylo popsáno v kapitole 5.1. V případě současné přítomnosti anotace `@Analyzer` je Elasticsearch serveru sděleno, že daný atribut má zpracovat uvedeným tokenizérem a uvedenými filtry. Pakliže je přítomna anotace `@Date`, je hodnota data uložena v předepsaném formátu.

```
public class Director {
        @Field private String name;
        @ContainedIn private Movie movie;
}

public class Movie {
        @Field private String name;
        @IndexEmbedded private Director director;
}
...
director.setName("Quentin Tarantino");
movie.setName("Pulp Fiction");
movie.setDirector(director);

annotationIndexManager.index(director);
```

Ukázka 5.3: Užití anotací pro indexaci vztahů mezi entitami

Dále je potřeba vyřešit problém denormalizace vztahů, který je popsán v kapitole 3.2. K označení indexace atributů neprimitivního typu je využita anotace `@IndexEmbedded`. V případě, že se jedná o vnořený objekt, stačí příslušný atribut označit anotací `@IndexEmbedded`. Indexační mechanismus přidá do indexu rekurzivně všechny atributy označené anotací `@Field` vnořeného objektu tak, že před jméno klíče vnořeného atributu přidá klíč atributu z „rodičovské“ entity oddělený tečkou. V případě, že vnořená entita obsahuje další atribut označený anotací `@IndexEmbedded`, je pokračováno rekurzivně dále. Jde o stejné chování jako u Hibernate Search, viz. obrázek 3.3.

Druhým ze vztahů, když jednotlivé entity spolu sice souvisejí, ale mohou existovat i nezávisle, je řešen následovně. Elasticsearch-Annotations potřebuje vědět, ve kterých ostatních entitách je právě indexovaná entita zmíněna. K tomu slouží anotace `@ContainedIn`. Tato anotace říká, že v případě, že



Obrázek 5.2: Ilustrace výsledného indexu po provedení kódu v ukázce 5.3

je entita změněna, je nutné provést aktualizaci indexu všech entit, na které je ukázáno pomocí `@ContainedIn`. To nutí označit vztah na obou stranách, na straně vlastníka vztahu (toho, přes něhož můžeme následně vyhledávat s informacemi o související entitě) anotací `@IndexEmbedded` a na straně odkazované entity pomocí `@ContainedIn`. Ukázka 5.3 uvádí kompletní příklad, kvůli úspornosti je vynechán všechen nepodstatný kód.

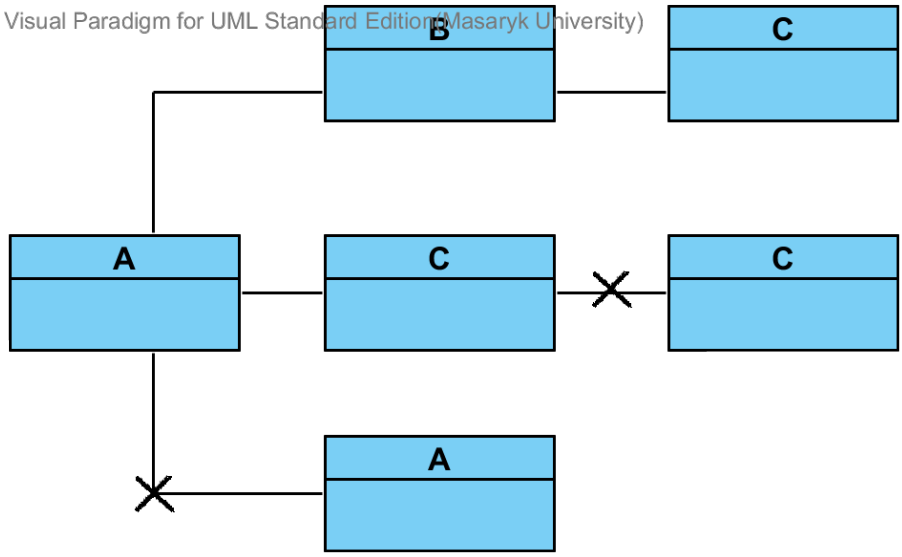
Obrázek 5.2 znázorňuje výsledný index. Za povšimnutí stojí, že správce indexu dostal za úkol zpracovat objekt `Director`, přesto je ve výsledku vytvořen i nový index entity `Movie` s aktuálními daty. Rovněž ukažme na nepřítomnost atributu `movie` v indexu `Director`.

Tento mechanismus je rovněž inspirován Hibernate Search a je s ním (téměř) totožný [2, str. 110].

