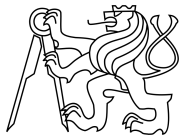
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Aplikace pro získávání dat z logů
Student:	Michal Stejskal
Vedoucí:	Ing. Josef Vogel, CSc.
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Pomocí metod softwarového inženýrství navrhněte a implementujte rest api a webovou aplikaci pro vyhledávání a agregaci dat z logů a podobných datových vstupů podle požadavků zadavatele - společnosti NEWPS s. r. o.

1. Seznamte se s uživatelskými požadavky na aplikaci pro získávání dat z logů.

2. Na základě analýzy vyberte technologie implementace.

3. Pomocí metod softwarového inženýrství analyzujte a navrhněte aplikaci.

4. Na základě předchozích kroků implementujte rest api pro vyhledávání a agregaci vstupních dat, které bude na základě uživatelských pravidel kontrolovat vstupní data a volat akce.

5. Pro rest api vytvořte webového klienta.

6. Rest api a webového klienta otestujte a nasaďte v infrastruktuře zadavatele.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 26. října 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Aplikace pro získávání dat z logů

Michal Stejskal

Vedoucí práce: Ing. Josef Vogel, CSc.

11. května 2017

Poděkování

Tímto bych rád upřímně poděkoval svému vedoucímu, panu Ing. Josefu Vogelovi, CSc. za rady, konzultace, který mi věnoval při psaní této bakalářské práce. Dále bych rád poděkoval všem zaměstnancům společnosti NEWPS.CZ, kteří konzultovali případné problémy. Děkuji také své rodině za podporu během celých mých studií.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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 11. května 2017 .....

Odkaz na tuto práci

Stejskal, Michal. *Aplikace pro získávání dat z logů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato bakalářská práce se zabývá návrhem a vývojem systému pro prohledávání a agregaci log souborů a generováním událostí na základě nalezených dat.

V první části je popsána architektura tohoto systému, příkladové případy užití, požadavky na tento systém, popis uživatelů systému a zvolené technologie.

Na základě výsledků první části, je ve druhé části popsán proces implementace jednotlivých komponent systému, rozhraní jejich komunikace a popis vyhledávání a zpracování výsledků indexovacího nástroje. Vyvinutými komponentami jsou RESTful Java API, které zpracovává požadavky klientské aplikace, vytváří vyhledávací objekty na indexovací nástroj a generuje události na základě nalezených dat. Dále byla vyvinuta klientská aplikace sloužící k interakci s uživatelem a zpracování výsledků vrácených systémovým RESTful API. Systémové RESTful API bylo implementováno v jazyce Java, klientská aplikace v AngularJS 2. Jako indexovací nástroj byl zvolen Elasticsearch.

Celý systém byl otestován a popis průběhu testování a zvolené testovací metody, jak automatického, tak manuálního testování, jsou popsány ve třetí kapitole.

Proces nasazení systému do produkčního či testovacího prostředí je popsán ve čtvrté kapitole. V této kapitole je popsána kontejnerizace jednotlivých komponent systému nástrojem Docker a propojení jejich komunikace.

V závěru je diskutována budoucnost systému a další možná rozšíření.

Klíčová slova Elasticsearch, prohledávání log souborů, agregace log souborů, generování událostí, REST, AngularJS 2

Abstract

In this bachelor thesis I am solving the design and development of a system for searching and aggregation of log files and generating events based on the data found.

In the first part is described the architecture of this system, examples of use cases, requirements for this system, description of the system users and the selected technologies.

Based on the results of the first part, the second part describes the process of implementation of the individual components of the system, the interface of their communication and a description of the search and processing of the results of the indexing tool. Developed components are the RESTful Java API that handles client application requests, creates search objects for an indexing tool, and generates events based on the data found. In addition, a client application was developed to interact with the user and process the results returned by the system's RESTful API. System RESTful API was implemented in Java, the client application in AngularJS 2. Elasticsearch was chosen as the indexing tool.

Proces of the deployment of the system to the production or testing environment is described in Chapter 4. In this chapter is described how to containerize each component of the system with the Docker and link their communications.

In the end, the future of the system and other possible extensions are discussed.

Keywords Elasticsearch, search in log files, aggregation in log file, generating events, REST, AngularJS 2

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Motivace práce

Každá moderní aplikace generuje při běhu určité informace o prostředí, ve kterém běží, o uživatelích a o operacích, které vykonává. Ve většině případů jsou tyto informace uloženy a přistupuje se k nim až ve chvíli, kdy se v aplikaci vyskytne určitý problém a je potřeba zjistit jeho příčinu. Cílem této práce je návrh a implementace systému, který bude aktivně prohledávat soubory obsahující textové nebo jiné zprávy, a na základě uživatelem předdefinovaných pravidel generovat události. Tyto události lze využít mnoha způsoby. Základní možností využití sledování log souborů je predikce bezpečnostních událostí, možného přetížení systému, či jiných chyb systému. Další možností využití tohoto sledovacího systému je nastavení pružného reagování marketingových kampaní na momentální poptávku. Teoreticky takovýto systém dokáže informovat o jakékoliv události, která ve sledovaných datech nastane, lze-li pro takovou událost sestrojit sledovací pravidlo.

Cíle práce

Vzniklý systém bude nabízet agregaci vstupních dat a zobrazení výsledků těchto agregací v různých formách na základě jejich typu. Systém bude nabízet spojení těchto agregací do vyhledávacích termů a vytvářet sledovací akce. Systém bude dále nabízet možnost vyhledávání ve vstupních datech, ať standardní vyhledávání podle hodnoty určitého pole, či fulltextové vyhledávání a nalezené hodnoty zobrazí uživateli v lidsky čitelné formě. Pro dosažení těchto cílů je potřeba navrhnout a implementovat či vybrat následující komponenty:

- nástroj pro sběr, transformaci a přenos vstupní dat,
- indexovací nástroj pro zpracování dat,

- RESTful API pro zpracování požadavků klientské komponenty a obsluhu indexovacího nástroje,
- klientská aplikace pro interakci s uživatelem.

Důležitou součástí této práce je testování funkcionalit systému a jejich následné nasazení do testovacího, či produkčního prostředí.

Struktura práce

Tato bakalářská práce je rozdělena do čtyř různých kapitol. Každá kapitola reprezentuje jednu část vývoje systému a jsou v ní popsány využité techniky a postupy.

Kapitola Analýza a návrh popisuje požadavky na systém, které vznikly jako výsledek studia agregačních a vyhledávacích systémů. Dále se zabývá architekturou systému, vnitřní komunikací a využitými technologiemi.

Kapitola Realizace se zabývá popisem použitých technik při implementaci systému. Jsou zde uvedeny základní ukázky kódu jednotlivých komponent systému a jejich administrace.

Kapitola Testování shrnuje testovací techniky jednotlivých komponent systému a ukazuje postupy automatického a manuálního testování.

Kapitola Nasazení popisuje postup nasazení systému v produkčním či testovacím prostředí. Dále vysvětluje proces kontejnerizace jednotlivých komponent systému a následnou spolupráci jednotlivých kontejnerů.

Kapitola 1

Analýza a návrh

1.1 Analýza požadavků

V této kapitole jsou popsané jednotlivé požadavky na systém, které vznikly jako výsledek zkoumání problému zpracování vstupních dat v reálném čase. Na základě těchto požadavků byla navržena architektura systému a funkčnosti jednotlivých komponent. Pro větší přehlednost jsou požadavky rozděleny do částí reprezentující jednotlivé komponenty systému. Těmito komponentami jsou nástroj pro sběr, transformaci a přenos vstupní dat, indexovací nástroj pro zpracování dat, RESTful API pro zpracování požadavků klientské komponenty a obsluhu indexovacího nástroje a klientská aplikace pro interakci s uživatelem.

1.1.1 Nástroj pro sběr, transformaci a přesun dat

Funkční požadavky

- Nástroj umožní transformaci dat.
- Nástroj umožní přenos dat do indexovacího nástroje.

Nefunkční požadavky

- Nástroj bude distribuován pod otevřenou licencí.
- Nástroj umožní pracovat s daty v téměř reálném čase.
- Nástroj poběží samostatně.

1.1.2 Indexovací nástroj

Funkční požadavky

- Nástroj umožní automatické vkládání dat.

- Nástroj umožní automatickou analýzu dat.
- Nástroj umožní vyhledávat v datech.
- Nástroj umožní agregovat data.

Nefunkční požadavky

- Nástroj bude distribuován pod otevřenou licencí.
- Nástroj umožní pracovat s daty v téměř reálném čase.
- Nástroj půjde jednoduše škálovat.
- Nástroj poběží samostatně.

1.1.3 RESTful API

Funkční požadavky

- Rozhraní umožní vytvářet nová agregační pravidla.
- Rozhraní umožní editovat stávající agregační pravidla.
- Rozhraní umožní vyhledávat podle agregačních pravidel.
- Rozhraní umožní mazat agregační pravidla.
- Rozhraní umožní získání seznamu agregačních pravidel.
- Rozhraní umožní získání geolokačních informací podle agregačních pravidel k tomu určených.
- Rozhraní umožní vytvářet nová vyhledávací pravidla.
- Rozhraní umožní editovat stávající vyhledávací pravidla.
- Rozhraní umožní vyhledávat podle vyhledávacích pravidel.
- Rozhraní umožní mazat vyhledávací pravidla.
- Rozhraní umožní získání seznamu vyhledávacích pravidel.
- Rozhraní umožní vytvářet nové akce s vyhledávacími a agregačními objekty.
- Rozhraní umožní editovat stávající akce s vyhledávacími a agregačními objekty.
- Rozhraní umožní mazat akce s vyhledávacími a agregačními objekty.

- Rozhraní umožní vyhledávat podle pravidel v akcích.
- Rozhraní umožní získání seznamu stávajících akcí.
- Rozhraní umožní automatické spuštění akcí.
- Rozhraní umožní generování nových událostí na základě výsledků akcí.
- Rozhraní umožní odesílání e-mailových varování podle nastavení událostí jednotlivých akcí.
- Rozhraní umožní editovat stávající nastavení událostí.
- Rozhraní umožní mazat stávající nastavení událostí.
- Rozhraní umožní získání seznamu nastavení událostí.
- Rozhraní umožní získání seznamu varovných zpráv vytvořených akcemi.
- Rozhraní umožní mazání seznamu varovných zpráv vytvořených akcemi.
- Rozhraní umožní získání seznamu indexů a jejich mapování.
- Rozhraní umožní získání informací o klastru indexovacího nástroje.

Nefunkční požadavky

- Rozhraní bude pracovat s daty v téměř reálném čase.
- Rozhraní půjde jednoduše škálovat.
- Rozhraní bude napsané v jazyce Java.

1.1.4 Klientská aplikace

Funkční požadavky

- Klientská aplikace umožní zobrazení informací o všech akcích, vyhledávacích i agregačních pravidlech a událostech.
- Klientská aplikace umožní editovat či vytvářet nová vyhledávací i agregační pravidla a události a výsledná data odešle na RESTful API.
- Klientská aplikace umožní vytvářet nové akce a definovat vztah vyhledávacích či agregačních pravidel a výsledná data odešle na RESTful API.
- Klientská aplikace umožní editovat stávající akce a definovat vztah vyhledávacích či agregačních pravidel a výsledná data odešle na RESTful API.
- Klientská aplikace umožní zobrazení výsledků vyhledávání.

- Klientská aplikace umožní zobrazení varování.
- Klientská aplikace umožní vytvářet a zobrazovat grafy na základě agregačních pravidel.
- Klientská aplikace umožní zobrazovat mapy na základě geolokačních údajů.

Nefunkční požadavky

- Pro vývoj klientské aplikace bude použit JavaScriptový framework AngularJS 2.

1.2 Profil uživatele systému

1.2.1 Administrátoři systémů

Typickým uživatelem systému pro zpracování textových logů z několika systémů jsou administrátoři, kteří díky přehledu informací z těchto vstupů mohou sledovat události, které se v aplikacích odehrávají. Díky možnosti spravovat akce a události, které jsou akcemi generovány, mohou být administrátoři upozorněni na změny v systému v téměř reálném čase.

1.2.2 Bezpečnostní konzultanti

Bezpečnost je velmi důležitým aspektem každého systému. Bezpečnostní konzultant může například nadefinovat akce hlídající lokace uživatelů na základě jejich ip lokace a identifikátoru.

1.2.3 Marketingová oddělení

Marketingová oddělení mohou být informována o zvýšeném přístupu na určitou sekci sledovaného e-shopu a upravit marketingové zprávy zabývající se danou sekcí.

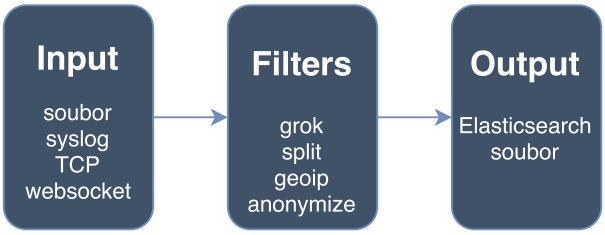
1.3 Výběr technologií

1.3.1 Logstash, Grok a geoip

Logstash[1] je nástroj využívaný ke sběru, transformaci a přenosu zpráv. Aplikace je napsána v jRuby, je distribuována pod otevřenou licencí a podporuje přenos různých textových i binárních vstupů v téměř reálném čase.

Tento modulární, otevřený systém využívá takzvaný třífázový proces. Data jsou nejdříve sebrána na určitém vstupu. Tento vstup může být například textový soubor, do kterého zapisuje aplikace. Logstash však dokáže poslouchat

na určeném portu, či dokonce sbírat data z webového soketu. Po sebrání dat dojde k jejich transformaci pomocí filtrů. Jelikož je Logstash založený na zásuvných modulech, lze získat velké množství již implementovaných a otestovaných zásuvných modulů. Díky otevřenosti projektu a dobré dokumentaci není pro vývojáře problém napsat vlastní modul dle potřeb. Data v těchto filtrech mohou být rozdělena na části, může dojít k syntaktickému rozboru podle mapování v konfiguračním souboru [1], nebo mohou být anonymizována či naopak obohacena o další informace. Příkladem obohacení informace může být doplnění lokace podle ip adresy, jak to dělá modul geoip. Po transformaci dat dojde k jejich přenosu na vydefinovaný koncový uzel. Těchto uzlů může být více, například Elasticsearch a zároveň textový soubor.



Obrázek 1.1: Třífázový proces zpracování vstupních zpráv nástrojem Logstash

Logstash je konfigurován pomocí JSON souboru, který mu je předán jako parametr při spuštění. Tento konfigurační soubor typicky obsahuje tři části. V části **input** se definuje zdroj dat. V části **filters** jsou nastavena transformační pravidla. Logstash a Elasticsearch nabízejí možnost automatického rozdělení jednotlivých řádků vstupního textu na sloupce. Tato analýza však není vždy správná, proto Logstash nabízí modul Grok, který slouží k rozdělení vstupu na sloupce podle regulárního výrazu zapsaného v konfiguraci. Grok však neslouží pouze k rozložení vstupních dat. Opět dokáže data doplnit o další informace, jako je například přidání dalšího pole se zprávou. Výsledná zpráva je následně odeslána na koncový uzel, který je definován v části **output**.

```
input {
 file {
    path => "/tmp/logs/log.log"
      start_position => "beginning"
   }
}

filter {
 grok {
    match => { "message" => "(%{IP:vzdalena_ip})?\s*(%{WORD:cn})
        " }
   }
```

[1] například za použití modulu grok

```
    geoip {
     source => "vzdalena_ip"
    }
}

output {
 elasticsearch {
    hosts => ["localhost:9200"]
    index => "logstash"
   }
}
```

Ukázka zdrojového kódu 1.1: Ukázka konfigurace nástroje Logstash

Typickým výstupem pro Logstash je index nástroje Elasticsearch, který poslouchá na určitém portu a předaná data dále zpracovává.

1.3.1.1 Beats

Logstash však nemusí sloužit jako primární nástroj pro přenos zpráv. Tento nástroj je výpočetně náročný na hostitelský systém a proto není vždy vhodné jej instalovat na koncový server, kde sbírá data. Samotnou transformaci lze provést na jiném serveru, který má dostatek výkonu. Pro přenos zpráv lze poté využít právě nástroj Beats[2], který není tak výpočetně náročný jako Logstash. Pro ještě větší úsporu dat a výkonu existuje mnoho různých konfigurací tohoto nástroje pro různá data s nimiž má pracovat. Lze tedy nainstalovat aplikaci, která je uzpůsobena pouze pro log zprávy a tím ušetřit potřebné místo a výkon serveru.