Omezení hloubky indexace Při indexaci vnořených entit může dojít k tomu, že je indexováno zbytečně mnoho entit, protože jednotlivé entity mohou mít mnoho vztahů, či dokonce k zacyklení. Elasticsearch-Annotations tento problém řeší parametrem `depth` anotace `@IndexEmbedded`, který určuje hloubku zanoření, po kterou se má indexace provádět.

V případě, že parametr není specifikován je postupováno následovně. Indexační mechanismus postupuje s teoreticky neomezenou hloubkou, cestou však kontroluje, zda v dané větvi nezpracovával objekt stejné třídy (nikoliv stejnou instanci). Pokud narazí na třídu v dané větvi znovu, indexaci pro tuto větvi zastaví a dál již nepokračuje. K zaznamenávání grafu, jak indexace postupovala, využívá Elasticsearch-Annotations implementaci rozhraní `Tree` zvanou `BranchDuplicationDetectionTree`. Ta reprezentuje strom, do něhož je možné pouze prvky přidávat. V případě, že se stejný prvek vyskytuje na jedné větvi, vyhodí výjimku `IllegalStateException`. Indexační mechanismus tuto výjimku zachytí a indexaci dané větve ukončí. Obrázek 5.3 demonstruje výchozí chování s nespecifikovaným atributem `depth`.

Může se však stát, že entita má odkaz na entitu stejné třídy, např. u objektu znázorňujícího člověka mít seznam jeho přátel, kteří jsou rovněž

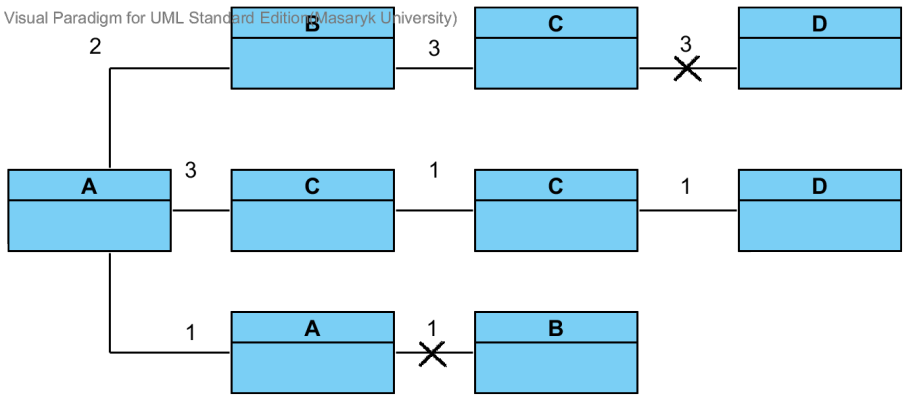


Obrázek 5.3: Vnořená indexace při výchozí hodnotě `depth`. Křížek znázorňuje asociaci, která již nebude indexována.

lidé. Pak by výchozí chování indexačního mechanismu nefungovalo, neboť by odmítlo indexovat objekt stejné třídy, přestože je to záměrně. Pokud chceme přesně kontrolovat, které entity budou indexovány do jaké hloubky, můžeme specifikovat parametr `depth` explicitně. Poté Elasticsearch-Annotations zcela ignoruje již projité entity a pouze počítá hloubku zanoření. Jakmile je dosažena uživatelem zvolená hloubka, indexace je zastavena. Nutno podotknout, že hloubka je počítáná od entity, na níž indexace započala. Na případné další uvedení parametru `depth` „po cestě“ není brán zřetel. Obrázek 5.4 ukazuje chování s explicitně uvedenou hloubkou.

5.4 Vyhledávací část

Pro vyhledávání nabízí Elasticsearch-Annotation pouze jednoduchou pomocnou funkcionalitu. Jediná třída vyhledávání části je `SearchManager`, která má metody `search` a `get`. Metoda `search` přebírá dva parametry – `SearchResponse` reprezentující vyhledávací dotaz pro Elasticsearch klienta a objekt typu `Class`. Metoda `search` pouze zpracuje nalezené výsledky dotazu tak, že z vráceného JSON objektu vytvoří objekt předané třídy a naplní atributy označené anotací `@Field` hodnotami z výsledku. Metoda `get` funguje analogicky, jako první parametr však vyžaduje objekt typu `GetResponse`,



Obrázek 5.4: Vnořená indexace při specifikované hodnotě `depth`. Křížek znázorňuje asociaci, která již nebude indexována.

což je požadavek na jeden konkrétní dokument.

5.5 Testy

Celý projekt Elasticsearch-Annotations je pokrytý jednotkovými a integračními testy. Jednotkové testy podrobně testují chování jednotlivých procesorů. Testováno je například chování na nenastavených atributech, dále správná detekce cyklické indexace či přejmenování klíče pole. Dále jsou k dispozici integrační testy, které testují správnou spolupráci všech komponent a že i při kombinaci více anotací systém vrací správné výsledky.