1.3.2 Elasticsearch

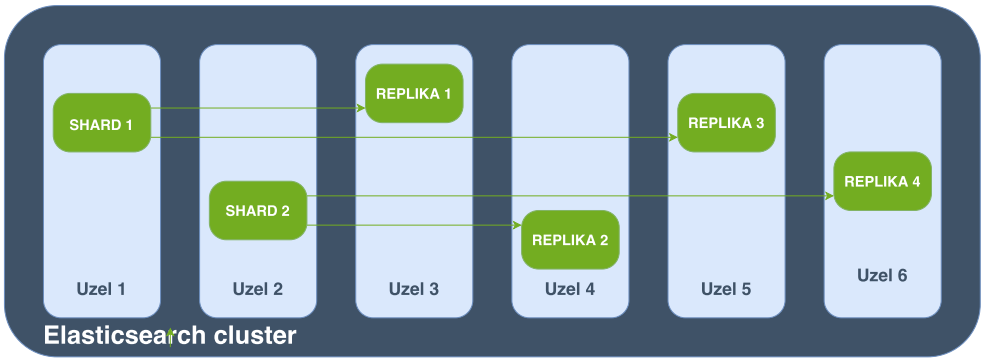
Elasticsearch[3] je vyhledávací a analytický indexovací nástroj, který je postaven na Apache Lucene a využívá jej k vyhledávání a indexaci vstupních dat v téměř reálném čase. Nástroj Elasticsearch podporuje indexaci textových vstupů, ale díky jeho otevřenosti a dobré dokumentaci lze doinstalovat mnoho zásuvných modulů, které podporují indexaci dalších formátů, jako jsou například PDF či XML soubory. Data se ukládají do takzvaných indexů, kde jsou analyzována a přístupná pro další operace jako je vyhledávání či agregace. Při vložení nového záznamu do indexu dojde k rozložení záznamu do jednotlivých polí podle definovaného mapování. Díky tomu, že indexy nemají schéma, lze v jednom indexu uchovávat záznamy s různým mapováním.

Každý index může potenciálně překročit fyzické možnosti hostitelského počítače. Proto je možné každý index rozdělit na menší oddíly. Těmto oddílům se v rámci nástroje Elasticsearch říká *shards*. Při vytváření indexu lze definovat kolik oddílů se má vytvořit. Poté je každý oddíl dostupný jako samostatný index, ale zároveň i jako součást nadřazeného indexu.

Jednou z velkých předností nástroje Elasticsearch je jeho škálovatelnost[4]. Po instalaci běží Elasticsearch v klastru obsahujícím uzly. V každém uzlu běží jedna instance. Jednotlivé uzly mezi sebou sdílejí data i zátěž. Každý klastr obsahuje jeden hlavní uzel, který vytváří a odebírá nové uzly podle momentální vytíženosti klastru. Hlavní uzel navíc provádí kontrolu zdraví jednotlivých uzlů a při kolapsu dokáže replikovat jejich data nebo uzly úplně nahradit. Celý klastr se pro uživatele stále tváří jako jediná instance.

Při vložení nového záznamu, je tento záznam uložen na primární uzel a následně zreplikován na další. Administrátor může ovládat počet replikovaných uzlů přemísťovat repliky či nastavit událost při pádu. Jelikož tento software slouží k indexování velkého objemu dat, je poměrně náročný na paměť. Ideální hardwarové požadavky na paměť RAM jsou od 16 do 64 GB. Velkou zátěží pro paměť jsou agregace, pro které je doporučeno 32 a výše GB RAM paměti. Elasticsearch připraven pro běh v Docker kontejneru.

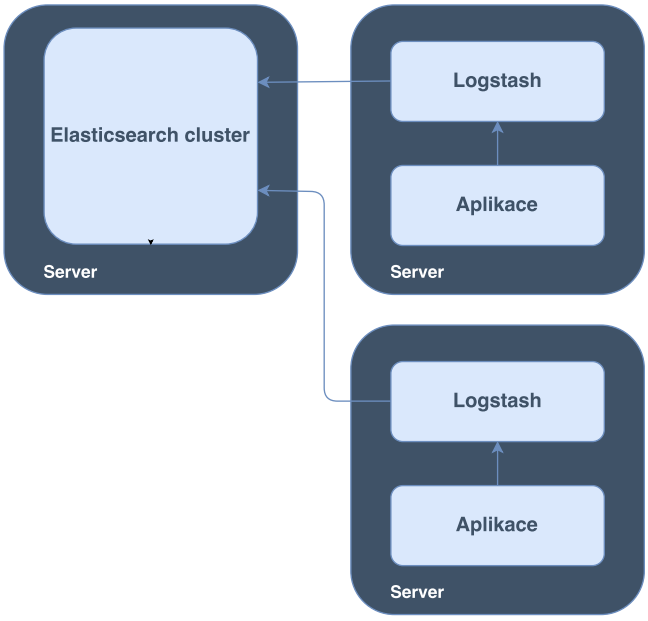
Kontejnerizací jednotlivých komponent pomocí nástroje Docker se zabývá sekce 1.3.8 a kapitola 4.



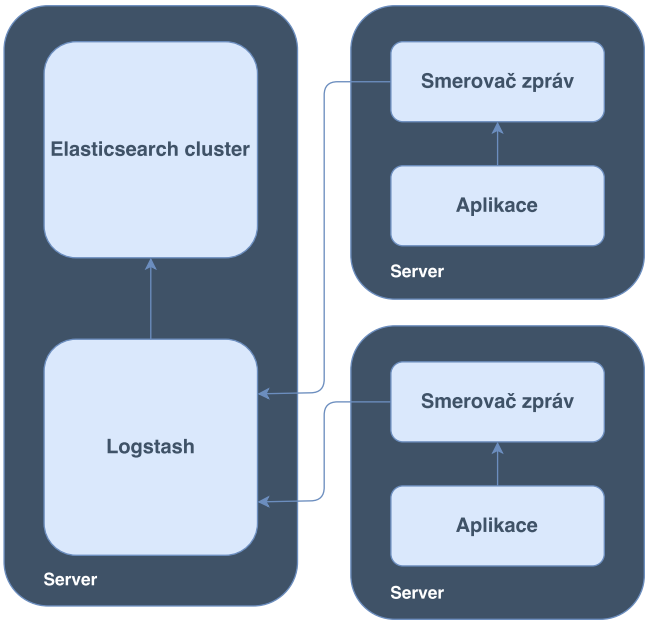
Obrázek 1.2: Uložení a replikace obsahu uzlů uvnitř Elasticsearch klastru

Vložení nových záznamů do indexu je možné několika způsoby. Nejjednodušší možností je vložení nových dat skrze RESTful API, které je vystaveno po instalaci nástroje. Skrze API lze tento nástroj ovládat, dále je možné vyhledávat či agregovat. Další možností je využití dávkové API pro vkládání dávkových dat. Možností je také vložení dat skrze Java API[5], či API pro jiný programovací jazyk. Spolupráce nástroje Elasticsearch a nástroje Logstash jako zdroje dat, je jednou z nejvyužívanějších možností vkládání dat. Tyto nástroje nemusí běžet na stejném serveru, jednou z možností je umístit Logstash klienta na každý server, kde chceme sbírat data a odesílat je do určitého klastru. Další možností je umístění instance nástroje Logstash na samostatný server, kde bude sbírat data, která k němu budou přenesena ze zdrojových

serverů a přeposílat je na třetí server, kde poběží Elasticsearch.



Obrázek 1.3: Schéma rozložení Elasticsearch klastru a nástroje Logstash při směrování zpráv aplikací



Obrázek 1.4: Schéma rozložení Elasticsearch klastru, nástrojů Logstash a Beats při směrování zpráv aplikací

Elasticsearch je využívaný pro široké možnosti vyhledávání a agregace. Při vyhledávání je možné využít dva režimy, fulltextové vyhledávání a vyhledávání na základě hodnoty určitého pole. Pokud chce uživatel využít fulltextové vyhledávání, musí využít obrácené indexy. Obrácený index[6] se vytváří při analýze dokumentu, kdy dojde k vytvoření seznamu obsahujícího každou unikátní hodnotu v dokumentech a její výskyt v každém dokumentu.

Pro příklad fulltextového vyhledávání si lze představit situaci se dvěma záznamy, kdy každý záznam obsahuje pole *content* s následujícími hodnotami:

- The quick brown fox jumped over the lazy dog.
- Quick brown foxes leap over lazy dogs in summer.

Při vytvoření obráceného indexu dojde k rozdělení každého řetězce na jednotlivá slova, která jsou uložena do seznamu všech slov všech záznamů. Společně s tímto seznamem slov dojde k vytvoření dalších seznamů, které obsahují informaci o výskytu daného slova v daném záznamu. Při fulltextovém vyhledávání dochází k hledání právě v těchto seznamech výskytů.

Slovo	záznam 1	záznam 2
Quick		X
The	X	
brown	X	X
dog	X	
dogs		X
fox	X	
foxes		X
in		X
jumped	X	
lazy	X	X
leap		X
over	X	X
quick	X	
summer		X
the	X	

Tabulka 1.1: Obrácené indexy nástroje Elasticsearch[6]

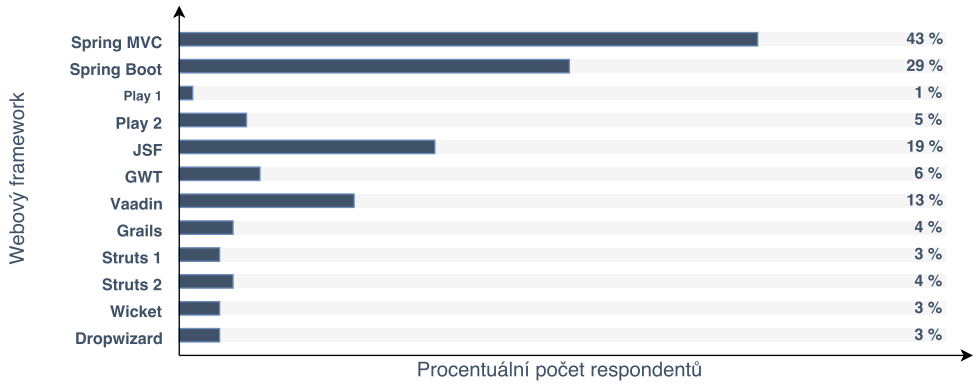
Další funkcionalitou pro získávání informací z dat je možnost agregovat nad zadanými daty. Agregace dat funguje na podobném principu jako agregace v jazyce SQL. Dojde tedy k seskupení záznamů podle určitého pole či polí.

1.3.2.1 X-Pack

Rozšíření X-Pack je skupina zásuvných modulů, které využívá jak Elasticsearch, tak i případně Kibana. Tyto moduly obsahují mnohá rozšíření pro zabezpečení, monitorování či sledování změn ve vstupních datech v rámci celého Elasticsearch ekosystému. V tuto chvíli, během analýzy a vývoje systému dataWatcher, je nejaktuálnější verze nástroje Elasticsearch i jeho rozšíření X-Pack s označením 2.4. Již je však avizována nová verze, kde dojde k výraznému přepracování nástroje Elasticsearch, jeho Java API, ale i rozšíření X-Pack. Aktuální skupina samostatných rozšíření, jako je například Watcher[7], bude sloučena právě do rozšíření X-Pack. Z tohoto důvodu, bylo během analýzy rozhodnuto, že funkcionality těchto rozšíření, včetně zabezpečení Shield, budou implementovány až po ustálení nových verzí a tedy nebudou v rozsahu této bakalářské práce. Samotné Java API nástroje Elasticsearch sloužící k vyhledávání a agregaci dat bude povýšeno ve chvíli, kdy bude dostupné v repositářích balíčkovacího systému Maven. Ukázky zdrojových kódů, i samotné systémové API tedy budou využívat nové verze 5.x.

1.3.3 Spring

Pro vývoj Java[8] aplikací je na trhu mnoho placených i volně licencovaných projektů, které usnadní vývoj. Spring projekt[9] patří mezi nejčastěji používaný framework na trhu. Jak je vidět na výzkumu společnosti RebelLabs, tvoří Spring Boot[10] a Spring MVC dominantní část využívaných technologií pro vývoj webových aplikací.



Obrázek 1.5: Procentuální využití webových technologií[11]

Standardní aplikace využívající Spring framework obsahuje jednotlivé knihovny této platformy a vývojář je poté konfiguruje pomocí Java či jiných konfiguračních souborů. Tím je zajištěna vysoká modularita jednotlivých částí. Idea

Spring Boot projektu je však jiná. Veškeré knihovny v aplikaci jsou již předkonfigurované. Zároveň startovací balíčky obsahují více knihoven. Další možností usnadnění vývoje je autokonfigurace projektu pomocí anotací. Vývojář například označí třídu jako *@Service* a Spring framework se již postará o její injektáž v místě, kde je potřeba. Spring Boot také podporuje dvě možnosti kompilace. Základními možnostmi je kompilace do .war balíčku, který je poté nasazen na aplikační server, nebo do .jar balíčku, obsahujícího zároveň lehký aplikační server Tomcat či Jetty. Takto zabalená aplikace lze spustit jako samostatný proces a například nasadit uvnitř Docker kontejneru.

1.3.4 Spring Data JPA

Spring Data JPA[12] projekt je součástí projektu Spring framework. Tento projekt přináší zjednodušení vývoje datové vrstvy aplikace. Chce-li vývojář využít auto-generování kódu DAO tříd[2], vytvoří mapovací Java třídu pro entitu a rozhraní, které dědí od rozhraní **CrudRepository**. V tomto rozhraní vývojář vytvoří metody bez těla a během kompilace dojde k vytvoření celých metod, které jsou v aplikaci využívány.

1.3.5 Rest

REST[13], neboli Representational state transfer je datově orientovaný architektonický styl, který zajišťuje jednotný a jednoduchý přístup ke zdrojům. Zdrojem je myšlen jakýkoliv konkrétní objekt, nebo i stav aplikace, lze-li jej popsat konkrétními daty. Každý zdroj má svoji jedinečnou URI, skrze kterou je k němu přistupováno.

Rest jako soubor pravidel představil Roy Fielding v roce 2000 v rámci své dizertační práce. REST zde využívá HTTP protokol pro odesílání dat skrze metody GET, POST, PUT, DELETE a další. RESTful rozhraní, rozhraní navrženo podle principů REST, slouží k provádění CRUD operací, kdy jsou tyto operace implementovány právě pomocí zmíněných HTTP metod.

REST je založen na paradigmatu klient – server, kde server poskytuje zdroje a provádí operace, které požaduje klient voláním specifických URI. Příkladem volání může být například: *http://localhost:3000/action/1*, kdy na základě odeslané operace [3] server rozpozná, jak má zdroj s daným identifikátorem upravit. Na základě zadané URI server navrátí požadovaná data nebo zavolá definované akce. Každá odpověď vrácená serverem, obsahuje také stavový kód, podle kterého klientská aplikace pozná výsledek operace. Server neodesílá klientovi přímo zdroje, ale jejich reprezentaci v předem definované podobě. Nejčastěji mohou být data zaslána v podobě JSON či XML formátu, lze však použít i jiné formáty jako RSS, YAML a další. Klient nejčastěji

[2] Data access object jsou třídy, vytvářející rozhraní pro manipulaci s databázovými objekty

[3] například DELETE

serveru odesílá upravené reprezentace dat ve stejném formátu, ve kterém je přijal.

Jedním z důležitých pravidel při návrhu RESTful rozhraní je pravidlo idempotence, kdy server odpoví na stejnou operaci, vždy stejnou odpovědí. Pokusí-li se klient například vymazat zdroj s identifikátorem 1 a operaci DELETE odešle vícekrát, server mu na všechna volání odpoví úspěchem [4]. Dalším pravidlem je pravidlo HATEOAS [5], které říká, že každý další stav aplikace, lze získat pomocí stavů, které byly doposud získány z posledního stavu.