Poslední fází je integrační test spolupráce třídy `AnnotationIndexManager` s Elasticsearch serverem, kterého je docíleno použitím testovacího nástroje *elasticsearch-test*[1]. Tento nástroj umožňuje v prostředí JUnit testů snadno nastartovat Elasticsearch server pro testovací účely.

1. `https://github.com/tlrx/elasticsearch-test`

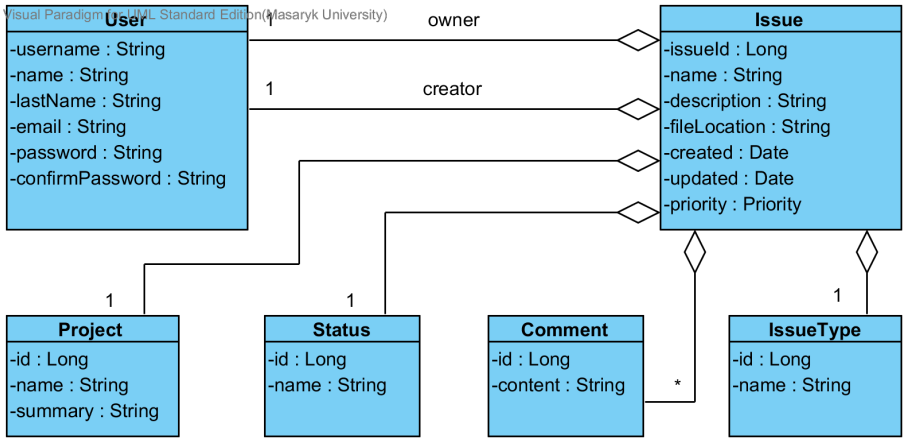
6 Implementace

Kapitola 4 představila specifikaci požadavků na vyhledávání v systému eShoe a navrhla způsob, jakým budou řešeny. Následující text popisuje implementaci uvedeného návrhu.

6.1 Indexace

K realizaci indexace je použit projekt Elasticsearch-Annotations podrobně představený v kapitole 5. Pro zajištění zavedení relevantních dat do indexu Elasticsearch je zapotřebí dvou věcí: vybrat atributy entit, které mají být indexovány a označit je vhodnými anotacemi a navázat události vytváření a úpravy indexu na změny entit v databázi.

V kapitole 4.1 jsou uvedeny dotazy, které systém musí podporovat. Podle nich jsem vybral atributy entit tak, aby všechny dotazy bylo možné provést. Obrázek 6.1 ukazuje relevantní část datového modelu eShoe, ze kterého jsou atributy vybrány.



Obrázek 6.1: Relevantní část datového modelu eShoe

Pro splnění všech typů vyhledávacích dotazů byly k indexaci vybrány následující atributy:

- `Issue.id`, `Issue.name`, `Issue.summary`, `Issue.description`, `Issue.priority`, `Issue.created` a `Issue.updated`
- `IssueType.name`

- `Status.name`
- `User.username`
- `Project.name`
- `Comment.content`

Kromě zřejmé nutnosti indexace atributů, podle kterých se filtruje na přímou shodu (např. jméno projektu, typ požadavku atd.) jsou do indexu zahrnuty položky, ve kterých by se mohly nacházet relevantní informace vzhledem k zadané frázi pro fulltextové vyhledávání. Jedná se o atributy jména, shrnutí a popisu požadavku, rovněž jsou do indexu zahrnuty všechny komentáře k požadavku. Indexace těchto položek by měla pomoci k očekávanějším výsledkům, neboť pokud se zadaná fráze vyskytuje pouze ve jméně požadavku, měl by být požadavek ve výsledku logicky dále, než ten, který obsahuje frázi ve jméně, popisu a ještě několika komentářích.

Uložení hodnoty některých atributů je modifikováno použitím jiných tokenizérů a filtrů. Atributy, které se používají pro filtraci na přímou shodu, není správné rozdělovat pomocí mezer, neboť se v podstatě jedná o klíčová slova, přes která probíhá filtrace. Následně není žádoucí, aby uživatel muset tyto hodnoty zadávat s ohledem na velikost znaků. Proto je těmto atributům (např. jménu projektu, typu požadavku atd.) nastaven tokenizér `KeywordTokenizer` a filtr `LowerCaseTokenFilter`. Na ostatní atributy je použit ve výchozím nastavení `StandardAnalyzer`, viz. 3.1.3.

Pro navázání indexace na změny v databázi byla vybrána servisní vrstva. Ta však z předchozího vývoje systému chybí, je proto nově vytvořena v balíku `com.issuetracker.service`. Posledním zajímavavým bodem je získání správce indexu (instance `AnnotationIndexManager` s nakonfigurovaným klientem pro přístup k serveru Elasticsearch) v jednotlivých servisních třídách. Toho je dosaženo pomocí *CDI* (*Contexts and Dependency Injection*). Získat nakonfigurovanou instanci správce indexu lze pomocí anotace `@Inject`. Ukázka 6.1 uvádí třídu `AnnotationIndexerProducer` a její metodu `getIndexManager`, která produkuje správce indexu, a tím jej umožňuje získat snadno kdekoliv v aplikaci.

```
@Produces
public AnnotationIndexManager getIndexManager() {
    if(manager == null) {
        manager = new AnnotationIndexManager(client);
    }
    return manager;
}
```

Ukázka 6.1: Použití CDI pro získávání správce indexu

6.2 Vyhledávání

V předchozí kapitole je popsána implementace indexace entit. Následující text předpokládá, že data jsou úspěšně indexována serverem Elasticsearch a řeší, jakým způsobem jsou dotazována. Jak je uvedeno ve specifikaci (viz. kapitola 4.1), je dotazování realizováno tvorbou dotazů ve vlastním dotazovacím jazyce.

Byl vytvořen vlastní dotazovací jazyk pro účely eShoe. Ukázka 6.2 definuje jeho gramatiku.

```
dotaz -> konjunkce
konjunkce -> vyraz | vyraz AND konjunkce
vyraz -> rovnost | mnozina | fulltext | mensi |
   mensi_rovno | vetsi | vetsi_rovno
rovnost -> <jmeno_atributu> = "<hodnota_atributu>"
mnozina -> <jmeno_atributu> =
        (<hodnoty_v_uvozovkach_oddeleny_carkami>)
fulltext -> text = "<fulltextova_fraze>"
mensi -> <jmeno_atributu> < "<hodnota_atributu>"
mensi_rovno -> <jmeno_atributu> <= "<hodnota_atributu
   >"
vetsi -> <jmeno_atributu> > "<hodnota_atributu>"
vetsi_rovno -> <jmeno_atributu> >=  "<hodnota_atributu
   >"
```

Ukázka 6.2: Gramatika dotazovacího jazyka eShoe

Podotkněme, že operátor ~(vlnka) je použitelný pouze s atributem „text“ a dohromady označují frázi pro fulltextové vyhledávání.

Ke kontrole správnosti syntaxe a následnému parsování dotazu je použita

knihovna *ANTLR*[1]. ANTLR (*ANother Tool for Language Recognition*) je generátor parseru pro čtení, zpracování, spouštění či překládání strukturovaného textu nebo binárních souborů. Je široce používaný pro vytváření jazyků, nástrojů a rámců. ANTLR z gramatiky vygeneruje parser, který lze následně použít pro prochází stromem vybudovaným ze zpracovaného textu. [1]

Ukázky výsledného dotazovacího jazyka jsou uvedeny v kapitole 4.1 společně se specifikací požadavků.

```
query: andExpression;
andExpression: expression (AND! expression)*;
expression: equals | in | tilda | lt | gt | lte | gte;
equals: FIELD_NAME '='^ fieldValue;
in: FIELD_NAME IN^
    '('! fieldValue (','! fieldValue)*') '!;
tilda: FIELD_NAME '~'^ fieldValue;
lt: FIELD_NAME LT^ fieldValue;
gt: FIELD_NAME GT^ fieldValue;
gte: FIELD_NAME GTE^ fieldValue;
lte: FIELD_NAME LTE^ fieldValue;
fieldValue: FIELD_VALUE;
```

Obrázek 6.2: Gramatika dotazovacího jazyka z obrázku 6.2 zapsána pomocí ANTLR

S využitím ANTLR je ve třídě `SearchServiceBean` z textového dotazu vytvořen odpovídající dotaz pro klienta Elasticsearch (objekt `SearchResponse`), který je předán třídě `SearchManager` z projektu Elasticsearch-Annotations. Manažer pak vrací seznam nalezených požadavků.