Jako velkou výhodu RESTful rozhraní, lze považovat jeho bezstavovost. Tím lze dosáhnout paralelního zpracování jednotlivých dotazů. Každý dotaz na server obsahuje veškerá data, která jsou k vykonání dané operace na straně serveru potřeba. Díky tomu je možné servery škálovat a jednotlivé dotazy odesílat na ten server, který je momentálně přístupný a má volný výpočetní čas.

1.3.6 JSON

JSON (JavaScript Object Notation) je jednoduchý formát připravený pro výměnu dat, který je uzpůsobený jak pro jednoduché čtení člověkem, tak i pro rychlé strojové zpracování. Tento textový formát není závislý na počítačové platformě a je podporován mnoha programovacími jazyky. Struktura JSON formátu je založena na kolekci párů ve tvaru *název:hodnota* nebo na seřazeném seznamu hodnot ve tvaru *název:[hodnota 1, hodnota 2]*. Obě možnosti lze do sebe volně kombinovat. JSON je momentálně jedním z nejpoužívanějších formátů pro výměnu dat v RESTful rozhraní. Další často využívanou možností je výměna dat ve formátu XML.

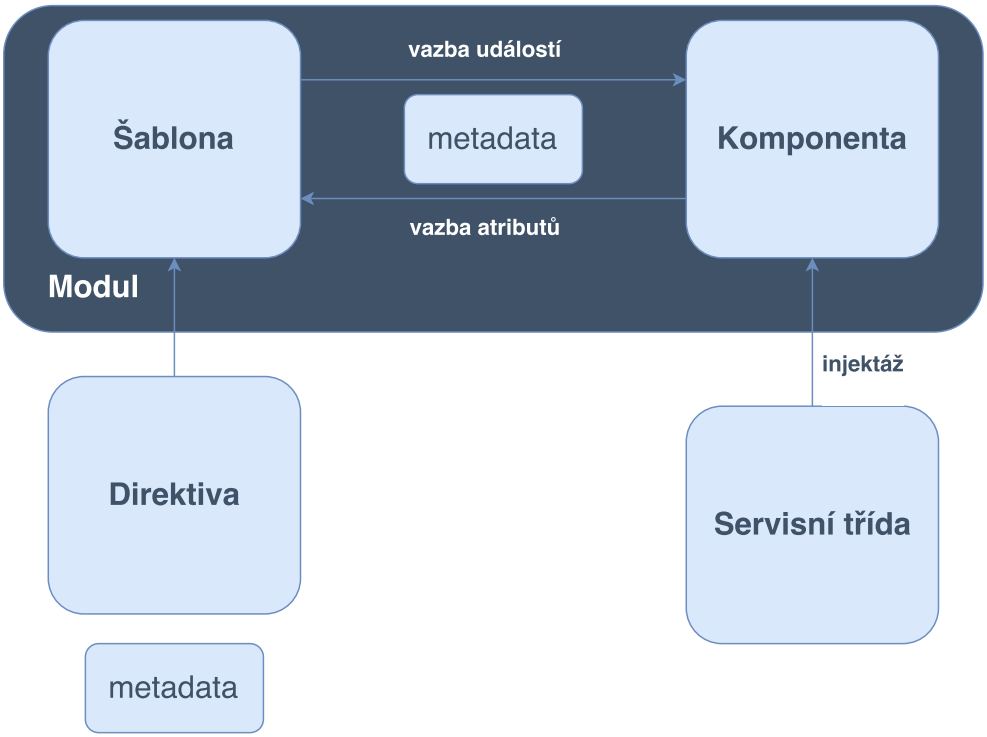
1.3.7 AngularJS 2

Současným trendem vývoje webových aplikací je vytvoření RESTful API a *single page* JavaScript aplikace běžící na straně klienta. Jednou z mnoha platforem pro vývoj těchto aplikací je AngularJS ve verzi 2 [15]. Tento poměrně mladý framework nevychází ze svého předchůdce, AngularJS. Nejedná se totiž o evoluci, ale o úplné přepsání. Tento framework, jak uvádí jeho tvůrci, podporuje vývoj pro více platforem naráz. Lze zde vyvíjet jak nativní mobilní aplikace, desktopové aplikace, tak i webové aplikace běžící v prohlížeči. AngularJS 2 přinesl mimo jiné podporu jazyka TypeScript, což je programovací jazyk kompilovaný do jazyka JavaScript vytvořený společností Microsoft. TypeScript se stal standardem pro vývoj aplikací a AngularJS 2.

AngularJS 2 aplikace se skládá z modulů[16]. Každá aplikace má minimálně jeden modul a každý modul se skládá z komponent, které spolu logicky

[4] nedojde-li na serveru k nějaké neočekávané chybě

[5] Hypermedia As The Engine Of Application State



Obrázek 1.6: Architektura AngularJS 2 komponent[14]

souvisí. Dále každý modul obsahuje routing modul, což je třída sloužící k navigaci a třídu anotovanou *@NgModule*, kde je popis tříd podle, kterých se AngularJS 2 aplikace kompiluje. AngularJS 2 dále podporuje vkládání závislostí[6], kdy lze opět pomocí anotací označit třídu jako připravenou na injektáž do komponent, či do jiných tříd.

Každá komponenta se skládá z šablony, která obsahuje HTML kód doplněný o AngularJS 2 direktivy a ze samotné komponenty obsahující kód a logiku.

1.3.8 Docker

Docker[17] je otevřený software umožňující zabalit aplikace do virtuálních kontejnerů. Na rozdíl od standardní virtualizace pomocí nástroje Vagrant či Virtualbox rozhraní CLI, se při použití nástroje Docker nevirtualizuje celý virtuální stroj, ale pouze linuxový proces. Takovýto proces se nazývá kontejner. Kontejner neobsahuje celý operační systém, ale pouze knihovny a nastavení, které je potřeba k jeho běhu. Tyto knihovny a nastavení jsou určeny v bázovém obrazu

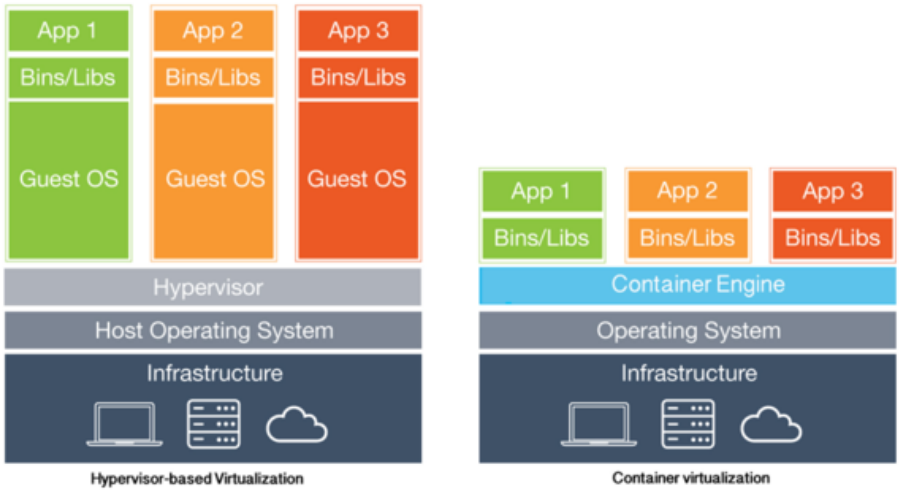
[6]Dependency injection

kontejneru. Díky tomu má Docker kontejner menší režii než plnohodnotná virtualizace, navíc je zaručeno, že aplikace bude mít vždy stejné běhové prostředí nezávisle na tom, kde je nasazena.

Pro konfiguraci obrazu se používá Dockerfile[18]. Jedná se o textový popis sestavení obrazu, kde se provádí prvotní nastavení systému. Zde je určen bázový obraz systému, dále se provede nastavení systému a nasazení aplikace a její spuštění. Tento Dockerfile se používá buďto samostatně s použitím nástroje Docker, nebo pokud spolu několik obrazů souvisí, pomocí nástroje Docker-compose.

Docker-compose je kompozice několika kontejnerů, které spolu určitým způsobem souvisí. Jedná se opět o textový popis procesů, který spouští jednotlivé kontejnery z několika různých obrazů.

Jednotlivé kontejnery jednoho obrazu, spravuje orchestrátor. Orchestrátor je aplikace, která kontroluje stav jednotlivých kontejnerů, jejich zatížení a v případě potřeby vytváří nové kontejnery, či vypíná ty nepotřebné.

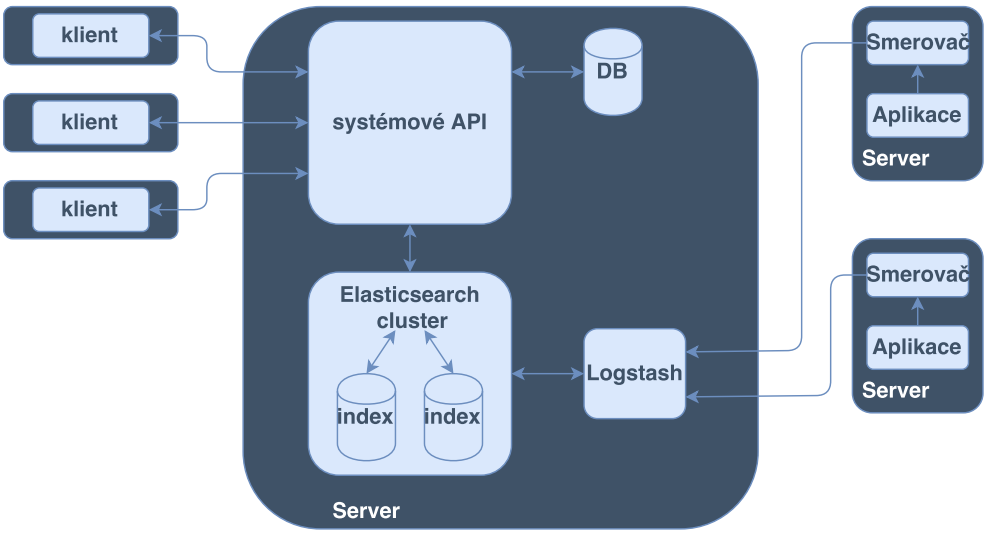


Obrázek 1.7: Architektura Docker kontejneru oproti standardní virtualizaci[19]

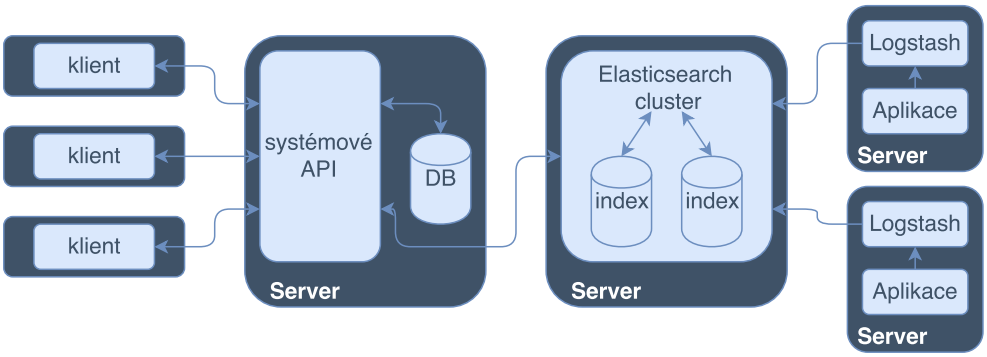
1.4 Architektura systému

Před implementací systému splňujícího funkční i nefunkční požadavky je potřeba správně navrhnout jeho architekturu a jeho jednotlivé komponenty. Každý prvek systému má svoji jedinečnou funkci, ale zároveň je každý prvek systému závislý na dalších částech systému.

Na diagramech 1.8 a 1.9 je vidět možné rozložení komponent systému a jejich vzájemná komunikace stejně jako komunikace uživatele se systémem. Tato architektura je pouze abstraktní, v reálném nasazení je možné všechny prvky systému nasadit na jeden fyzický server, nebo naopak dekomponovat na velké množství fyzických serverů.



Obrázek 1.8: Serverové rozložení komponent systému 1



Obrázek 1.9: Serverové rozložení komponent systému 2

Celý systém se skládá z následujících částí:

- instance nástroje Logstash a směrovač zpráv,
- Elasticsearch klastr a jeho indexy,

- databázový server,
- RESTful API aplikace,
- klientská aplikace.

1.5 Komponenty systému

1.5.1 Instance nástroje Logstash a směrovač zpráv

Nástroj Logstash je v systému použit jako směrovač a transformátor zpráv. Stejně tak, jako Logstash, je možné použít další aplikace třetích stran, či aplikaci na transformaci a předání zpráv implementovat. Logstash lze umístit na každý server generující zprávy, kde převezme data, transformuje je a odešle do Elasticsearch klastru. Transformace zpráv má určitou režii, proto pokud je na každém serveru umístěna instance tohoto nástroje, musí být tyto servery dostatečně výkonné. Další možností je umístit Logstash na samostatný server a zprávy z jednotlivých serverů přesměrovat na určitý port a nakonfigurovat Logstash, aby četl data z tohoto portu, či data odesílat pomocí méně náročného nástroje. Velice vhodným nástrojem pro přenos dat je elastic Beats. Pro přenos dat je možné také nakonfigurovat samotnou aplikaci, která logy vytváří.

1.5.2 Elasticsearch klastr a jeho indexy

Elasticsearch klastr funguje jako samostatná komponenta, do které jsou data po transformaci odeslána. Zde jsou uložena do jednotlivých indexů a proběhne jejich analýza. Po uložení a analýze lze nad daty provádět vyhledávání a agregace. Tato komponenta systému komunikuje se systémovým API aplikace pomocí Java API. Elasticsearch klastr může běžet na samostatném serveru, nebo může být umístěn na stejném fyzickém serveru jako je systémové API.

Elasticsearch klastr lze jednoduše škálovat do šířky i do výšky. Při běhu lze nastavit kolik systémové paměti lze využít, nebo přidávat další uzly do klastru. Díky připravenosti tohoto systému na běh uvnitř kontejneru, je také možné ovládat životní cykly skrze orchestrátor.

1.5.3 Databázový server

Systém při běhu potřebuje určité servisní informace. Mezi tyto informace patří například informace o připojení, či nastavení agregačních a vyhledávacích dotazů. Tyto data jsou uložena na databázovém serveru, ke kterému se připojuje systémové API.

1.5.4 RESTful API systému

O logiku systému se stará systémové API. Klientská aplikace odesílá požadavky uživatelů skrze HTTP protokol a API jejich požadavky zpracovává a vrací požadovaná data. API dále komunikuje s Elasticsearch klastrem skrze Java API rozhraní. API odesílá klastru požadavky na vyhledávání a agregace, které požaduje klientská aplikace, nebo v rámci akcí prohledává vstupní data, dále komunikuje s databázovým serverem, kde získává informace o agregacích, připojeních a dalších entitách.

API umožňuje vytváření vyhledávacích akcí, která v sobě drží několik agregací či dotazů. Pro každou akci lze vytvořit podmínka, při jejímž splnění dojde k vygenerování události a upozornění uživatele.Systémová část systému se také stará o odeslání e-mailových zpráv administrátorům. E-mailová zpráva je odeslána v případě splnění podmínky nastavené v akci.

RESTful API je připraveno na kontejnerizaci a škálování. Díky bezstavovosti požadavků lze jednoduše přidávat další instance do klastru, který je obsluhován orchestrátorem nebo nad jednotlivými kontejnery běžícím vyvažovačem zátěže[7]. Pokud je použit orchestrátor, pak supluje vyvažování zátěže, vytváří nové kontejnery s aplikací při vyšší zátěži a ukončuje nadbytečné kontejnery.

1.5.5 Klientská aplikace

Klientská aplikace slouží k interakci mezi uživatelem aplikace a RESTful API. Veškerá komunikace putuje skrze HTTP protokol.