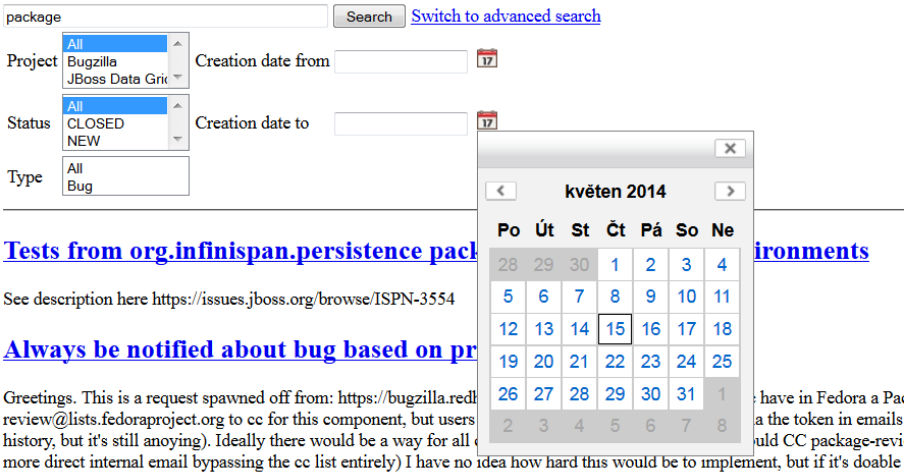
Zbývá vyjmenovat, jaká jména atributů lze v dotaze použít a uvést jejich korespondující smysl.

- `id` – ID požadavku
- `project` – jméno projektu
- `status` – status požadavku
- `issue_type` – typ požadavku
- `created` – datum vytvoření požadavku

1. `http://www.antlr.org/`

- `updated` – datum poslední modifikace požadavku
- `owner` – jméno uživatele, kterému je požadavek přiřazen
- `creator` – jméno uživatele, který požadavek vytvořil
- `priority` – priorita požadavku

Pomocí vytvořeného dotazovacího jazyka lze realizovat všechny dotazy uvedené ve specifikaci. Rovněž lze dotazovací jazyk v budoucnu snadno rozšířit o další atributy, na kterých půjde vyhledávat.



Obrázek 6.3: Prototyp grafického rozhraní pro fulltextové vyhledávání

6.3 Uživatelské rozhraní

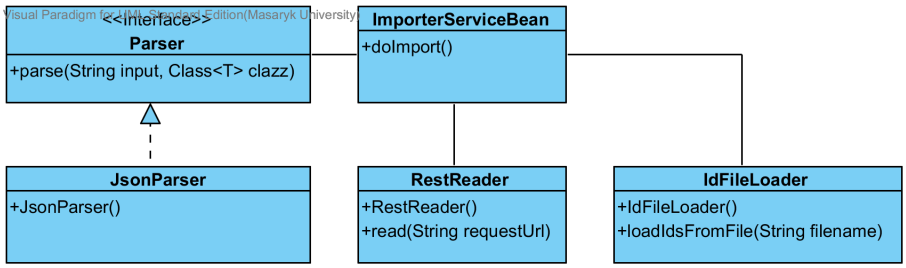
Součástí implementace je velice jednoduché grafické uživatelské rozhraní, jak je uvedeno ve specifikaci (viz. 4.1). Stejně jako GUI zbytku systému je uživatelské rozhraní pro vyhledávání napsáno s pomocí technologie *Apache Wicket*. GUI slouží v podstatě jen jako tvůrce vyhledávacích dotazů v dotazovacím jazyce popsaném v kapitole 6.2. Rovněž umožňuje dotaz přímo zadat, neboť z důvodu jednoduchosti není možné „naklikat“ dotaz z GUI se všemi možnými omezeními.

Skrze uživatelské rozhraní je možné zadat pouze frázi pro fulltextové vyhledávání, filtrovat podle jména projektu, statusu, typu požadavku a data

vytvoření. Pro pohodlné zadávání data je využita komponenta Apache Wicket `DateTextField`, která zobrazí miniaturní kalendář, ve kterém se dá datum zvolit bez nutnosti psát jej na klávesnici. Obrázek 6.3 ilustruje tento prototyp grafického rozhraní pro fulltextové vyhledávání v systému eShoe.

6.4 Import dat

Poslední věcí, kterou je třeba implementovat, je import dat ze systému *Red Hat Bugzilla* (viz. kapitola 4.1). K realizaci importovacího mechanismu (dále jen importeru) je potřeba získat relevantní data ze systému skrze REST rozhraní, které Red Hat Bugzilla nabízí. Následně získaná data převést na strukturu datového modelu eShoe. Všechen kód týkající se importování se nachází v balíku `com.issuetracker.importer` a třídě `com.issutracker.service.ImporterServiceBean`. Započít import dat může uživatel z hlavního menu aplikace kliknutím na položku „Import issues“.



Obrázek 6.4: Diagram tříd importeru

Řídícím prvkem je třída `ImporterServiceBean`. Pomocí `IdFileLoader` načte seznam požadavků, které chce uživatel do systému importovat. Tento seznam je uložen v souboru `resources/importer-bugzilla-ids.txt`. Jedná se o textový soubor, který obsahuje ID požadavků oddělená novým řádkem. Uživatel si tak může přesně zvolit, které požadavky do systému importovat.