1.6 Zabezpečení systému a rozšíření X-Pack

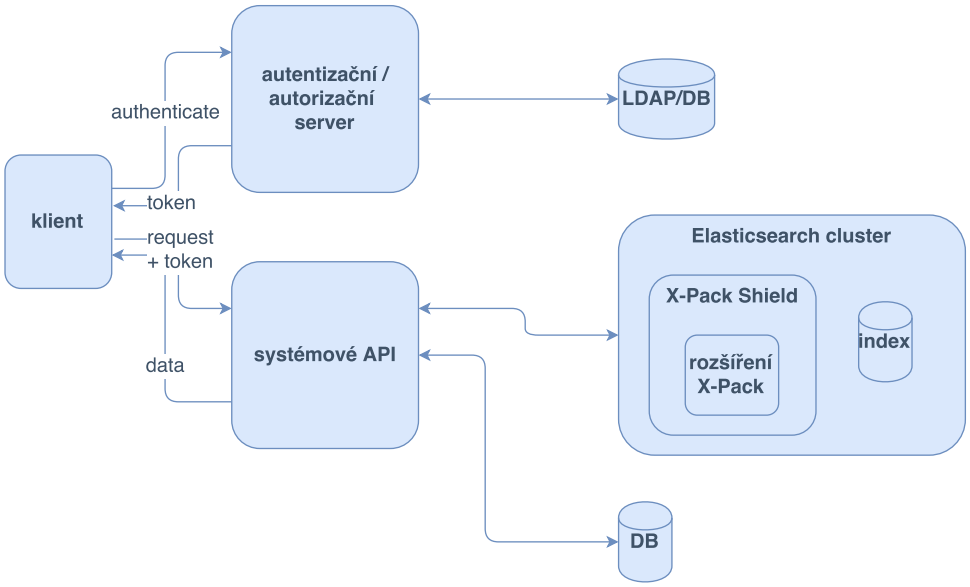
Jak již bylo zmíněno v kapitole 1.3.2.1, která se zabývala rozšířením funkčnosti nástroje Elasticsearch moduly X-Pack, bude implementace zabezpečení celého systému probíhat v rámci budoucího vývoje. Toto rozhodnutí bylo dále podpořeno možností budoucího napojením systému dataWatcher na Access Gateway NEWPS.CZ, které slouží jako jednotný autentizační a autorizační bod pro aplikace v rámci dané organizace.

S budoucím zabezpečením systému je však třeba počítat již při jeho návrhu. Proto je v této kapitole popsán teoretický model zabezpečení systému.

Spring Boot aplikace umožňují několik druhů zabezpečení díky knihovnám Spring Security, které podporují pokročilé metody autentizace i autorizace. Pomocí těchto knihoven lze zabezpečit Java aplikace. Tato bezpečnostní platforma také podporuje autentizaci a autorizaci pomocí externího serveru vracejícího bezpečnostní tokeny. Tímto serverem může být NEWPS.CZ Access Gateway, nebo také Facebook, Google či jiný server vyvinutý pouze pro účely

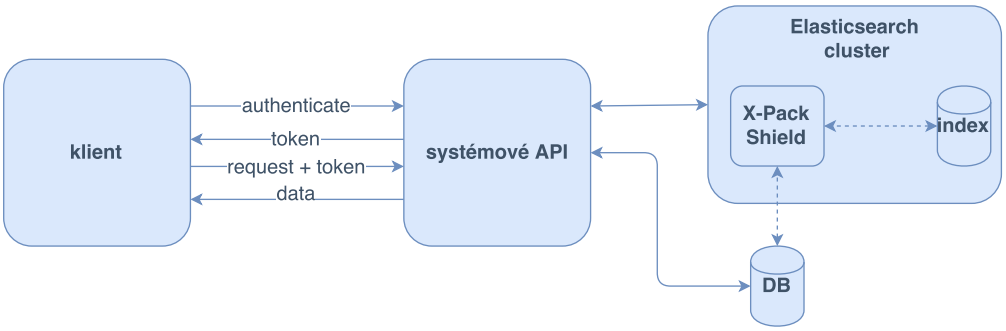
[7] Vyvažování zátěže, neboli load balancing, je technika rozložení zátěže mezi několik uzlů

systému dataWatcher. Autorizační a autentizační server komunikuje s databází, či s jiným zdrojem dat. Bohužel, bezpečnostní rozšíření X-Pack knihovny Shield, nepodporuje autentizaci ani autorizaci pomocí tokenů[20], proto bude v budoucnu vyvinuto rozšíření pro tento typ autorizace a autentizace[21].



Obrázek 1.10: Schéma zabezpečení systému pomocí samostatného autentizačního a autorizačního serveru

Pokud by systém nevyužíval autorizační a autentizační server, lze využít ověření pomocí databáze, či samostatného Elasticsearch indexu. Systémové API by v tu chvíli fungovalo i jako autorizační a autentizační server pro klientské aplikace. Uživatelská data by byla uložena v databázi, či na samostatném Elasticsearch indexu. X-Pack by poté tato data získal a provedl autorizaci k přistupovaným datům.



Obrázek 1.11: Schéma standardního zabezpečení systému oproti databázi či Elasticsearch indexu

Zabezpečení klientské aplikace bude spočívat v instalaci bezpečnostních knihoven, s jejichž pomocí bude upravena navigace uvnitř aplikace. Přihlašovací obrazovka bude zprostředkována buďto autentizačním serverem, pokud bude využit, nebo klientskou aplikací. Po přihlášení uživatele se zobrazí standardní obrazovky, jejichž data budou vrácena systémovým API. Na základě uživatelské role API vrátí pouze ta data, ke kterým má uživatel přístup. Při vyhledávání zajistí autorizaci k nalezeným datům naopak X-Pack.

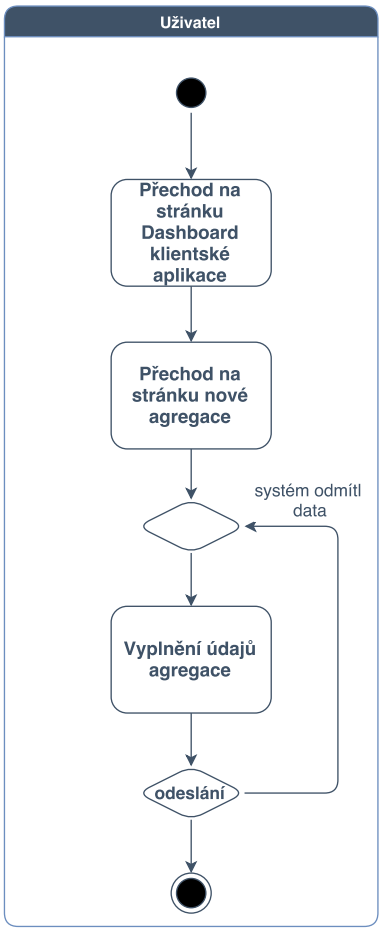
1.7 Procesy systému

V této sekci jsou vypsány některé nejčastější případy užití a jejich popis.

1.7.1 Vytvoření nové agregace

Pokud chce uživatel vytvořit novou agregaci, přejde na klientskou část systému, kde se mu zobrazí stránka Dashboard s výpisem všech akcí, agregací a vyhledávacích dotazů. Zde klikne na tlačítko plus v sekci agregací a je přesměrován na stránku pro vytvoření nové agregace. Na této stránce se nachází formulář pro zadání jednotlivých detailů nové agregace. Pokud uživatel nevyplní některé z povinných polí, formulář mu neumožní odeslání požadavku a zobrazí chybovou hlášku. Po úspěšném vyplnění formuláře odešle klientská aplikace HTTP POST požadavek na systémové API. API přijatý požadavek opět validuje, a pokud je vše v pořádku, přijatá data uloží a vrátí klientské

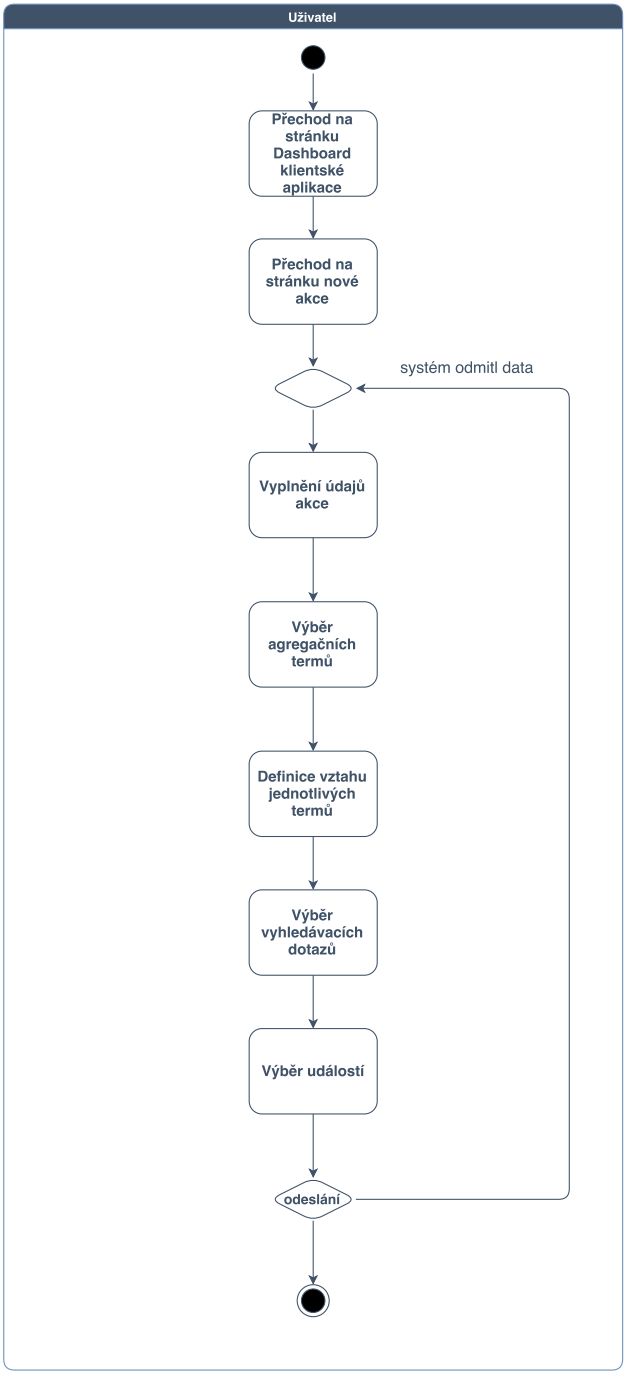
aplikaci novou agregaci s doplněním identifikátorem agregace. Po přijetí odpovědi serveru, klientská aplikace buďto zobrazí chybovou hlášku, pokud došlo k chybě na straně systémového API, nebo přejde na stránku s detailem nově vytvořené agregace. Zde se po načtení detailů o agregaci odešle HTTP GET požadavek na systémové API. V tomto požadavku je identifikátor agregace. API tento požadavek zpracuje, odešle požadavek na databázový server, kde získá informace o agregaci. Dále vytvoří dotazovací objekt, který odešle na Elasticsearch klastr. Ten podle typu připojení vyhledá v některém z indexů a vrátí nalezená data API. Systémové API tato data zpracuje a vrátí je v odpovědi klientské aplikaci. Klientská aplikace po přijetí výsledků zobrazí nalezené záznamy uživateli a zobrazí graf o nalezených hodnotách. Je-li agregace nastavena na pole obsahující geolokační informace, zobrazí se uživateli i mapa s nalezenými výsledky.



Obrázek 1.12: Model procesu vytvoření nové agregace

1.7.2 Vytvoření nové akce a zpracování výsledků

Pokud chce uživatel vytvořit novou akci s událostmi, přejde na klientskou část aplikace, kde se mu zobrazí stránka Dashboard s výpisem všech akcí agregací a vyhledávacích dotazů. Zde klikne na tlačítko plus v sekci akcí a je přesměrován na stránku pro vytvoření nové akce. Nejdříve vyplní popisující informace o akci a nastaví podmínku, kdy má akce vyvolat událost a časovač spuštění. Po nastavení těchto informací se uživatel přesune do sekce, kde je definován vztah agregací pro vytvářenou akci. Nejdříve klikne na tlačítko upravit v sekci agregací. V tu chvíli dojde k zobrazení všech agregací, které jsou uložené v databázi systému. Uživatel postupně kliká na jednotlivé agregace, které jsou přidávány do sekce vztah agregací. Zde uživatel mění pořadí agregací a definuje vztah, který mezi sebou jednotlivé agregace mají. Po úspěšném definování vztahů agregací akce, se uživatel přesune do sekce událostí, kde určí, které události mají být vyvolány je-li podmínka akce splněna. Pokud uživatel nechce používat agregace, ale vyhledávací dotazy, provede podobný výběr jako v sekci agregací s vyhledávacími dotazy. Po určení všech informací o akci uživatel odešle požadavek na systémové API, které data validuje a případně uloží. Systémové API vrátí klientské aplikaci detaily nově vytvořené akce společně s jejím identifikátorem. Klientská aplikace následně přesměruje uživatele na stránku s detailem akce a odešle HTTP požadavek na vyhledání podle identifikátoru akce. Systémové API na základě identifikátoru vyhledá potřebné detaily akce, sestaví vztah agregací. Pro jednotlivé agregace dojde k vytvoření dotazovacích objektů, jedná-li se o vnořené agregace, vytvoří se pro ně společný dotazovací objekt. Tyto objekty jsou po vytvoření odeslány skrze Java API na Elasticsearch klastr, který na základě jednotlivých připojení vyhledá v indexech a vrátí nalezená data. Tato data jsou sbírána do společného výsledku a vrácena jako odpověď HTTP požadavku. Klientská aplikace tato data zobrazí uživateli. API dále na základě časovače provádí vyhledávání nad vstupními daty a pokud je splněna podmínka akce, odešle e-mail uživateli či zobrazí varovnou zprávu v klientské aplikaci.



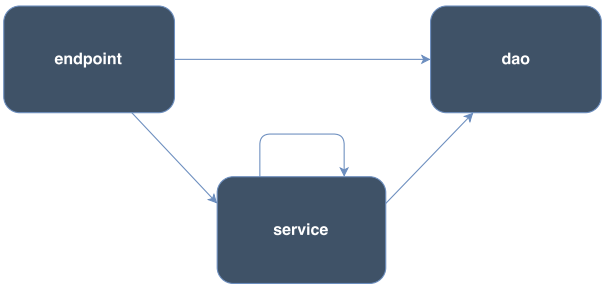
Obrázek 1.13: Model procesu vytvoření nové akce

Kapitola 2

Realizace

V této kapitole jsou popsány postupy, které byly použity při implementaci systému.

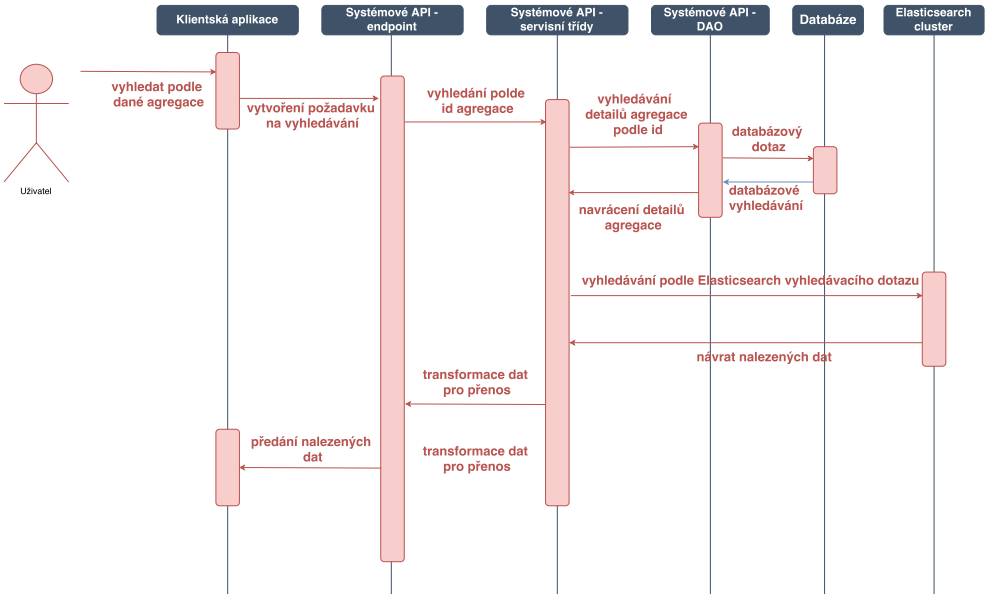
2.1 Komunikace uvnitř systémového API a jednotlivé vrstvy



Obrázek 2.1: Schéma komunikace uvnitř systémového API

Systémové API je rozloženo do tří logických vrstev. Nejnižší vrstva, starající se o přístup k databázi a mapování databázových tabulek na entity systémového API, se nachází v balíčku DAO. Třídy v tomto balíčku nevolají metody tříd z vyšších balíčků, pouze vracejí požadovaná data. Nad touto vrstvou se nachází balíček se servisními třídami. Třídy v tomto balíčku volají metody DAO vrstvy nebo dalších servisních tříd. Dále komunikují s Elasticsearch klastrem pomocí Java API. Nad touto vrstvou se nachází vrstva s kontrolery RESTful API. Tyto třídy volají metody, jak DAO vrstvy, tak servisních tříd.

2.1.1 Příklad komunikace uvnitř systémového API



Obrázek 2.2: Komunikace uvnitř systému při založení nové agregace

1. Uživatel pomocí klientské aplikace zažádá o vyhledání v klastru na základě agregace nebo vyhledávacího dotazu. Ve chvíli kdy klientská aplikace přejde na detail vyhledávacího objektu, odešle HTTP GET požadavek na **SearchEndpoint** systémového API.

2. Systémové API přijme tento požadavek a předá ho servisní vrstvě.

3. Servisní vrstva odešle požadavek DAO vrstvě.

4. DAO vrstva na základě identifikátoru vyhledávacího objektu vyhledá tento objekt v databázi a předá jej zpět servisní vrstvě.

5. Po přijetí tohoto objektu sestaví servisní vrstva dotazovací objekt pro Java API nástoje Elasticsearch a dotáže se na data.

6. Elasticsearch na základě identifikátoru indexu uloženého v dotazovacím objektu vyhledá data a vrátí je servisní vrstvě.

7. Ta tyto data zpracuje a předá je zpět třídě **SearchEndpoint**.

8. **SearchEndpoint** tato data předá zpět klientovi.

2.2 Realizace systémového Java API

2.2.1 Datová vrstva

Systém využívá dvě datová úložiště jako zdroje dat. Prvním je MySQL databáze, kde jsou uloženy konfigurační informace o jednotlivých objektech, skrze které probíhá vyhledávání. Dále jsou zde uloženy data o operacích, které definují vztahy vyhledávacích objektů, varovné zprávy a další systémové informace. Druhým zdrojem dat jsou Lucene indexy umístěné v Elasticsearch klastru. V těchto indexech se nachází samostatná data, nad kterými poté systém vyhledává. Jelikož jsou jednotlivé indexy bezschématové, není zde uvedena jejich struktura. Každý záznam je rozdělen na jednotlivé sloupce a jejich mapování je určeno v mapovacím filtru nástroje Logstash.

2.2.1.1 Popis entit systému

- Tabulka **Action** – v této tabulce se nachází informace o jednotlivých akcích. Akce je pomyslná obálka udržující agregace či vyhledávací dotazy. Administrátor při definici nové akce nastaví jednotlivé agregace a jejich vztah, podmínku spuštění a jaká událost se má vykonat v případě splnění podmínky.

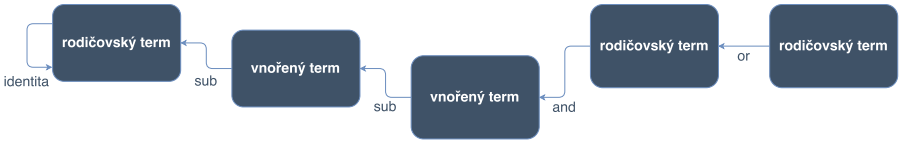
  Každá akce má definovaný název a popis. Dále jsou pro každou akci definované vztahy agregací v tabulce **ActionTermRelationShip** a vztahy vyhledávacích dotazů v tabulce **ActionQueryRelationShip**. Tyto tabulky určují, které dotazy a agregace náleží dané akci a skrze něž bude provedeno vyhledávání. Každá akce má určeno, jak často má být spuštěna. Na základě těchto údajů, dojde každých n minut k prohledání vstupních dat a jsou-li nalezeny nějaké záznamy, které splňují podmínku akce, je vyvolána událost. Vztah mezi akcí a událostmi je definován v tabulce **ActionJobRelationShip**.

- Tabulka **Aggregation** – v této tabulce se nachází informace o jednotlivých agregacích. Každá agregace má určené jméno a typ. Důležitým sloupcem je sloupec pole, kde se určuje podle kterého sloupce indexu budou data agregována. Informace o indexu jsou uložené ve sloupci připojení, jehož identifikátor má agregace rovněž k dispozici. Každá agregace shlukuje data jen do vymezené historie, lze tedy nastavit jak hluboko po časové ose budou data agregována. Dále lze nastavit počet záznamů, které budou zobrazeny a minimální počet výskytů záznamů, které se ve vstupních datech musí nacházet, aby byla data brána jako relevantní. Každá agregace lze doplnit o vyhledávací dotaz. Zápis těchto dotazů je uložen ve sloupci *queryString*.

- Tabulka **Term** – každá akce může definovat několik agregačních dotazů a jejich vztah mezi nimi. K uložení těchto informací slouží rekurzivní ta-

bulka **Term**, kde je definována rodičovská agregace, její potomek a vztah mezi těmito agregacemi. Vztah agregací je uložen v tabulce **Operation**. Podporované typy operací jsou následující:

– konjunkce,
– disjunkce,
– vztah typu rodič – potomek za pomoci vnořených agregací.



Obrázek 2.3: Vztahy mezi jednotlivými vyhledávacími termy

- Tabulka **Query** – v této tabulce se nachází informace o jednotlivých vyhledávacích dotazech. Každý dotaz má své jméno, informace o připojení a nastavený sloupec, kde mají být data vyhledána. Jedná-li se o fulltextový vyhledávací dotaz, je tento sloupec prázdný, ale naopak je vyplněn sloupec *queryString*. Opět je zde přidána možnost určení, jak hluboko po časové ose budou data kontrolována.
- Tabulka **Connection** – v této tabulce se nachází informace o jednotlivých indexech Elasticsearch klastru. Každé připojení má své jméno a jméno indexu, skrze které je identifikovatelné v klastru. Jelikož jde do jednotlivých indexů zapisovat data s různými schématy, má každé připojení definovaný ještě druhý identifikátor dat, který slouží k přesné identifikaci záznamů.
- Tabulka **Job** – v této tabulce se nachází informace o jednotlivých událostech. Každá událost je jednoduchý objekt obsahující pouze popis svého typu a zprávu, která se zobrazí uživateli.
- Tabulka **WarningMessage** – v této tabulce se nachází informace o varovných zprávách a jejich text.

2.2.2 DAO vrstva systémového API

Pro přístup k databázovým údajům využívá systémové API Spring data JPA framework. Každá entita, má vytvořenou POJO[8] třídu, doplněnou o metainformace pomocí anotací *javax.persistance*. Pomocí těchto anotací je JPA poskytovatel informován o jakou tabulku databáze se jedná, jaké sloupce daná

[8]Třída známá jako Plain Old Java Objects je standardní Java třída, která nepoužívá žádné další pokročilé knihovny a zároveň většinou neobsahuje logiku aplikace

tabulka má, a jak je namapovat na entity systémového API. Pro samotný přístup k datům je vytvořeno rozhraní, které dědí od Spring data **CrudRepository** rozhraní. Toto rodičovské rozhraní poskytuje metody pro základní CRUD operace s daty na základě hlaviček metod deklarovaných v poděděném rozhraní.

```
@Entity
@Table(name = "Connection")
public class ConnectionBO {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idConnection", unique = true, nullable =
        false)
    private int idConnection;

    @NotNull
    @Column(name = "name", length = 60, unique = true, nullable =
         false)
    private String name;

    @JsonIgnore
    @OneToMany(cascade = CascadeType.PERSIST, mappedBy = "
        connectionBO")
    private List<AggregationBO> aggregationBOList;

    //konstrokutory
    //getery a setery
}
```

Ukázka zdrojového kódu 2.1: Ukázka entity systému

```
public interface ConnectionDAO extends CrudRepository<
    ConnectionBO, Long> {
    ConnectionBO findByIdConnection(int id);
    List<ConnectionBO> findAll(boolean enable);

}
```

Ukázka zdrojového kódu 2.2: Ukázka DAO vrstvy systému při využití Spring Data JPA

Toto rozhraní je pak možné označit třídu jako připravenou na injektáž v dalších vrstvách systémového API a využívat jeho metody pro získání dat z databázového serveru.

2.3 Logická vrstva

Logická část systémového API lze rozdělit na dvě sekce. První sekce zastřešuje CRUD[9] operace, kdy volá metody datové vrstvy a předává je vrst-

[9]Create, Read, Update, Delete – Vytvoření, Čtení, Editace, Mazání záznamů

vám, které o data zažádaly. Může se jednat o kontrolery zpracovávající požadavky klientské aplikace nebo metody vyhledávacích servisních tříd. Druhou část tvoří servisní třídy vyhledávající data pomocí Elasticsearch Java API klienta.

2.3.1 Servisní třídy – CRUD

Metody spadající do servisních tříd, načtou data z databáze pomocí datové vrstvy, tato data doplní o další informace, či je transformují do DTO[10] objektů. Rozdílnou třídou od těchto tříd je **TermService**, která se stará o vytvoření vztahu mezi jednotlivými vyhledávacími termy. Tato třída obsahuje metody pro vytvoření seznamu agregačních termů, kde každý term má nastaveného rodiče a logický vztah s ním. Tato metoda je použita pro vytvoření agregačních termů, které jsou použity při vyhledávání. Druhá metoda slouží pro vytvoření seřazeného seznamu termů a nastavení jejich operace.

2.3.2 Elasticsearch Java API

Nástroj Elasticsearch nabízí několik API pro jednotlivé programovací jazyky. Jedním z těchto API je API pro obsluhu klastru z Java aplikací. Veškeré operace, které toto API nabízí jsou asynchronní a lze je provádět v dávkách. Díky asynchronii lze odeslat vyhledávací dotaz a ve chvíli, kdy je vyhledávání dokončeno, vrátí Java API nalezená data. Veškeré operace jsou tedy prováděny na straně Elasticsearch klastru. Pokud chce vývojář využít Elasticsearch Java API v aplikaci a využívá balíčkovací systém **Maven**, stačí pouze přidat závislost do *pom.xml* souboru a dojde ke stažení veškerých knihoven pro práci s Elasticsearch klastrem.

Pro připojení k indexovacímu klastru se využívají Java klienti, kteří slouží jak k administraci, tak i pro vyhledávání či agregace. Při implementaci komunikace systémového API s Elasticsearch klastrem byl použit **transportní klient**. Při vytváření transportního klienta je potřeba zadat URL a port na kterém běží Elasticsearch klastr. Klient se k klastru nepřipojí na přímo, ale komunikuje s ním pomocí transportní vrstvy.

```
private Client createClient(){
 try{
       Settings settings = Settings.builder().put(
           elasticsearchClusterName,
           elasticsearchClusterNameValue).build();

       elasticClient = new PreBuiltTransportClient(settings).
           addTransportAddress(new InetSocketTransportAddress(
           InetAddress.getByName(elasticsearchHost),
           elasticsearchPort));
```

[10]Data transfer object je objekt sloužící k přenosu dat mezi jednotlivými komponentami systému

```
    }catch (Exception e){
        throw new ElasticsearchException("Error occurred while
            creating search client!!");
    }
}
```

Ukázka zdrojového kódu 2.3: Ukázka získání transportního klienta Elasticsearch Java API

2.3.3 Servisní třídy pro vyhledávání pomocí Elasticsearch Java API

Jak pro vyhledávání, tak pro agregaci vstupních dat je použita servisní třída **SearchRequestBuilder**. Tato třída zabaluje dotazovací objekty a doplňuje je o informace o indexu. Každý dotaz je validován v rámci Elasticsearch Java API a není-li syntakticky správně, dojde k vyhození výjimky.

Systémové API využívá pro vyhledávání v Elasticsearch klastru dvě servisní třídy. První třída slouží k vyhledávání záznamů pomocí vyhledávacích dotazů. Každý vyhledávací dotaz obsahuje popis indexu, ve kterém se má vyhledávání provést, dále pole ve kterém se kontroluje hledaná hodnota a hledaná hodnota samotná. Pro vyhledávání se využívá Elasticsearch Java Search API, jehož třídy slouží k přenosu dotazů na klastr, jejich zpracování a manipulaci výsledků. Nejjednodušším dotazem je vyhledávání záznamů podle hodnoty určitého pole. Nejdříve se vytvoří objekt třídy **SearchRequestBuilder**, který drží informace o připojení, typu dotazu a samotném poli s hledanou hodnotou. Typ vyhledávacího dotazu je nastaven pomocí statické metody poskytované třídou **QueryBuilders**. Objektu třídy **SearchRequestBuilder** lze nastavit mnoho dalších atributů před samotným vyhledáváním, například maximální počet prvků které má vrátit, dále lze přidat další agregační termy nebo další instance třídy **QueryBuilder**.

```
SearchRequestBuilder searchRequestBuilder = client.
    prepareSearch(query.getConnectionBO().getIndexName())
        .setTypes(query.getConnectionBO().getTypeName());

searchRequestBuilder.setQuery(QueryBuilders.matchQuery(
    query.getField(), query.getValue())).setSize(10000);

SearchResponse searchResponse = searchRequestBuilder.execute().
    actionGet();
```

Ukázka zdrojového kódu 2.4: Ukázka vytvoření stavitele pro vyhledávání

MatchQueryBuilder je základní typ vyhledávání, kdy je kontrolována hodnota definovaného pole. Pokročilejším typem dotazu jsou dotazy využívající třídu **QueryStringQueryBuilder**. Elasticsearch využívá Apache Lucene indexy a tyto indexy podporují vlastní syntax vyhledávacích dotazů.

Uživatel má tedy možnost napsat pokročilejší dotazy přímo do klientské aplikace, která text dotazu předá systémovému API a to jej zpracuje. O samotné zpracování se starají právě metody třídy **MatchQueryBuilder**. Při exekuci vyhledávacího dotazu dojde k jeho interpretaci na Lucene Query dotaz pomocí Query analyzátoru.

```
searchRequestBuilder.setQuery(queryStringQuery("Petr OR Pavel")
    .analyzeWildcard(true));
```

Ukázka zdrojového kódu 2.5: Ukázka fulltextového vyhledávání Elasticsearch Java API

Pro agregování vstupních dat je využita servisní třída **AggregationSearchService**. Tato třída vytváří jednotlivé dotazovací objekty, provolává klastr a zpracovává výsledky. Objekty systému dataWatcher nejsou kompatibilní s Elasticsearch Java API, proto vzniklo rozhraní **CustomAggregationBuilder** jako most. Rozhraní **CustomAggregationBuilder** zastřešuje skupiny agregačních objektů používaných v Elasticsearch Java API. Tato třída byla vytvořena pro jednodušší předání agregačních stavitelů servisním třídám. Každý agregační stavitel vytváří specifický dotaz na klastr a podle jeho typu jsou vráceny specifické výsledky. Agregační stavitel slouží také k nastavení detailů vyhledávání nad množinou dat. Mimo pole, nad kterým má proběhnout agregace, lze nastavit počet vrácených záznamů, minimální počet výskytů, či pořadí vrácených výsledků.

```
public AggregationBuilder getAggregationBuilder(AggregationBO
    aggregationBO) {
  return AggregationBuilders.terms(aggregationBO.getName())
                .field(aggregationBO.getField() + ".raw")
                .size(aggregationBO.getSetSize())
                .minDocCount(aggregationBO.getMinDocs())
                .order(Terms.Order.term(true));
}
```

Ukázka zdrojového kódu 2.6: Ukázka vytvoření agregačního dotazu Elasticsearch Java API z agregačního dotazu systému dataWatcher

Elasticsearch nabízí dvě skupiny agregačních stavitelů. První skupina jsou metrické agregace. Tyto agregace vrací statistické údaje nalezených dat v číselné podobě jako například maximální hodnotu, průměrnou hodnotu či součet polí. Druhou skupinou jsou **bucket** agregace. Výsledkem takovéto agregace je **bucket** obsahující jednotlivé možné hodnoty pole a počet jejich výskytu v hledané množině dat.