Následně jsou požadavky se zvolenými ID načteny přes REST API vzdáleného systému včetně jejich komentářů. Za zaslání HTTP požadavku a získání odpovědi je odpovědná třída `RestReader`. Red Hat Bugzilla nabízí několik přístupových bodů pro REST API [3], z nichž jsem vybral ten, který vrací data ve formátu JSON.

Po získání JSON odpovědi je nutné data namapovat na entity eShoe. Pro přehlednější práci jsem se rozhodl převést JSON do pomocných objektů, se kterými se následně pracuje. K tomu je využita třída `JsonParser`.

Třída interně využívá knihovnu *Jackson*[2] pro namapování JSON dokumentů na Java objekty. Jako pomocné třídy, na které se mapuje JSON odpověď, jsou vytvořeny `BugzillaBug`, `BugzillaBugResponse`, `BugzillaComment` a `BugzillaCommentReponse` tak, aby přesně odpovídaly struktuře vráceného JSON dokumentu, a tím mohly být úspěšně vytvořeny.

Jakmile jsou pomocné objekty vytvořeny, třída `ImportServiceBean` vytvoří entity z datového modelu eShoe a pomocí *set* metod je naplní odpovídajícími daty. Následně je předá servisní vrstvě, která je uloží do databáze i fulltextového indexu. Následuje seznam hlavních entit a jejich vlastností, které jsou importovány. Jedná se především o vlastnosti bezprostředně související s vyhledáváním. Kompletní seznam je uveden v příloze B.

- `Issue`
  - ⋆ jméno
  - ⋆ priorita
  - ⋆ vazba na uživatele, který požadavek vytvořil
  - ⋆ vazba na uživatele, kterému je požadavek přiřazen
  - ⋆ vazba na typ požadavku
  - ⋆ vazba na projekt
  - ⋆ vazba na status
  - ⋆ vazba na komentáře K požadavku
  - ⋆ datum vytvoření požadavku
  - ⋆ datum poslední modifikace požadavku
- `Project` – jméno projektu
- `IssueType` – název typu požadavku
- `Status` – název statusu
- `User` – jméno a příjmení uživatele
- `Comment` – obsah komentáře

2. `https://github.com/FasterXML/jackson`

6.5 Testy

Součástí zadání práce a specifikace (viz. 4.1) je naprogramování automatizovaných testů, které ověří správnost řešení. Správná funkčnost indexační části je pokryta testy přítomnými v projektu Elasticsearch-Annotations. Zbývá ověřit správnou funkčnost vyhledávání v systému eShoe.

Pro implementaci testů pro vyhledávání je znovu použita knihovna `elasticsearch-test` (viz. 5.5). Otestování správné funkčnosti vyhledávání spočívá především v ujištění, že systém je schopen odpovědět na všechny typy dotazů, které byly definovány v kapitole 4.1. Všechny testy jsou přítomny ve třídě `FulltextSearchTest`. Pro tento účel byly vytvořeny 4 testovací entity. Hodnoty jejich atributů i podrobné rozepsání testovacích případů společně s očekávanými výsledky jsou uvedeny v příloze C.

Pro každý typ dotazu je přítomna zvláštní testovací metoda, která ověřuje správnou funkčnost pro daný typ dotazu. Rovněž je testována možnost kombinace více podmínek pomocí operátoru `AND` či možnost shody hodnoty atributu s jednou nabízených hodnot pomocí operátoru `IN`. Zvláštním případem je pak otestování vrácení relevantnějšího výsledku dříve než ostatních (obsahuje více klíčových slov).

7 Závěr

7.1 Zhodnocení výsledků práce

Cílem práce bylo implementovat fulltextové vyhledávání v systému pro správu požadavků eShoe a poskytnout tak uživateli možnost vyhledávat požadavky podle zadaných kritérií. Jednotlivé části cíle uvedené v sekci 1.1 byly splněny následovně:

- Pochopení principů fulltextového vyhledávání a seznámení se s dostupnými technologiemi na platformě Java: V rámci práce byly vysvětleny principy, na kterých fulltextové vyhledání funguje a uvedeny příklady problémů, které dokáže fulltextové vyhledávání vyřešit oproti vyhledávání pomocí SQL. Dále byly uvedeny tři konkrétní technologie pro implementaci fulltextového vyhledávání na platformě Java a popsány jejich charakteristiky a možnosti.

- Zvládnutí indexace entit v systému eShoe: V rámci práce byl navržen a implementován mechanismus pro indexaci entit, na jejichž základě uživatel hledá odpovídající výsledky. Práce uvádí seznam dotazů, na které je systém schopen odpovědět a na jejich základě popisuje, které entity byly k indexaci vybrány.

- Vytvoření prostředku pro zadávání dotazů pro vyhledávání: Pro systém eShoe byl navržen dotazovací jazyk umožňující pokrýt všechny typy dotazů, které byly pro systém specifikovány.

- Import dat: Pro demonstraci vyhledávání byl naprogramován mechanismus pro import požadavků z existujícího systému pro správu chyb Red Hat Bugzilla. Importovaná data jsou vybrána uživatelem na základě ID požadavku, uživatel tak může přesně zvolit, které požadavky budou do systému importovány. Počet požadavků, které lze do systému importovat, není teoreticky omezen.

- Otestování funkčnosti vyhledávání: Součástí implementace jsou i testy pokrývající veškerou funkčnost vyhledávání. Otestovány jsou všechny části implementace, indexační mechanismus i vyhledávání.

Za přidanou hodnotu práce lze kromě implementace samotného vyhledávání v systému eShoe považovat i vznik nástroje Elasticsearch-Annotations, který jako první poskytuje indexační možnosti pro Elasticsearch na bázi anotací.

7.2 Možnosti pokračování

Možnosti rozvíjení implementace dál lze hodnotit ze dvou pohledů: vylepšení vyhledávání v samotném systému a další směr projektu Elasticsearch-Annotation. Pro vyhledávání v systému se nabízí:

- Zdokonalit uživatelské rozhraní pro vyhledávání. Kromě vytvoření graficky přívětivějšího vzhledu se jedná např. o možnost našeptávání hledané fráze, poskytnutí statistik o nalezených požadavcích, filtrování na základě dalších vlastností entit.
- Vystavit funkčnost fulltextového vyhledávání skrze REST API.
- Umožnit pokročilejší funkce vyhledávání jako opravu překlepů, vyhledávání podle synonym apod.

Z hlediska pokračování projektu Elasticsearch-Annotations je ambicí začlenit jej přímo do projektu Elasticsearch, tedy otevřít komunitě. Pro splnění tohoto cíle bude zapotřebí doprogramovat velké množství funkcí, např. možnost použití anotace `@FieldBridge` jako v Hibernate Search, umožnit větší množství nastavení mapování skrze anotace atd.

Literatura

[1] *About The ANTLR Parser Generator* [online]. 2014 [cit. 2014-05-15]. Dostupné z: `http://www.antlr.org/about.html`

[2] BERNARD, Emmanuel a John GRIFFIN. *Hibernate search in action*. Greenwich, CT: Manning, c2009, xxiv, 463 p. ISBN 19-339-8864-9.

[3] Bugzilla::WebService::Server::REST. *Bugzilla 4.5.4+ API Documentation* [online]. 2013 [cit. 2014-05-15]. Dostupné z: `http://www.bugzilla.org/docs/tip/en/html/api/Bugzilla/WebService/Server/REST.html`

[4] Elasticsearch. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-04]. Dostupné z: `http://en.wikipedia.org/wiki/Elasticsearch`

[5] *Elasticsearch: The Definitive Guide* [online]. 2014 [cit. 2014-05-04]. Dostupné z: `http://www.elasticsearch.org/guide/en/elasticsearch/guide/current/`

[6] GOTTVALDOVÁ, Monika. *Modern open source Java EE-based process and issue tracker*. Brno, 2014. Diplomová práce. FI MU.

[7] HORTON, Ivor R. *Ivor horton's beginning java, java 7 edition*. 1st ed. Indianapolis, IN: Wiley Pubishing, Inc., 2011, p. cm. ISBN 04-704-0414-0.

[8] KOFLER, Michael. *Mistrovství v MySQL 5*. Vyd. 1. Překlad Jan Svoboda, Ondřej Baše, Jaroslav Černý. Brno: Computer Press, 2007, 805 s. ISBN 978-80-251-1502-2.

[9] MCCANDLESS, Michael, Erik HATCHER, Otis GOSPODNETIC a Otis GOSPODNETIC. *Lucene in action*. 2nd ed. Greenwich: Manning, c2010, xxxviii, 488 p. ISBN 19-339-8817-7.

[10] PANDA, Debu, Reza RAHMAN a Derek LANE. *EJB 3 in action*. Greenwich, CT: Manning Publications Co., c2007, xxix, 677 p. ISBN 19-339-8834-7.

A Obsah přiloženého archívu

Obsah archívu přiloženého k práci:

- adresář *zdrojove_kody*, který obsahuje:
  - ⋆ zdrojový kód systému eShoe
  - ⋆ zdrojový kód projektu Elasticsearch-Annotations
  - ⋆ adresář *resteasy-jboss-modules-3.0.6.Final*
  - ⋆ adresář *mysql-connector*
  - ⋆ textový soubor *README.txt* s instrukcemi pro spuštění eShoe i importovacího mechanismu
- adresář *text_prace*, který obsahuje:
  - ⋆ zdrojový kód této práce ve formátu *.tex*
  - ⋆ text práce ve formátu *.pdf*
  - ⋆ obrázky a diagramy použité v této práci

B Seznam importovaných entit

V rámci importu dat ze systému Red Hat Bugzilla jsou do systému eShoe zavedeny tyto entity a jejich atributy:

- `Issue`
  - jméno
  - priorita
  - vazba na uživatele, který požadavek vytvořil
  - vazba na uživatele, kterému je požadavek přiřazen
  - vazba na typ požadavku
  - vazba na projekt
  - vazba na komponentu projektu
  - vazba na verzi projektu
  - vazba na status
  - vazba na komentáře k požadavku
  - datum vytvoření požadavku
  - datum poslední modifikace požadavku
- `Project`
  - jméno
  - vazba na verze projektu
  - vazba na komponenty projektu
- `ProjectVersion`
  - jméno
- `Component`
  - jméno
- `IssueType`
  - jméno
- `Status`

    - jméno
- `User`
    - jméno
- `Comment`
    - obsah komentáře

C Testy pro vyhledávací část

Tato příloha uvádí 4 entity, které jsou použity pro testování vyhledávání v systému eShoe a hodnoty jejich atributů. Dále uvádí testovací případy i s očekávanými výsledky.

ID	1	2
souhrn	Failing unit tests	Graceful shutdown should be supported
typ	Bug	Feature request
priorita	HIGH	LOW
projekt	Infinispan	Infinispan
status	Open	Resolved
vlastník	Emmanuel Bernard	Martin Gencur
tvůrce	Jiri Holusa	Tomas Sykora
datum vytvoření	1. 1. 2014	1. 3. 2014
datum poslední úpravy	2. 1. 2014	4. 3. 2014

Tabulka C.1: Hodnoty testovacích entit s ID 1 a 2

ID	3	4
souhrn	notifyMessage – stack attribute – only default value works	EAP 6 JAAS cache does not work correctly
typ	Bug	Enhancement
priorita	HIGH	LOW
projekt	RichFaces	EAP
status	Open	Coding in progress
vlastník	Juraj Huska	Pavol Pitonak
tvůrce	Juraj Huska	Brian Leathem
datum vytvoření	7. 7. 2014	7. 7. 2014
datum poslední úpravy	7. 7. 2014	7. 7. 2014

Tabulka C.2: Hodnoty testovacích entit s ID 3 a 4

Pro realizaci testování jsou použity dotazy ve vytvořeném dotazovacím jazyce (viz. kapitola 6.2).

Test 1. Použití filtru na projekt. Dotaz: $project = "Infinispan"$
Očekávaný výsledek: Entity s ID 1 a 2.

Test 2. Použití filtru na status. Dotaz: $status = "Open"$
Očekávaný výsledek: Entity s ID 1 a 3.

Test 3. Použití filtru na typ požadavku. Dotaz: $issue\_type = "Bug"$
Očekávaný výsledek: Entity s ID 1 a 3.

Test 4. Použití filtru na datum vytvoření požadavku. Rovněž test operátorů $>=$, $<=$ a AND. Dotaz: $created >= "2014-01-01" AND$
$created <= "2014-03-01"$
Očekávaný výsledek: Entity s ID 1 a 2.

Test 5. Použití filtru na datum poslední modifikace požadavku. Rovněž test operátorů $>=$, $<=$ a AND. Dotaz: $updated >= "2014-03-04" AND$
$updated <= "2014-04-05"$
Očekávaný výsledek: Entita s ID 2.

Test 6. Použití filtru na jméno tvůrce. Dotaz: $creator = "Tomas\ Sykora"$
Očekávaný výsledek: Entita s ID 2.

Test 7. Použití filtru na jméno vlastníka. Dotaz: $owner = "Pavol\ Pitonak"$
Očekávaný výsledek: Entita s ID 4.

Test 8. Použití filtru na ID požadavku. Dotaz: $id = "1"$
Očekávaný výsledek: Entita s ID 1.

Test 9. Použití operátoru IN. Dotaz: $id\ IN\ ("1", "4")$
Očekávaný výsledek: Entity s ID 1 a 4.

Test 10. Použití operátoru $\sim$ a vrácení relevantnějších výsledků na vyšších

pozicích. Dotaz: $text \sim$ "$unit\ tests\ infinispan$"
Očekávaný výsledek: Na prvním místě vrácena entita s ID 1, protože obsahuje klíčová slova jak v souhrnu, tak ve jméně projektu. Druhá vrácena entita s ID 2, protože obsahuje klíčové slovo pouze ve jméně projektu. Ostatní entity vráceny až poté, jelikož neobsahují žádné klíčové slovo.