Z těchto dvou skupin agregačních dotazů byly pro účely systému zvoleny a implementovány tito metričtí agregační stavitelé:

- **AvgAggregationBuilder** vytvářející agregační dotaz pro průměrnou hodnotu v hledané množině.

- **PercentileAggregationBuilder** vytvářející agregační dotaz pro zjištění procentuálního výskytu jednotlivých hodnot určeného pole v hledané množině.

- **StatsAggregationBuilder** vytvářející agregační dotaz pro statistické údaje hodnot určeného pole v hledané množině.

Dále byly implementovány tito bucket agregační stavitelé:

- **MissingAggregationBuilder** vytvářející agregační dotaz vracející záznamy, které nemají hodnotu v hledaném poli v rámci hledané množiny.

- **TermAggregationBuilder** vytvářející agregační dotaz vracející standardní agregaci, tak jak je známa.

Každý agregační stavitel má metodu pro vyextrahování vracených výsledků Elasticsearch Java API do objektů systémového API dataWatcher. Pro přenos dat mezi systémovým API a klientskými aplikacemi vznikla transportní třída **ResultDTO** obsahující atributy pro přenos výsledků. Každý agregační stavitel namapuje vrácené výsledky, metrické či bucket agregace, na atributy třídy **ResultDTO**.

```
public List<ResultDTO> getResponse(SearchResponse response,
    AggregationBO aggregation) {
        List<ResultDTO> resultList = new ArrayList<>();

        Terms agg = response.getAggregations()
                .get(aggregation.getName());

        agg.getBuckets().forEach(entry -> {
            resultList.add(new ResultDTO(
                    aggregation.getName(),
                    entry.getKeyAsString(),
                    (double) entry.getDocCount())
            );
        });

        return resultList;
    }
```

Ukázka zdrojového kódu 2.7: Ukázka vyextrahování nalezených dat

Po vytvoření je stavitel předán jako atribut objektu třídy **SearchRequestBuilder**, který již obsahuje informace o klastru. Jedna instance třídy **SearchRequestBuilder** může nést několik agregačních a několik dotazovacích stavitelů. Díky tomu lze mezi jednotlivými agregacemi vytvářet vztah vnořených agregací, kdy se nejdříve data prohledají na základě prvního pole, a poté na základě dalších polí.

Jako příklad lze předpokládat, že každý záznam v určitém indexu obsahuje unikátní identifikátor uživatele a zároveň jeho polohu. Vytvoří-li administrátor

agregačního stavitele, který bude obsahovat agregaci právě na identifikátor uživatele a vnořenou agregaci na jeho momentální lokaci, vrácená data budou obsahovat bucket s jednotlivými uživateli a tento bucket budou obsahovat jejich jednotlivé polohy.

Pro vytváření vnořených agregačních stavitelů slouží metoda *searchMultipleAggregations* třídy **AggregationSearchService**. Tato metoda vezme rodičovský agregační term, prochází vnořené termy a na základě vztahu vytváří vnořené agregační termy či rovnou provádí vyhledávání na vstupních datech. Výsledky těchto operací jsou uloženy a přeneseny do klientské aplikace.

```
private List searchSingleAggregationBuilder(AggregationBuilder
    builder, TermBO term) {
 ConnectionBO connection = term.getAggregationBO().
     getConnectionBO();

 SearchResponse response = clientService.getClient()
                       .prepareSearch(connection.getIndexName())
                       .addAggregation(builder)
                       .execute().actionGet();
}
```

Ukázka zdrojového kódu 2.8: Ukázka exekuce vyhledávacího dotazu Elasticsearch Java API

2.3.4 Automatické spouštění akcí a generování událostí

Pro každou akci lze nastavit automatické spouštění, při kterém jsou prohledána data indexů určených ve vyhledávacích objektech. O spuštění akcí v definovaný čas se stará třída **ScheduledTasks**, která spouští jednotlivé úkoly každých n minut. Každý takto spuštěný úkol prohledá databázi a vybere akce, které se mají vykonat. Nalezne-li takovéto akce, provede vyhledání na daném indexu a v případě splnění podmínky nalezených dat vygeneruje událost. Události mohou být dvou typů. Prvním typem je vytvoření varovné zprávy, která se uloží do databáze a klientská aplikace ji zobrazí. Druhým typem události jsou e-mailové zprávy které jsou odesílány na administrátorský účet. Tento účet lze nastavit pomocí klientské aplikace.

Každá e-mailová zpráva je uložena do fronty **EmailQueue** a následně zavolána metoda *sendEmail* třídy **EmailSender**. Tato třída využívá metod knihovny **javax.mail**, která zpracovává e-mailové zprávy a zároveň je i odesílá.

2.4 API rozhraní

RESTful rozhraní systémového API slouží jako jediný přístupový bod klientských aplikací do systémového API. Každý kontroler obsahuje skupinu metod, které zpracovávají HTTP požadavky a vracejí požadovaná data doplněná o HTTP kódy. HTTP kódy, které systémové API vrací jsou rozdílné podle požadované operace a jejího výsledku. Návratové kódy jsou následující:

- 200 OK – tento návratový kód znamená, že požadovaná operace proběhla v pořádku.
- 201 CREATED – došlo k vytvoření požadovaného objektu.
- 400 BAD REQUEST – požadovaná operace nemůže být vykonána z důvodu nesprávné syntaxe požadavku.
- 404 NOT FOUND – požadovaná data nebyla nalezena.
- 409 CONFLICT – data nemohou být uložena z důvodu konfliktu s jinými daty.
- 500 INTERNAL SERVER ERROR – při zpracování požadavku došlo k chybě na straně serveru.

Rozhraní podporuje CRUD operace pro vytváření vyhledávacích objektů a akcí dále podporuje vyhledávání či zobrazení základních informací o Elasticsearch klastru. Úplný přehled rozhraní je následující:

- ActionEndpoint – tento kontroler zpracovává CRUD operace akcí.
- AggregationEndpoint – tento kontroler zpracovává CRUD operace agregací.
- ClusterHealthEndpoint – tento kontroler vrací základní informace o zdraví Elasticsearch klastru.
- ConnectionEndpoint – tento kontroler zpracovává CRUD operace připojených indexů Elasticsearch klastru.
- ErrorHandlerEndpoint – tento kontroler slouží k zpracování a zachycení neočekávaných chybových stavů systémového API.
- FieldsMappingEndpoint – tento kontroler slouží k získání mapování polí pro jednotlivé indexy Elasticsearch klastru.
- GeoLocationEndpoint – tento kontroler zpracovává požadavky geolokačních dotazů na jednotlivé indexy Elasticsearch klastru.
- JobEndpoint – tento kontroler zpracovává CRUD operace událostí.
- QueryEndpoint – tento kontroler zpracovává CRUD operace vyhledávacích dotazů.
- RootController – tento kontroler vrací instanci SwaggerUI pro testování a dokumentaci API.
- SearchEndpoint – tento kontroler obsluhuje vyhledávací požadavky vyhledávacích objektů.

- WarningEndpoint – tento kontroler zpracovává CRUD operace varovných zpráv.

Pro obsluhu HTTP požadavků využívají kontrolery anotace a metody knihovny HTTP Spring framework. Tento oblíbený framework nabízí knihovny jak pro vývoj RESTful aplikačních rozhraní tak i pro další návrhy jako například MVC architekturu. Každý kontroler má nastavenou URL, pokud je odeslán HTTP požadavek na tuto URL, je tento požadavek předán kontroleru ke zpracování. Ten na základě HTTP metody a případného rozšíření URL určí, jaká metoda bude tento požadavek obsluhovat.

```
@RestController
@RequestMapping("/api/v1/actions")
public class ActionEndpoint {

    ...

    @GetMapping
    public ResponseEntity<?> getAll() {

        List<ActionDTO> actions = actionService.findAll();
        if (actions != null) {
            return new ResponseEntity<>(actions, HttpStatus.OK);
        }

        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    @GetMapping("/{idAction}")
    public ResponseEntity<?> getById(@PathVariable int idAction)
        {

        ActionBO action = actionService.findByIdAction(idAction);
        if (action != null) {
            return new ResponseEntity<>(new ActionDTO(action),
                HttpStatus.OK);
        }

        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Ukázka zdrojového kódu 2.9: Ukázka kontroleru RESTful API

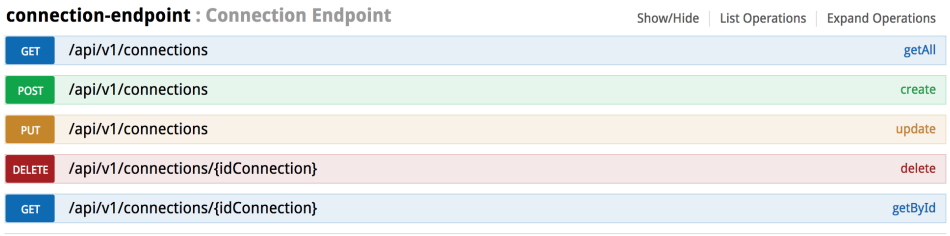
Při zpracování požadavků se může vyskytnout výjimka, na kterou není program připraven. Pokud by tato výjimka nebyla zachycena, zobrazil by se její popis, nazývaný stacktrace, uživateli. Toto chování systémového API samozřejmě není přípustné, proto je vytvořen **ErrorHandlerEndpoint**, který zachytává všechny výjimky typu **Exception** a klientské aplikaci vrátí pouze chybový kód.

```
@ControllerAdvice
public class ErrorHandlerEndpoint {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<?> globalError(Exception e) {
        return new ResponseEntity<>("Something went wrong",
        HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Ukázka zdrojového kódu 2.10: Ukázka kontroleru zpracovávajícího nezachycené výjimky systému

2.4.1 SwaggerUI – automatická dokumentace API

SwaggerUI je jednoduchý nástroj sloužící k vytvoření automatické dokumentace a testování RESTful rozhraní. Swagger automaticky skenuje aplikaci a vyhledává všechny kontrolery a na základě jejich nastavení vytvoří přístupový bod, kde je seznam všech kontrolerů, jejich operací a návratových hodnot. Dále nabízí možnost okamžitého provolání jednotlivých metod kontrolerů a jejich testování.



Obrázek 2.4: Ukázka rozhraní testovacího nástroje SwaggerUI

2.5 Realizace klientské aplikace

Pro vývoj frontendové aplikace byl využit JavaScriptový framework AngularJS 2. Tento framework nabízí kompletní řešení pro vývoj MVC aplikací běžících ve webovém prohlížeči.

2.5.1 Moduly a navigace uvnitř aplikace

Aplikace se skládá z modulů, které mezi sebou komunikují a navzájem sdílejí data. Každý modul aplikace má své využití a může se skládat z dalších modulů. Aplikace se skládá z hlavního *root* modulu, který slouží ke spojení ostatních modulů a k navigaci mezi nimi. Pro každou logickou část aplikace byl vytvořen modul, v jehož rámci probíhá vlastní navigace.

Každý modul obsahuje jednu či více servisních tříd, které slouží k odesílání a zpracování HTTP požadavků na systémové API a k následnému zpracování dat. Jednotlivé servisní třídy vytvářejí URL dotazu a provolávají metody třídy **SharedService**, která provede samotné vytvoření HTTP požadavku a předá vrácená data volající servisní třídě.

```
@Injectable()
export class QueryService {

   constructor(private sharedService: SharedService) {}

 public getAll(): Observable<Query[]> {
     return this.sharedService.getAll(API_QUERIES_ENDPOINT);
 }
}
```

Ukázka zdrojového kódu 2.11: Ukázka servisní třídy volající HTTP servisní třídy klientské aplikace

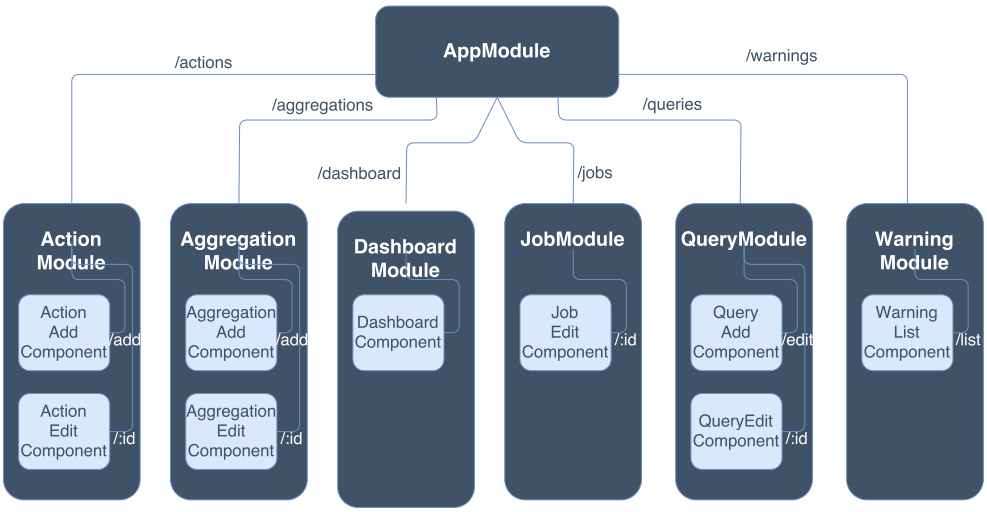
```
@Injectable()
export class SharedService extends AbstractHttpService {

   constructor(private http: Http) {
       super();
   }

   public getAll(endUrl: string): Observable<[any]> {
       return this.http.get(API_ROOT_DOMAIN + endUrl).map(
           (response: any) => response.json()
       ).catch(this.handleError);
   }
}
```

Ukázka zdrojového kódu 2.12: Ukázka servisní třídy zpracovávající HTTP požadavky

Jednotlivé moduly jsou následující:

- AppModule,
- ActionModule,
- AggregationModule,
- DashboardModule,
- JobModule,
- QueryModule,
- WarningModule,
- SharedModule.

Navigace mezi jednotlivými moduly je zajištěna díky třídě **AppRoutingModule**. Tato třída obsahuje konfigurační pole **Routes** jednotlivých URL aplikace, na jejichž základě proběhne navigace do určitého modulu. Každý další modul má opět svoji navigační třídu, která opět obsahuje konfigurační pole navigačních URL na jednotlivé komponenty či vnořené moduly.



Obrázek 2.5: Schéma modulů a hlavních komponent klientské aplikace

2.5.1.1 AppModule

Modul **AppModule** obsahuje základní navigaci, spojuje jednotlivé moduly a skládá se z jediné komponenty. Tato komponenta obsahuje navigační menu aplikace a informace o připojeném klastru. Každá AngularJS 2 aplikace je *single page* aplikace a stránky se skládají z jednotlivých komponent, které jsou dynamicky vkládány podle aktuální potřeby.

Každá AngularJS 2 aplikace obsahuje hlavní modul podle konvence pojmenovaný *app.routing.module*[22], ve kterém je základní navigace do dalších vnořených modulů. Vnořené moduly mohou mít své *routing.module* moduly, kde jsou nastavené jednotlivé cesty a komponenty, které mají být načteny.

```
const routes: Routes = [
    {path: '', redirectTo: '/dashboard', pathMatch: 'full'},
    {path: 'queries', loadChildren: 'app/dataWatcher/query/query.
        module#QueryModule'},
    {path: '**', component: ErrorNotFoundComponent},
];

@NgModule({
    imports: [
```

```
        RouterModule.forRoot(routes),
    ],
    exports: [
        RouterModule
    ],
})
export class AppRoutingModule {

}
```

Ukázka zdrojového kódu 2.13: Ukázka navigačního modulu aplikace

2.5.1.2 DashboardModule

DashboardModule slouží jako základní rozhraní, které se zobrazí po příchodu do aplikace. Obsahuje popis všech akcí, agregací a vyhledávacích dotazů, skrz které je možno přejít na detail daného objektu či vytvořit nový. Dále jsou zde zobrazeny základní informace o klastru a o varovných informacích.

2.5.1.3 ActionModule

Modul **ActionModule** slouží k vytvoření a editaci akcí, které budou následně volány na systémovém API a generovat události. Tento modul se skládá z několika komponent. Nejnadřazenějšími komponentami jsou **ActionDetailEditComponent** a **ActionDetailAddComponent**. Tyto komponenty slouží k editaci a vytváření akcí. Komponenty se dále skládají z vnořených komponent, které zobrazují jednotlivé detaily akcí, či agregace a vyhledávací dotazy, které lze přiřadit dané akci.

K zobrazení vnořené komponenty v nadřazené komponentě slouží direktivy. Každá komponenta má přiřazenu unikátní direktivu v rámci celé aplikace a tato direktiva slouží k umístění komponenty v rámci jiné komponenty. **ActionModule** obsahuje mimo jiné komponentu **TermListComponent**, která zobrazuje jednotlivé agregační termy a umožňuje vytvářet vztah mezi agregacemi v rámci jedné akce.

```
@Component({
    selector: 'term-list',
    templateUrl: './term-list.component.html',
    styleUrls: ['./term-list.component.css']
})
export class TermListComponent {
    @Input()
    private action: Action;

    //zdrojovy kod komponenty (typescript)
}
```

Ukázka zdrojového kódu 2.14: Ukázka zdrojového kódu komponenty

Komponenta **TermListComponent** má nastavený selektor *term-list*, díky kterému jí lze zobrazit kdekoliv v rámci modulu **ActionModule**, nebo je-li z tohoto modulu vyexportována do dalších modulů i mimo něj. Jako vstupní parametr této komponenty slouží objekt třídy **Action**, která obsahuje veškerá data, která tato komponenta zpracovává a zobrazuje. Pro zobrazení komponenty **TermListComponent** uvnitř jiné komponenty, stačí přidat její direktivu do HTML šablony komponenty a předat příslušný objekt.

```
<form>
   <div class="container" *ngIf="action">
      <term-list [action]="action"></term-list>
      <!-- dalsi kod komponenty (HTML) -->
   </div>
</form>
```

Ukázka zdrojového kódu 2.15: Ukázka vložení komponenty do jiné komponenty

2.5.1.4 AggregationModule, QueryModule a JobModule

Tyto moduly obsahují logicky podobnou množinu komponent. Moduly obsahují komponenty pro vytvoření a editaci vyhledávacích objektů či událostí a komponenty pro jejich zobrazení v jiných modulech.

2.5.1.5 WarningModule

Tento modul obsahuje komponentu pro zobrazení a zpracování varovných zpráv v klientské aplikaci.

2.5.1.6 SharedModule

Modul **SharedModule** obsahuje servisní třídy a modely, které jsou použity napříč různými moduly. Po editaci či vytvoření vyhledávacích objektů je odeslán HTTP požadavek na systémové API žádající vyhledání na základě těchto objektů. Výsledky vrácené z API se zpracují a servisní třída **SearchService** vrátí seznam nalezených výsledků. Tyto výsledky jednotlivé komponenty zpracují a je-li to možné zobrazí jejich graf či mapu umístění. Pro vytvoření grafu je připravena servisní třída **ChartConfiguratorService**, která přijme na vstupu kolekci výsledků, zpracuje jej a vrátí připravený graf pro zobrazení.

Některé nalezené záznamy jsou příliš dlouhé na zobrazení v jednom řádku. Pro zkrácení takových to záznamů byla implementována roura neboli *pipeline*, která na vstupu dostane dlouhý řetězec, ořízne jej podle nastavené délky a přidá znak ... značící zkrácení řádku.

```
@Pipe({name: 'myTruncate'})
export class TruncatePipe implements PipeTransform {
    public transform(value: string,limit: number): string {
```

```
        return value.length > limit ? value.substring(0,limit) +
            '...' : value;
    }
}
```

Ukázka zdrojového kódu 2.16: Ukázka modifikující roury

Takovouto rouru lze pak v aplikaci využít

```
<div *ngFor="let query of results" class="content">
  <div class="message">{{query.key | myTruncate : 200}}</div>
</div>
```

Ukázka zdrojového kódu 2.17: Ukázka užití modifikující roury

2.6 Závěr

Při implementaci systému bylo vyvinuto systémové RESTful Java API umožňující práci s Elasticsearch klastrem a AngularJS 2 aplikace, která spravuje požadavky uživatele a předává je API. Při vývoji byla snaha o co nejčistší, sebe popisující kód, který je připravený na další rozšíření. Nejdůležitější a nejsložitější části zdrojového kódu jsou okomentované a kód je rozumně členěn do adresářové struktury.

Kapitola 3

Testování

Tato kapitola se zabývá automatickým a manuálním testováním systému a jeho jednotlivých komponent. Nejdříve jsou popsány strategie testování, které byly použity, dále jsou popsány automatické testy a průběh manuálního testování. Výsledky testování jsou diskutovány v sekci Závěr.

3.1 Vybrané strategie testování systému

3.1.1 Smoke testy

Smoke testování[23] slouží k rychlému ověření, zda jsou všechny části systému vyvinuty a plně funkční. Toto testování se provádí u těch částí systému, které by se při dalším vývoji již neměly měnit. Proto jsou tyto testy i vhodné jako základní test stálosti jednotlivých částí systému. Tyto testy jsou automatizované a měly by být spouštěny po každém přidání či úpravě funkčnosti systému.

3.1.2 Funkční a nefunkční testy

3.1.2.1 Funkční testy

Při funkčním testování probíhá kontrola, zda jsou veškeré funkcionality správně vyvinuty. Tato kategorie testů je nejrozsáhlejší. U velkých systémů dochází k částečné automatizaci funkčních testů pro neměnné části.

3.1.2.2 Nefunkční testy

Nefunkční testování nekontroluje správnost funkcionalit systému, ale chování systému jako celku. Při nefunkčním testování je kontrolován výkon systému, jeho obnova po pádu[11], či zátěž sítě při zpracovávání požadavků uživatelů.

[11] takzvané disaster testy

3.1.3 Integrační testy

Při integračních testech je testována komunikace mezi jednotlivými komponentami systému a zároveň i komunikace jednotlivých částí komponent[12].

3.2 Automatické testování API systému

Během vývoje systému byla vytvořena množina automatických testů, pozitivních i negativních, které pokrývají důležité části jednotlivých komponent. Automatické testování systémového API probíhá pomocí Unit a integračních testů. Spring Boot aplikace jsou na automatické testovaní připraveny. Pro vývoj testů je připraven startovací balíček pokrývající většinu knihoven, které jsou pro vývoj testů potřeba. Konfigurace jednotlivých testů probíhá skrze anotace. Anotace *@SpringBootTest* konfiguruje testovací třídu, nalezne konfiguraci aplikace či nastaví potřebné porty. Další používanou anotací je *@RunWith*, která určuje třídy ve které test poběží. Pokud není tato anotace využita, jako běhové prostředí je použito běhové prostředí JUnit.

Každý test obsahuje několik bloků kódu, které ovlivňují jeho průběh. Nejdříve dojde k inicializaci potřebných testovacích objektů v sekci *@Before*. Poté dojde k samotnému testu a následné destrukci dat v sekci *@After*. Ukázkový test pokrývá životní cyklus agregace, ovládaný skrze kontroler pro správu agregací. Dochází tedy ke skutečnému odeslání HTTP požadavků na RESTful API.

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class, webEnvironment =
    SpringBootTest.WebEnvironment.RANDOM_PORT)
public class AggregationEndpointTest {
  @Before
  public void init() {
    ConnectionBO connectionBO = new connectionBO();
    this.aggregationBO = new AggregationBO(connectionBO,"name","
        type","field",1,1,1,true);
  }
  @After
  public void deleteAggregation() {
    aggregationDAO.delete(this.aggregationBO);
  }

  @Test
  public void crudOperationTest() {
    HttpEntity<AggregationBO> entity = new HttpEntity<
        AggregationBO>(this.aggregationBO);
    ResponseEntity<AggregationBO> response = restTemplate.
        postForEntity("/api/v1/aggregations",entity,
        AggregationBO.class);
```

[12] například komunikace servisní vrstvy s datovou

```
        Assert.assertEquals(HttpStatus.CREATED,response.
            getStatusCode());
        Assert.assertEquals(response.getBody(),this.aggregationBO);
    }
}
```

Ukázka zdrojového kódu 3.1: Ukázka zdrojového kódu automatického testu systémového API

Jednotlivé testy, ať jednotkové či integrační, probíhají nad testovací množinou dat, kde lze jednoznačně předvídat výsledek testovaných operací. Automatizované testy kontrolují tuto množinu operací systémového API.

- Životní cyklus akcí (vytvoření, získání detailů, editace, smazání, nenalezení po smazání).
- Životní cyklus agregací (vytvoření, získání detailů, editace, smazání, nenalezení po smazání).
- Životní cyklus vyhledávacích dotazů (vytvoření, získání detailů, editace, smazání, nenalezení po smazání).
- Vytvoření vztahu agregací a vyhledání dat pro tento vyhledávací term (Smoke test).
- Vytvoření samostatné agregace a vyhledání dat (Smoke test).
- Vytvoření akce společně se vztahem agregací a následné vyhledání dat na základě této akce (Smoke test).
- Vytvoření vyhledávacího dotazu a vyhledání dat (Smoke test).
- Testování jednotlivých kontrolerů a jejich odpovědí (Smoke test).

3.3 Manuální testování a nefunkční testování

3.3.1 Manuální testování

Manuální testování probíhalo podle definovaných testovacích scénářů i zároveň jako volné testování bez scénáře. Systém byl otestován několika uživateli, jak na připravených testovacích datech, tak i na anonymizované kopii produkčních dat. Během testů byli prověřeny pozitivní i negativní scénáře. Testování probíhalo na skupině uživatelů, kteří byli předem seznámeni s ideou systému a s jeho základním ovládáním a strukturou. Mezi testery se zapojili, jak technicky vzdělaní uživatelé, kteří během testů zachytávali požadavky a kontrolovali, je dále kontrolovali i záznamy v databázích a v indexech nástroje Elasticsearch, tak i standardní uživatelé, kteří testovali pouze odpovědi klientské aplikace. Během testování bylo nalezeno několik chyb, které byly reportovány a následně opraveny.

3.3.1.1 Testovací scénáře

Pro účely manuálního testování vzniklo několik různých sad testovacích scénářů. Každá sada se skládá z jednotlivých testů zabývající se specifickou částí systému. Ukázkový testovací scénář je následující.

Krok: Uživatel přejde na adresu aplikace dataWatcher.
Očekávaná odpověď: Bude zobrazena stránka Dashboard s informacemi o akcích, agregacích, vyhledávacích dotazech a připojeném klastru.
Krok: Uživatel klikne na tlačítko "+"v sekci agregací.
Očekávaná odpověď: Bude zobrazen formulář pro vytvoření nové agregace.
Krok: Uživatel klikne na tlačítko uložit.
Očekávaná odpověď: Zobrazí se chybová hláška o povinnosti vyplnění jména.
Krok: Uživatel vyplní jméno nové agregace delší 100 znaků.
Očekávaná odpověď: Nelze zadat více než 100 znaků.
Krok: Uživatel vyplní jméno nové agregace.
Očekávaná odpověď: Jméno zadáno.
Krok: Uživatel vybere připojení.
Očekávaná odpověď: Na základě připojení se změní mapovaná pole.
Krok: Uživatel vybere geolokační pole připojení.
Očekávaná odpověď: Pole vybráno.
Krok: Uživatel klikne na tlačítko uložit.
Očekávaná odpověď: Zobrazí se chybová hláška o povinnosti minimálního počtu záznamů.
Krok: Uživatel vyplní minimální počet záznamů textovou hodnotu.
Očekávaná odpověď: Textová hodnota nelze vložit.
Krok: Uživatel vyplní minimální počet záznamů numerickou hodnotou.
Očekávaná odpověď: Pole vyplněno.
Krok: Uživatel klikne na tlačítko uložit.
Očekávaná odpověď: Zobrazí se chybová hláška o povinnosti vyplnění velikosti.
Krok: Uživatel vyplní velikost textovou hodnotu.
Očekávaná odpověď: Textová hodnota nelze vložit.
Krok: Uživatel vyplní velikost numerickou hodnotou.
Očekávaná odpověď: Pole vyplněno.
Krok: Uživatel klikne na tlačítko uložit.
Očekávaná odpověď: Zobrazí se chybová hláška o povinnosti vyplnění historie.
Krok: Uživatel vybere jak hluboko do historie vyhledávat.
Očekávaná odpověď: Pole vybráno.
Krok: Uživatel klikne na tlačítko uložit.
Očekávaná odpověď: Zobrazí se chybová hláška o povinnosti vyplnění typu agregace.
Krok: Uživatel vybere typ agregace.
Očekávaná odpověď: Pole vybráno.
Krok: Uživatel klikne na tlačítko uložit.
Očekávaná odpověď: Aplikace je přesměrována na detail nové agregace.
Krok: Kontrola správnosti uložených dat.

Očekávaná odpověď: Data odpovídají údajům vloženým v předchozích krocích.
Krok: Kontrola zobrazení grafu a jeho legendy.
Očekávaná odpověď: Graf i legenda jsou zobrazeny a data odpovídají datům v klastru.
Krok: Kontrola zobrazení mapy
Očekávaná odpověď: Mapa zobrazena a data odpovídají datům klastru.

3.3.1.2 Integrační testy

Během integračních testů byly kontrolovány veškeré zprávy, které byly odeslány jednotlivými komponentami systému a to včetně e-mailových upozornění a jejich obsahu. Integrační testování probíhalo na dvou testovacích sestaveních. První sestavení obsahovalo kontejner systémového API, databázového serveru a kontejner s klientskou aplikaci na jednom serveru, kontejner pro Elasticsearch a Logstash běžely na odděleném serveru. Druhé sestavení se skládalo opět ze dvou serverů, kde na jednom serveru běžely kontejnery se systémovým API, databázovým serverem, klientskou aplikací a nástrojem Elasticsearch na druhém serveru běžel Logstash, který zasílal data do nástroje Elasticsearch. Každá komponenta systému běžela v samostatném kontejneru.

3.3.2 Testování obnovy

Důležitou součástí testovaní byly testy znovuobnovení systémů takzvané disaster testy. Během těchto testů byly některé komponenty systému vypnuty a následně znovu spuštěny. Během těchto testů bylo sledováno opětovné uvedení komponent do běžícího stavu a náprava celého systému. Nejdříve byly vypnuty tyto komponenty:

- databázový server,
- Elasticsearch klastr,
- systémové API.

Průběh testu odpovídal předpokladům, že při vypnutí jedné komponenty sytému dojde k omezenému využití funkčnosti ostatních komponent, ale nedojde k jejich pádu. Po obnovení běhu vypnuté komponenty nebylo nutné zbylé části systému restartovat díky automatickému znovu – připojení či bezstavovosti komponent. V produkčním nasazení je však potřeba zajistit vysoká dostupnost systému. Proto by měl být nad jednotlivé kontejnery postaven orchestrátor, který by zajistil vyvažování zátěže, a zároveň by vytvářel nové kontejnery při nedostupnosti či vypínal nepotřebné.

3.4 Závěr

Během testování bylo nalezeno několik menších chyb, které byly ihned opraveny. Zároveň se během testování ukázalo, že data vrácená při vyhledávání vnořených agregací, nejsou pro uživatele vždy čitelná. Tento designový nedostatek byl zaznamenán a bylo rozhodnuto, že bude odstraněn v rámci budoucího vývoje.

Kapitola 4

Nasazení

4.1 Docker

Nasazení systému v testovacím, či v produkčním prostředí, je díky kontejnerizaci jednotlivých vrstev jednodušší, než při standardním nasazení na aplikační servery. Každý kontejner má vytvořený Dockerfile, ve kterém jsou v jednotlivých vrstvách uvedeny kroky potřebné pro správné nasazení kontejneru.

```
FROM java:8-jdk

EXPOSE 8090


VOLUME /tmp
COPY dataWatcher.jar /dataWatcher.jar
ENTRYPOINT ["java","-jar","/dataWatcher.jar"]
```

Ukázka zdrojového kódu 4.1: Ukázka Docker build souboru

V první vrstvě je určen bázový obraz kontejneru. V bázovém obrazu je nastaveno základní běhové prostředí pro proces v kontejneru. Dále dojde k vystavení portu 8090 mimo kontejner tak, aby byl viditelný i pro hostitelský operační systém, na kterém kontejner běží. V další vrstvě dojde k vytvoření adresáře tmp, do kterého je následně překopírován .jar archiv se systémovým API. V posledním kroku dojde ke startu samotného API.

V rámci systému je kontejner se systémovým API sám o sobě nepoužitelný. Pro správný běh je potřeba kontejner propojit s databázovým serverem a Elasticsearch klastrem. Tyto kontejnery mají rovněž vytvořený Dockerfile s popisem nasazení. Pokud chce administrátor nasadit více kontejnerů na jeden fyzický server, lze kromě orchestrátoru využít i nástroj docker-compose. Docker-compose je nástroj pro administraci několika kontejnerů z různých obrazů. Administrace startu kontejnerů probíhá pomocí docker-compose souboru kde jsou uvedeny a nastaveny jednotlivé služby – kontejnery.

```
version: '2'
services:

  dw-backend:
    build: ./backend
    container_name: datawatcher-backend
    ports:
      - 8090:8090
    volumes:
      - '../dataWatcher:/dataWatcher'
    links:
      - dw-db
    depends_on:
      - dw-db

  dw-db:
    build: ./db
    container_name: datawatcher-db
    ports:
      - 3306:3306
```

Ukázka zdrojového kódu 4.2: Ukázka docker-compose build souboru

V tomto docker-compose souboru dojde nejprve k vytvoření kontejneru s databázovým serverem a k vystavení portu 3306 mimo kontejner. Následně je vytvořen kontejner se systémovým API, které připojuje na databázový kontejner a získává z něj data. Nástroj docker-compose vystaví kontejner pod názvem služby uvedeným v docker-compose souboru.

Kapitola 5

Budoucnost systému a další vývoj

Veškeré plánované funkčnosti byly implementovány a otestovány. Nicméně je již naplánován budoucí vývoj pro zvýšení možností využití systému a probíhá studium využitelnosti systému, jakožto možnost náhrady některých SIEM aplikací. Pro toto využití je potřeba rozšířit možnost vizualizace dat v reálném čase, vyvinout zabezpečení aplikace tak jak je popsáno v sekci 1.6 a rozšířit možnosti agregace o další typy.

Důležitou kapitolou budoucího vývoje je možnost označení vyřešených událostí a zamezení systému generování varovných zpráv pro tyto události. Během testování se tento nedostatek projevil jako obtěžující, při práci se systémem.

V tuto chvíli také probíhá integrace systému do nástroje Kibana. Tento nástroj podporuje široké spektrum možností vizualizace dat v reálném čase, dále nabízí pokročilé možnosti zabezpečení a omezení uživatelského přístupu na základě rolí díky rozšíření X-Pack.

V rámci dalšího vývoje kontejnerizace komponent systému, je naplánováno využití orchestrátoru Openshift pro nasazení systému v produkčním prostředí.

Jistou možností je vývoj mobilní klientské aplikace, se znovupoužitím zdrojových kódů aktuální frontendové aplikace. Jednalo by se tedy rozšíření klientské aplikace o šablony sloužící k zobrazení dat na mobilních telefonech a její migrace do nativního formátu pro iOS a Android[13].

[13]Za použití platforem Appache Cordova nebo NativeScript

Závěr

V rámci této práce proběhl výzkum systému pro získávání informací jak z logových souborů tak i jiných zdrojů dat. Na základě tohoto výzkumu byl vyvinut vyhledávací a agregační systém, který dokáže v reálném čase prohledávat vstupní data a reagovat na jejich změny generováním událostí.

V teoretické části jsou popsány použité technologie a architektura systému. Při návrhu se autor snažil co nejvíce rozložit systém na funkční komponenty, které půjde jednoduše škálovat a budou mít jednoduchou a samostatnou administraci. V sekci požadavků na systém jsou rozepsány jednotlivé požadavky, které vznikly jako výsledek studia dané problematiky.

Praktická část je rozdělena do tří částí, **Realizace, Testování a Nasazení**. V sekci realizace jsou popsané jednotlivé komponenty a proces jejich vývoje. Další část se zabývá testováním automatickým i manuálním, během kterého byly nalezeny některé chyby a zároveň proběhl výzkum dalšího rozšíření systému. Sekce Nasazení je zaměřena na proces nasazení systému rozděleného na jednotlivé kontejnery v rámci Docker prostředí. V této sekci jsou ukázky Docker build souborů pro jednotlivé komponenty systému, tak i pro celý systém jako celek.

V poslední kapitole došlo ke zhodnocení budoucího vývoje systému. Studium rozšíření probíhalo jak při testování systému, tak samostatně z potenciálních potřeb zákazníků.

Největším přínosem pro autora je studium platformy AngularJS 2, která přináší moderní přístup k vývoji MVC aplikací běžících v internetovém prohlížeči, ale i mimo něj a seznámení se s indexovacími nástroji jako je Elasticsearch.

Literatura

[1] Logstash Reference. [online], [cit 11-03-2017]. Dostupné z: https://www.elastic.co/guide/en/logstash/5.1/index.html

[2] Beats Platform Reference. [online], [cit 8-03-2017]. Dostupné z: https://www.elastic.co/guide/en/beats/libbeat/5.1/index.htmll

[3] Gheorghe, R.; Hinman, M. L.; Russo, R.: *Elasticsearch in action*. Shelter Island, NY: Manning Publications Co, první vydání, 2016, ISBN 978-1617291623.

[4] Elasticsearch Reference. [online], [cit 1-03-2017]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/5.1/index.html

[5] Kuc, R.; Rogozinski, M.: *Spring Data Standard Guide*. Birmingham: Packt Publishing, první vydání, 2013, ISBN 978-1783281435.

[6] Elasticsearch Reference - Inverted Index. [online], [cit 16-03-2017]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/guide/current/inverted-index.html

[7] Elasticsearch Watcher. [online], [cit 12-10-2016]. Dostupné z: https://www.elastic.co/guide/en/watcher/2.4/introduction.html

[8] Pečinovský, R.: *Java 8*. Praha: Grada, první vydání, 2014, ISBN 978-8024746388.

[9] Rod, J.: *Professional Java development with the Spring Framework*. Indianapolis, Ind: Wiley, první vydání, 2005, ISBN 978-1617291623.

[10] Spring Boot Reference Guide. [online], [cit 25-03-2017]. Dostupné z: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/

[11] Java Tools and Technologies Landscape Report 2016. [online], [cit 19-03-2017]. Dostupné z: https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/

[12] Kainulainen, P.: *Spring Data Standard Guide*. Birmingham: Packt Publishing, první vydání, 2012, ISBN 978-1849519045.

[13] Masse, M.: *REST API design rulebook*. Farnham: O'Reilly, první vydání, 2012, ISBN 978-1449310509.

[14] Architecture Overwiew. [online], [cit 10-03-2017]. Dostupné z: https://angular.io/docs/ts/latest/guide/architecture.html

[15] Coury, F.; Lerner, A.; Murray, N.; aj.: *ng-book 2*. San Francisco: Fullstack.io, první vydání, 2016, ISBN 978-0991344611.

[16] NgModule. [online], [cit 26-03-2017]. Dostupné z: https://angular.io/docs/ts/latest/guide/ngmodule.html

[17] Miell, I.; Sayers, A. H.: *Docker in Practice*. Shelter Island, NY: Manning Publications Co, první vydání, 2016, ISBN 978-1617292729.

[18] Docker overview. [online], [cit 22-03-2017]. Dostupné z: https://docs.docker.com/engine/docker-overview/

[19] Containers 101: Linux containers and Docker explained. [online], [cit 11-03-2017]. Dostupné z: http://www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html

[20] How Security Works. [online], [cit 15-10-2016]. Dostupné z: https://www.elastic.co/guide/en/x-pack/current/how-security-works.html

[21] How Security Works. [online], [cit 22-10-2016]. Dostupné z: https://www.elastic.co/guide/en/x-pack/current/custom-realms.html

[22] Routing & Navigation. [online], [cit 25-03-2017]. Dostupné z: https://angular.io/docs/ts/latest/guide/router.html

[23] Ali, M.; Fairouz, T.: *Software testing*. Indianapolis, Ind: Wiley, první vydání, 2015, ISBN 978-1118662878.

Příloha **A**

Instalační manuál

Jednotlivé komponenty systému běží jako uzavřený proces uvnitř Docker kontejnerů. Pro jejich běh je tedy potřeba nainstalovat nástroje Docker a Docker-compose do hostitelského systému. Konfigurace jednotlivých obrazů je v adresáři **exe/docker**[14], ze kterého proběhne i jejich start. Pro spuštění je tedy potřeba přejít do tohoto adresáře a zadat příkaz:

```
$docker-compose up
```

První spuštění kontejnerů trvá déle, protože dochází ke stažení všech bázových obrazů, knihoven a dalších závislostí, které jsou potřeba ke správnému běhu celého systému.

Po startu systému dojde k vystavení klientské aplikace na port 3000. Aplikace je pak přístupná na URL *http://localhost:3000/*

Pro testování aplikace naběhne v demo režimu, ve kterém jsou připravena ukázková data a vyhledávací objekty.

Vypnutí systému se provádí stiskem Ctrl+C nebo zadáním příkazu:

```
$docker-compose stop
```

Tímto příkazem dojde k vypnutí běžících kontejnerů, jejich obrazy a sestavení však budou nadále v hostitelském systému. Pro jejich odstranění je potřeba získat seznam jejich identifikátorů. Seznam všech sestavení se zobrazí po zadání příkazu:

```
$docker ps -a
```

Pro odstraněn kontejnerů slouží příkaz:

```
$docker rm <identifikator_kontejenru>
```

Pro smazání a bázových obrazů kontejnerů slouží příkaz:

```
$docker rmi <identifikator_obrazu>
```

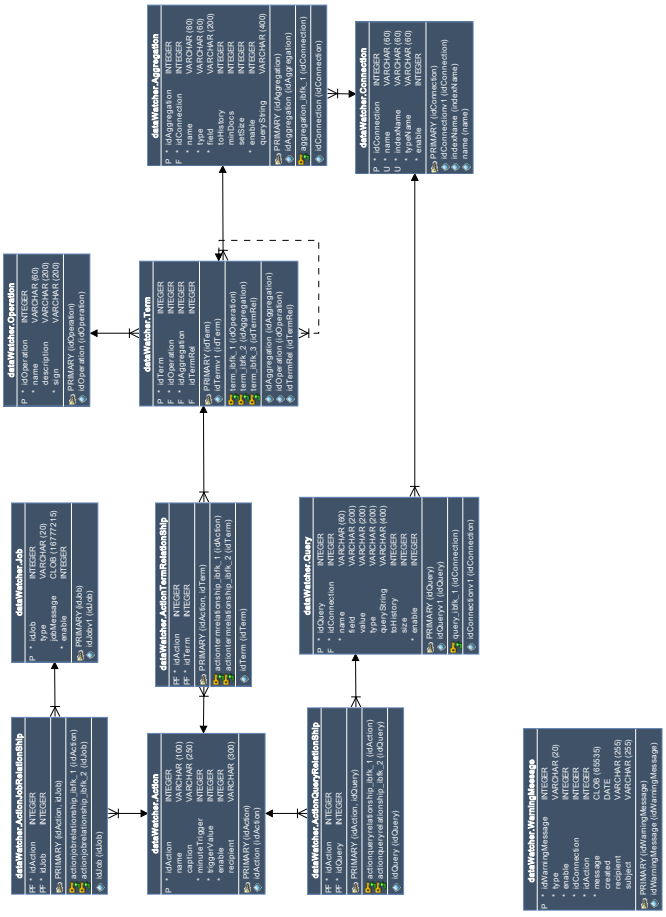
Identifikátory obrazů se zobrazí po zadání příkazu:

[14]Pro rychlejší start kontejnerů, je doporučeno přesunout adresář **exe** se všemi podadresáři z média do hostitelského systému.

```
$docker images
```

Příloha B

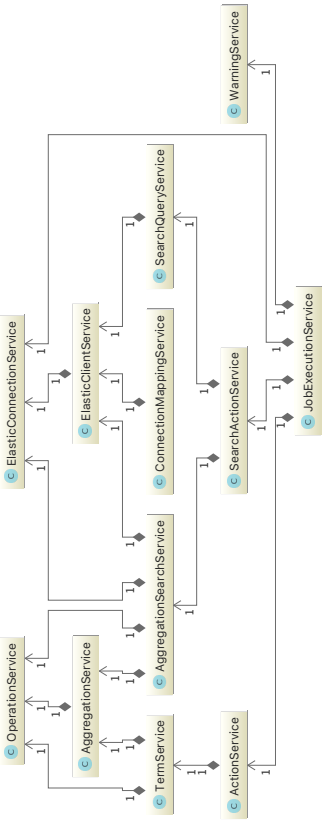
Relační databázový model



Obrázek B.1: Relační databázový model

Příloha C

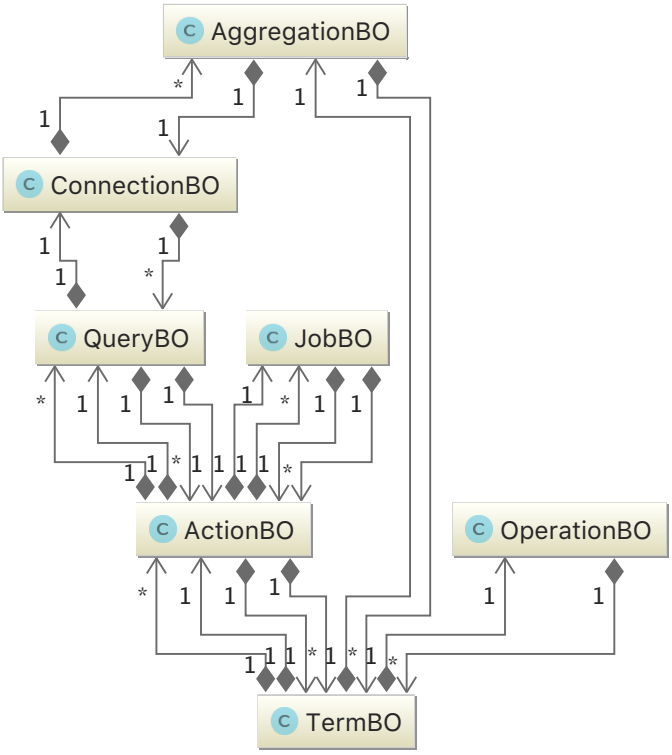
Třídní model servisní vrstvy systémového API



Obrázek C.1: Třídní model servisní vrstvy systémového API

Příloha D

Třídní model DAO vrstvy systémového API



Obrázek D.1: Třídní model DAO vrstvy systémového API

Příloha **E**

Seznam použitých zkratek

API Application Interface

CRUD Create, Read, Update, Delete

DB Database

GUI Graphical user interface

HATEOAS Hypermedia as the Engine of Application State

HTTP Hypertext Transfer Protocol

IP Internet Protocol

JSON JavaScript Object Notation

MVC Model View Controller

POJO Plain Old Java Object

REST Representational State Transfer

SQL Structured Query Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

XML Extensible markup language

Příloha F

Seznam použitých pojmů

Bucket je množina nalezených záznamů, které pojí určitý atribut.

Builder je abstraktní návrhový vzor popisující stavbu instancí různých tříd za využití *get* a *set* metod.

Framework je množina nástrojů knihoven a konvencí, které mají za cíl odstínit rutinní a základní problémy do modulů, které lze později znovu využívat.

Load balancing neboli vyvažování zátěže je technika rozdělení zátěže mezi několik uzlů, počítačů či jiných zařízení.

Load balancer hardwarové zařízení či softwarový program starající se o load balancing.

Pipeline je funkcionalita platformy AngularJS 2, která umožňuje transformaci objektů předávaných komponentám. tato transformace je iniciována přímo z *HTML* kódu.

RESTful API je rozhraní navržené podle pravidel REST architektury.

Elasticsearch shard reprezentuje jeden Apache Lucene index v Elasticsearch klastru.

Single page aplikace je aplikace, která je uživateli předána jako v rámci jediného *HTML* souboru obohaceného o JavaScriptové skripty.

Příloha G

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `server` ..... zdrojové kódy systémového API
    - `client` ..... zdrojové kódy klientské aplikace
  - `thesis` ..... zdrojové kódy práce
    - `thesis.tex` ..... zdrojová forma práce ve formátu LaTeX
    - `src` ..... obrázky a přílohy práce
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF