

**Vysoká škola ekonomická v Praze**

**Fakulta informatiky a statistiky**

**Katedra informačních technologií**

Študijný program: Aplikovaná informatika

Obor: Informační systémy a technologie

## **Esper Web**

# **Webové rozhranie pre spracovanie udalostí v reálnom čase**

**DIPLOMOVÁ PRÁCA**

Študent : Bc. Martin Kravec

Vedúci : Mgr. Zbyněk Šlajchrt

Oponent : Ing. Rudolf Pecinovský, CSc.

**2015**

# Prehlásenie

Prehlasujem, že som diplomovú prácu spracoval samostatne, a že som uviedol všetky použité pramene a literatúru, z ktorej som čerpal.

# Podakovanie

Chcem poďakovať všetkým, ktorí ma podporovali a pomáhali mi, najmä svojim rodičom a starým rodičom za trpezlivosť a podporu pri štúdiu. V neposlednom rade ďakujem vedúcemu mojej práce, Mgr. Zbyňkovi Šlajchrtovi za cenné rady, ktoré mi dal.

# Abstrakt

Táto diplomová práca sa zaoberá problematikou spracovania komplexných udalostí. Teoretická časť obsahuje vysvetlenie základných pojmov a popis použitých technológií. Čitateľ sa tiež zoznámi so základmi práce v Esper.

Praktická časť práce sa zaoberá vytvorením administračného rozhrania, ktoré po napažení na serverovú časť aplikácie umožní použitie základnej funkcionality Esper engine aj bez predchádzajúcej znalosti programovania. Súčasťou praktickej časti je postup prípravy systému, inštalácie a spustenia aplikácie, keďže ide o netriviálnu úlohu.

## Kľúčové slová

cep, nosql, esper, cassandra, rest, rails, gui

# Abstract

This thesis deals with the complex event processing problem. In theoretical part you can find explanation of basic terms and description of used technologies. Reader also becomes familiar with basics of work with Esper.

Practical part is about building administrative interface, which after connecting to server part of application enables user to use basic functionality of Esper engine without previous programming knowledge. Part of this chapter is about system set-up, installation and launching the application as it is not considered to be a trivial task.

## Keywords

cep, nosql, esper, cassandra, rest, rails, gui

# Obsah

Úvod	1
<b>1 Definícia pojmov</b>	<b>3</b>
1.1 CEP . . . . .	3
1.2 NoSQL . . . . .	6
<b>2 Technológie</b>	<b>9</b>
2.1 Esper . . . . .	9
2.1.1 Typy udalostí . . . . .	10
2.1.2 Event Processing Language . . . . .	12
2.1.3 Api . . . . .	20
2.2 Cassandra . . . . .	22
2.2.1 Keyspace . . . . .	24
2.2.2 Primárny kľúč . . . . .	25
2.2.3 Dátové typy . . . . .	26
2.2.4 Offset, Join, Order, Count . . . . .	27
2.3 Frameworky . . . . .	28
2.3.1 Spring . . . . .	28
2.3.2 Ruby On Rails . . . . .	29
2.3.3 Sinatra . . . . .	30
2.3.4 Bootstrap . . . . .	31
<b>3 Návrh a Implementácia</b>	<b>33</b>
3.1 ThesisApi . . . . .	34
3.1.1 Konfiguračné súbory . . . . .	34
3.1.2 Databáza . . . . .	34

3.1.3	Spring framework . . . . .	39
3.1.4	Esper engine . . . . .	40
3.1.5	Maven . . . . .	42
3.2	ThesisWeb . . . . .	42
3.3	UseCase . . . . .	44
3.3.1	Twitter Stream . . . . .	44
3.3.2	Contacts Web . . . . .	45
3.4	Možnosti rozšírenia . . . . .	45
<b>4</b>	<b>Inštalácia &amp; Použitie</b>	<b>48</b>
4.1	Príprava systému . . . . .	48
4.2	Vytvorenie databázy . . . . .	51
4.2.1	Cassandra . . . . .	51
4.2.2	HSQLDB . . . . .	52
4.3	Spustenie aplikácie . . . . .	52
4.4	Použitie . . . . .	54
4.4.1	ThesisApi . . . . .	54
4.4.2	ThesisWeb . . . . .	55
	<b>Záver</b>	<b>61</b>

# Úvod

Rýchly nárast množstva dát produkovaných užívateľmi a aplikáciami prináša problémy s ich spracovaním a vyhodnocovaním. Pri analýze statických dát z bežnej databázy narážame na obmedzenia spôsobené dávkovým spracovaním dotazov. Technológia CEP<sup>1</sup> prináša možnosť spracovania prúdov informácií a ich komplexných závislostí v reálnom čase. Definuje pojem udalosť, ktorá predstavuje jednotku informácie s ktorou systém pracuje. Bližšie sa touto technológiou zaoberá kapitola 1.

Prvotným cieľom tejto diplomovej práce je vytvorenie grafického nástroja, ktorý umožní vykonávanie základných operácií potrebných pre prácu s Esper engine. Tými sú správa schém a príkazov, odosielanie udalostí a zobrazenie nájdených výsledkov. Použitím tohoto nástroja odpadá užívateľovi potreba znalosti programovacích platforiem java a net, pre ktoré je Esper oficiálne dostupný.

Druhotným cieľom je rozdelenie väzby medzi administratívnym rozhraním a Esper engine a umožnenie prístupu k základným operáciám definovaným v prvom ciele pomocou restovej api. Toto rozdelenie umožní vytváranie ďalších aplikácií, ktoré budú využívať Esper engine taktiež bez nutnosti znalosti programovania v jave alebo NET platforme.

So splnením druhého cieľa je tiež viazaný koncept deklaratívnej tvorby webových aplikácií. Bežne sa stretujeme s imperatívnym programovacím prístupom, kde je funkcionality riešená presne definovanými algoritmami, teda postupom ako danú úlohu vyriešiť. Deklaratívne programovanie naproti tomu hovorí čo sa má vykonať, nie ako sa to má vykonať. Využitie restovej api pre komunikáciu s Esper engine umožňuje vytvárať jednoduché webové aplikácie, ktorých dátová časť je riešená deklaratívne. Pred použitím je nutné definovať schému dát napríklad pomocou administratívneho rozhrania a nastaviť príkazy, ktoré budú zachytávať a filtrovať prichádzajúce dáta. Následne sa stačí odosielať nové záznamy na definovanú URL. Tie vyhovujúce definovaným filtrom budú prístupné vo výsledkoch

---

<sup>1</sup>Complex Event Processing



daného príkazu.

Tretím cieľom práce je umožnenie práce s historickými dátami. Jedným spôsobom realizácie bude možnosť odosielania súborov obsahujúcich dáta vo formáte XML. Druhým zaujímavejším riešením bude možnosť presmerovania výsledkov konkrétneho príkazu na vstup Esper engine ako nový zdroj dát.

Súčasťou práce je úvod do problematiky NoSQL databáz, ktorých použitie je vhodné hlavne pri spracovaní veľkého množstva dát. Keďže Esper engine je stavaný na takéto úlohy bude pre ukladanie výsledkov použitá práve NoSQL databáza.

Použitie diplomovej práce vyžaduje základnú znalosť Esper syntaxe a je koncipovaná ako pre začiatočníkov, ktorí môžu k Esperu pristupovať bez znalosti javy tak pre pokročilých užívateľov, ktorým môže uľahčiť a sprehľadniť prácu. Súčasťou práce je tiež postup prípravy systému a inštalácia, keďže tieto úkony nie sú triviálne.

Pri písaní práce som čerpal prevažne z online dokumentácie Esperu a Ruby On Rails frameworku. Tiež som využil bakalársku prácu Štefana Repčeka [1] a diplomovú prácu Jána Dema [2] ako množstvo iných dostupných materiálov dostupných prevažne online.

# Kapitola 1

## Definícia pojmov

### 1.1 CEP

CEP je definovaný ako sada nástrojov a techník na analýzu a kontrolu komplexného sledu vzájomne prepojených udalostí, na ktorých sú postavené moderné distribuované informačné systémy. Táto technológia pomáha ľuďom v IT obore rýchlo identifikovať a vyriešiť mnohé problémy. [3]

CEP je dôležitou súčasťou vývoja reaktívnych aplikácií a monitorovacích programov. Použitie nachádza v mnohých oblastiach, napríklad pre automatické obchodné systémy, detekcie podvodov, analýzu sentimentu trhu, optimalizáciu dodávateľského reťazca, transporte a logistike alebo systémoch pre rýchlu zdravotnú pomoc. Predpokladá sa tiež nárast jeho využitia v informačných systémoch spolu so zvyšujúcim sa počtom decentralizovaných zdrojov dát, napríklad blogov, a rozvojom tagovacích a senzorových technológií.

CEP stavia na dvoch nutných podmienkach: [4]

- Oddeľuje tvorcov a príjemcov informácií. Tvorcovia nevyžadujú informácie o príjemcoch, rovnako ako príjemci nepotrebujú vedieť kto dáta produkuje.
- CEP systémy okrem predávania informácií medzi tvorcami a príjemcami vo forme udalostí, ale umožňujú detekciu vzťahov medzi udalosťami.

Príkladom takého vzťahu je dočasný vzťah definovaný pomocou korelačných pravidiel. Pomocou agregácie a kompozície je možné generovať nové udalosti a z nich zase odvodiť ďalšie udalosti.

CEP sa podobne ako iné technológie vyvíja v čase a je postavený na základe definovanom SEP<sup>1</sup> a ESP<sup>2</sup> [5].

- SEP je najjednoduchším prípadom, kde je udalosť spracovaná izolovane bez ohľadu na ostatné udalosti.
- ESP spracováva prúdy udalostí ako kolekcie, identifikuje typy udalostí použitím kontinuálnych výrazov a selektuje zaujímavé udalosti.
- CEP je rozšírenie ESP o mechanizmy ECA<sup>3</sup> umožňujúce vykonávanie definovaných príkazov v závislosti na spracovávanej udalosti vyhovujúcej stanoveným podmienkam.

Pre prácu s komplexnými udalosťami je potrebné najprv pochopiť čo sa pod týmto pojmom skrýva. V ďalšom texte aj pri samotnej práci v administračnom rozhraní sa stretne s typmi udalostí a samotnými udalosťami. Tieto pojmy sú definované nasledovne [6]:

**Udalosť** je definovaná ako výskyt v danom systéme alebo doméne. Je to niečo čo sa stalo, alebo je predpokladané že sa stalo v danej doméne. Slovo udalosť je tiež používané vo význame programátorského objektu, ktorý reprezentuje výskyt udalosti v počítačovom systéme.

**Typ udalosti** je špecifikáciou pre skupinu objektov udalostí, ktoré majú rovnaký účel a štruktúru. Každý objekt udalosti je považovaný za inštanciu typu udalosti.

Pred samotným spracovaním udalostí je nutné definovať ich typ, príkazy spracovania a akcie, ktoré budú vykonané v prípade výskytu udalosti. Po splnení týchto úkonov stačí na vstup procesného engine poslať udalosti daného typu. Model fungovania takéhoto systému vidíme na obrázku 1.1. Na rovnakom princípe funguje serverová časť programu, ktorou sa bude bližšie zaoberať kapitola 3.

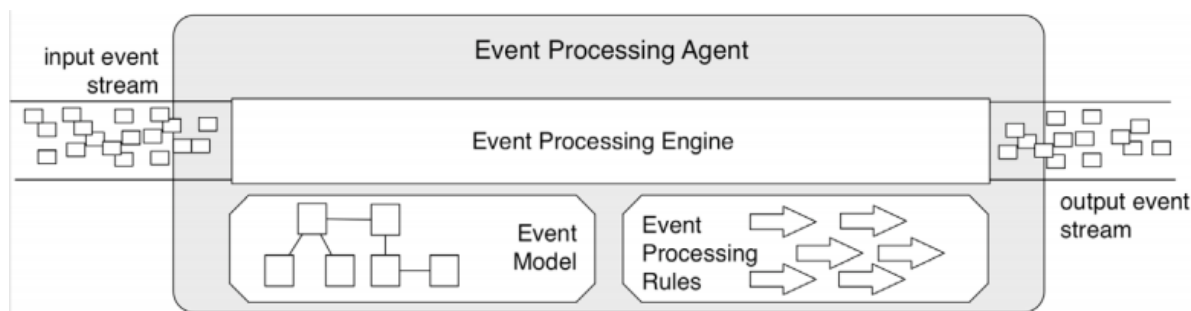
Na rozdiel do databázových systémov sú dotazy na do prúdu udalostí vyhodnocované priebežne, v momente výskytu danej udalosti. Aj keď databázy zvyčajne pracujú s údajmi o udalostiach (napríklad históriou objednávok), dotazy v databázach sú jednorázové a

---

<sup>1</sup>Simple Event Processing

<sup>2</sup>Event Stream Processing

<sup>3</sup>Event Condition Action



Obr. 1.1: Model spracovania udalostí [7]

ad-hoc, oproti konečnej množine dát. Naopak dotazy v CEP sú predom definované a množinou dát je nekonečný tok udalostí cez nich prechádzajúci [8].

Samotné spracovanie udalostí je postavené na logike, ktorú definujú pravidlá spracovania (príkazy). Tieto pravidlá si môžeme predstaviť ako filter, cez ktorý prichádzajúce udalosti pretekajú. Ak udalosť filtru vyhovuje, prevedú sa vopred definované akcie. Príkazy je možné definovať pomocou jazyka EPL, ktorý je syntaxou veľmi podobný SQL.

Systémy riadené udalosťami môžu automaticky reagovať na detekované udalosti podľa predom stanovených pravidiel. Tie zvyčajne pozostávajú z notifikácie (užívateľa alebo ďalšieho systému), jednoduchých akcií (automatický nákup akcií, aktivácia požiarneho systému) alebo interakciou s iným systémom (spustením nového procesu).

CEP patrí do nástrojov v kategórii BI<sup>4</sup>. Je modernou formou spracovania udalostí v reálnom čase a poskytuje takmer okamžitý pohľad na stav systému alebo jeho zmeny (udalosti). Rovnako ako ostatné BI nástroje je jeho úlohou pomáhať spoločnostiam rýchlejšie a lepšie sa rozhodovať. Pri porovnaní s tradičnými BI nástrojmi si je možné všimnúť viacero rozdielov, prevažne v architektúre a role tohoto systému. Taktiež rozhodnutia, ktoré sú založené na tradičných BI nástrojoch sú typicky pomalšie, napríklad rozhodnutia managementu ohľadom alokácie zdrojov, zmeny cien, plánovaní rozpočtu či úprave stavov zásob. Naproti tomu CEP je používaný k rozhodnutiam, ktoré je potrebné vykonávať s minimálnym časovým oneskorením. Takýmito sú napríklad monitorovanie elektrickej siete kvôli možnosti detekcie výpadkov, odhaľovanie finančných podvodov pri online transakciách či hľadanie vzorov v senzorových systémoch pre prevenciu voči mechanickým poruchám, čo umožňuje včasné objednanie náhradných dielov.

CEP nie je kompletná aplikácia, jeho logika je zvyčajne využívaná v rozsiahlejšom

---

<sup>4</sup>Business Intelligence

systeme alebo aplikácii. Na rozdiel od BI riešení, ktoré dávkovo spracovávajú dátové súbory je spustený nepretržite a spracováva prichádzajúce toky dát vo forme udalostí. Aplikácie využívajúce CEP majú zvyčajne zabudovaný systém upozornení pre oznámenie výskytu hľadanej udalosti alebo spustenie reakcie na daný stav systému.

Na trhu existuje viacero komerčných CEP riešení od firiem ako Tibco, Oracle či IBM. Zo zástupcov open-source produktov sú to napríklad JBoss Drools Fusion alebo Esper od firmy EsperTech, na ktorom bude postavená aj táto diplomová práca.

## 1.2 NoSQL

Priekopníkom vzniku NoSQL databázy boli vedúce internetové spoločnosti ako Google, Facebook, Amazon a LinkedIn - prekonávali tak limitácie konceptu relačných databáz pri použití v moderných webových aplikáciách. Dnes používajú organizácie NoSQL na riešenie problémov, ktoré priniesli štyri trendy: [9]

**Big Users** Pred pár rokmi bolo 1000 užívateľov konkrétnej aplikácie veľa, 10 000 užívateľov bol extrémny prípad. Dnes sú pripojení k internetu 2 miliardy ľudí, ktorí strávia online približne 35 miliónov hodín mesačne. Pre webové aplikácie teda nie je výnimočné mať milióny rôznych užívateľov denne.

**Big Data** Obrovský nárast používania internetu vyúsťuje v nárast údajov, ktoré užívatelia a aplikácie produkujú. Podľa odhadu bola v roku 2013 veľkosť uložených údajov 4,4 zetabajtov. Do roku 2020 sa navyše predpokladá desať násobný nárast. [10]. Užívateľské informácie, geografické dáta, sociálne grafy, údaje vyprodukované užívateľmi a aplikáciami a senzorové dáta sú príkladom nekonečných generátorov dát.

**Internet of Things** je trend, ktorý je definovaný stále rastúcim množstvom prepojených zariadení - ktoré generujú dáta. Dnes je k internetu pripojených okolo 20 miliónov zariadení, vrátane telefónov, tabletov, zariadení v nemocniciach, autách či skladoch. Tieto zariadenia získavajú údaje o svojom okolí, pohybe alebo počasí z ich 50 miliónov senzorov. [9] Napríklad telemetrické údaje, ktoré sú čiastočne štruktúrované a kontinuálne a pre SQL databázy s pevne definovanou schémou a štruktúrovaným spôsobom ukladania dát predstavujú problém.

**Cloud Computing** Množstvo aplikácií je dnes postavených na cloud infraštruktúre a využíva trojvrstvovú architektúru. V tej je k aplikáciám prístupované cez internet pomocou mobilnej aplikácie či internetového prehliadača. Load balancery zodpovedajú za rozloženie záťaže a smerujú prichádzajúce požiadavky na jednotlivé servery, ktoré obsluhujú logiku aplikácie. Pri rastúcej záťaži nie je problémom pridať do konfigurácie load balancera ďalší server a takto rozložiť záťaž. Problém nastáva v databázovej vrstve. Relačné databázy boli zvyčajne pôvodným riešením, avšak ich použitie je čoraz viac problematické pretože databáza je centralizovaná a škálovateľná vertikálne, nie horizontálne. To predstavuje nevýhodu pre dynamicky rastúce aplikácie. NoSQL databázy sú distribuované a škálovateľné horizontálne, čo umožňuje podobne ako pri load balanceroch jednoducho pridať ďalší server a rozložiť záťaž.

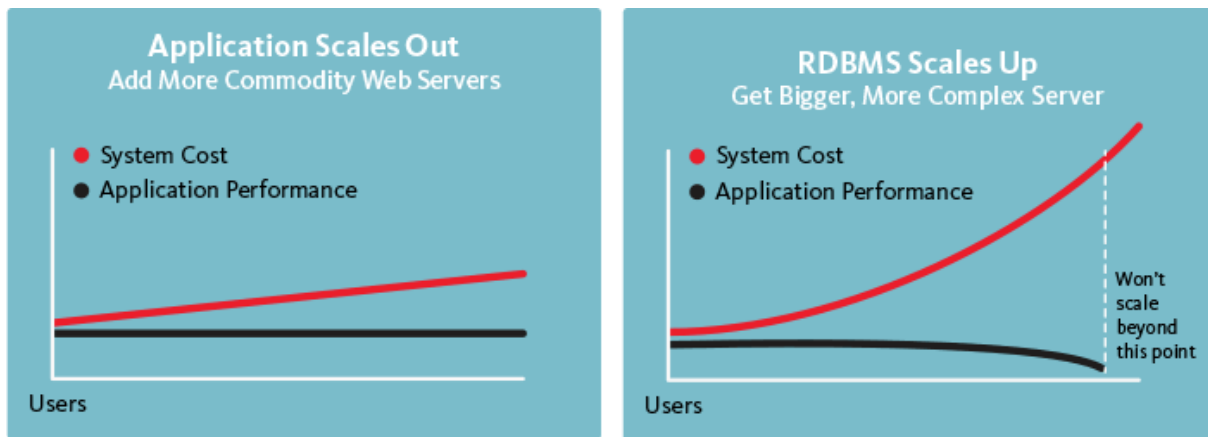
NoSQL používa rozdielny dátový model ako relačné databázy. Relačný model rozprestrie dáta do viacerých vzájomne prepojených tabuliek obsahujúcich riadky a stĺpce. Pri získavaní informácií z relačnej databázy musia byť údaje z týchto tabuliek spojené. Podobne pri zápise musia byť údaje rozložené do jednotlivých tabuliek.

Naproti tomu NoSQL dátový model agreguje ukladané dáta do jedného objektu - napríklad dokumentu v prípade objektových databáz. V NoSQL nie je definovaná operácia JOIN, čo vedie k duplikácii informácií vo viacerých tabuľkách. Tento problém však vyvažuje dnes lacný úložný priestor, flexibilita dátového modelu, zlepšený výkon operácií čítania a zápisu a škálovateľnosť tohoto systému.

Pre vyriešenie problémov s rastúcim počtom užívateľov a údajov je nutné aplikácie škálovať - a to horizontálne alebo vertikálne.

- Vertikálne škálovanie predstavuje centralizovaný prístup založený na vylepšovaní konfigurácie serverov, prípadne ich nahradení výkonnejšími servermi.
- Horizontálne škálovanie predstavuje distribuovaný prístup, kde sú do konfigurácie pridané ďalšie servery, ktoré umožňujú rozloženie záťaže.

Pred príchodom NoSQL bolo obvyklé vertikálne škálovanie. Pri raste množstva dát boli potrebné výkonnejšie servery s väčšou RAM, výkonnejšími procesormi a väčším diskovým priestorom. Cena a komplikovanosť takýchto serverov však neúmerne stúpa so zvyšujúcimi sa požiadavkami, ako je možné vidieť na obrázku 1.2.



Obr. 1.2: Škálovanie aplikácie vzhľadom na jej cenu [9]

Po istej úrovni už nie je možné konfiguráciu serveru vylepšiť, je nutné zaobstarať ďalší a rozloženie zaťaženia databázy riešiť na aplikačnej úrovni, čo je veľmi náročná úloha pre programátorov aj správcov databázy.

NoSQL zahrňuje širokú škálu rôznych databázových technológií v reakcii na prudký nárast množstva ukladaných dát, frekvencie prístupu k údajom a výkonnostným požiadavkom, ktoré z tohoto nárastu vyplývajú.

Relačné databázy neboli navrhnuté so zreteľom na požiadavky a nároky s ktorými sa dnešné aplikácie stretávajú. Tiež neumožňujú využiť výhody lacných úložísk dát a výpočtového výkonu.

# Kapitola 2

## Technológie

### 2.1 Esper

Esper je komponenta, ktorá umožňuje spracovanie komplexných udalostí CEP. Umožňuje vývoj aplikácií spracovávajúcich veľké množstvo udalostí - v reálnom čase ako aj historických. Tieto udalosti je možné filtrovať a analyzovať podľa potreby a reagovať v reálnom čase na predom definované stavy. Esper je dostupný v troch verziách:

**Esper** je open source s možnosťou komerčnej podpory. Táto verzia obsahuje základ potrebný pre realizáciu CEP, užívateľ však musí jednotlivé príkazy, schémy a nastavenia realizovať programovo. Je preto náročný na použitie pre ľudí, ktorí nevedia programovať. Riešenie je vhodné pre firmy, ktoré buď nevyužijú platenú verziu alebo majú špecifické požiadavky na výsledný produkt a sú schopné túto verziu podľa svojich potrieb upraviť.

**Esper HA** je riešenie umožňujúce vysokú dostupnosť Esperu. Zabezpečuje že stav je po vypnutí alebo havárii obnoviteľný. Príkazy, schémy a iné nastavenia si EsperHA pri reštarte uchováva, čo je výhoda oproti open source verzii, kde je nutné tieto úkony riešiť programovo. Táto verzia je vhodná pre projekty závislé na vysokej dostupnosti Esperu a subjekty, pre ktoré je kritická neustála kontrola prichádzajúcich udalostí. EsperHA je spoplatnený, dostupná je trial len verzia, pre ktorej použitie je nutné identifikovať sa ako spoločnosť. Cena nie je na webových stránkach dostupná.

**Esper Enterprise Edition** je kompletný produkt “na kľúč”, obsahujúci všetky komponenty potrebné pre nasadenie do podniku. V jednom balíku je obsiahnuté GUI pre



správu Esperu, restové služby poskytujúce prístup zvonku, EPL<sup>1</sup> editor, nástroje umožňujúce kontinuálne zobrazenie výsledkov v grafoch a tabuľkách. EsperEE je možné skombinovať s EsperEA pre dodatočné zabezpečenie vysokej dostupnosti. EsperEE je spoplatnený, rovnako ako pri EsperEA je dostupná trial verzia po splnení určitých podmienok. Cena nie je zverejnená a tieto dve riešenia sú určené predovšetkým pre podnikový sektor.

Pre tento projekt je použitá verzia Esper, ktorú som rozšíril o prístup k základným funkciám pomocou restovej api a persistenciu niektorých nastavení a nájdených výsledkov. Aktuálna verzia 5.1 je dostupná pod GNU General Public License (GPL) (GPL v2).

### 2.1.1 Typy udalostí

Každá udalosť spracovávaná Esperom je definovaná schémou, takzvaným typom udalosti. Tie môžu byť načítané pri štarte aplikácie, alebo nastavené programovo počas behu. EPL obsahuje klauzulu CREATE SCHEMA umožňujúcu definovanie typu udalosti. Prehľad základných typov udalostí je v tabuľke 2.1.

Trieda	Popis
java.lang.Object	Akýkoľvek Java POJO <sup>2</sup> s getter metódami. Takáto definícia je najjednoduchšia na úkor možnosti úprav počas behu programu.
java.util.Map	Udalosti definované ako implementácia java.util.Map interface, kde každá hodnota záznamu je vlastnosť udalosti.
Object[] (pole objektov)	Udalosti definované objektovým poľom, kde každá hodnota poľa je vlastnosť udalosti.
org.w3c.dom.Node	XML objektový model dokumentu popisujúci štruktúru udalosti.

Tabuľka 2.1: Možnosti definície typu udalosti

Definície typu udalosti sú rozšíriteľné zásuvnými modulmi. Aplikácia môže používať kombináciu týchto typov, nemusí všetky typy definovať jedným spôsobom. Definície typov

---

<sup>1</sup>Event Processing Language

udalostí je možné reťaziť, kde typom udalosti môže byť iná komplexná udalosť.

Z dôvodu nutnosti pridávania a mazania udalostí počas behu programu nemôže byť v tejto implementácii použitá definícia typu pomocou POJO. A pretože klient musí mať možnosť definovať typ, bola zvolená definícia pomocou XML schémy. Príklad jednoduchej schémy udalosti znázorňuje výpis 2.1.

---

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="StockEvent">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="time" type="xs:string"></xs:element>
        <xs:element name="open" type="xs:float"></xs:element>
        <xs:element name="high" type="xs:float"></xs:element>
        <xs:element name="low" type="xs:float"></xs:element>
        <xs:element name="close" type="xs:float"></xs:element>
        <xs:element name="volume" type="xs:float"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

---

#### Výpis 2.1: Príklad XML schémy udalosti

Po definovaní typu udalosti je Esper engine schopný prijímať udalosti v XML formáte. Príklad udalosti vyhovujúcej schéme 2.1 je vo výpise 2.2.

---

```
<?xml version="1.0"?>
<events>
  <StockXsd>
    <time>2014-02-03 01:58:00.000</time>
    <open>1.34850</open>
    <high>1.34854</high>
    <low>1.34850</low>
    <close>1.34853</close>
    <volume>76.9400</volume>
  </StockXsd>
  <StockXsd>
    <time>2014-02-03 01:59:00.000</time>
    <open>1.34852</open>
```

```
<high>1.34853</high>
<low>1.34845</low>
<close>1.34850</close>
<volume>89.5800</volume>
</StockXsd>
</events>
```

---

### Výpis 2.2: Príklad XML udalosti

Ako je z výpisu vidieť je možné udalosti zaobaliť do koreňového elementu events. Ten je vhodné použiť v prípade, že na Esper engine odosielame súbory obsahujúce veľké množstvo udalostí, pretože tým obmedzíme počet HTTP volaní a predídeme možnému zahlteniu serveru. Táto funkcionálna je realizovaná v implementačnej časti aplikácie a nie je súčasťou Esperu.

Po definovaní typu udalosti sa na ne môžeme odkazovať klauzulou FROM v EPL príkazoch. Tie sú bližšie popísané v nasledujúcej sekcii.

## 2.1.2 Event Processing Language

EPL je jazyk umožňujúci definovanie príkazov a vzorov v CEP. Syntaxou je podobný SQL, pretože obsahuje klauzuly ako SELECT, FROM, WHERE, GROUP BY, HAVING alebo ORDER BY. Namiesto tabuliek však pracuje s tokmi udalostí, kde riadok tabuľky nahrádza prichádzajúca udalosť. Toky udalostí je možné spájať pomocou JOIN, filtrovať alebo agregovať.

EPL definuje koncept pomenovaných okien (named windows), ktoré slúžia ako štruktúra uchovávať udalosti. Je možné do nej vkladať nové udalosti a mazať staré. Výhodou tejto štruktúry je možnosť jej použitia viacerými príkazmi, pretože je globálna, teda zdieľaná v rozsahu daného service providera.

Pomocou EPL môžeme tiež definovať premenné, ktoré sa dajú následne použiť napríklad na vkladanie parametrov do príkazov.

### Syntax

Príkazy musia spĺňať pravidlá definované EPL syntaxou. Tá však nie je jednotná a jednotlivé CEP riešenia poskytujú svoje implementácie. Táto časť práce sa zaoberá pravidlami, ktoré používa jazyk EPL Esperu. Výpis 2.3 zobrazuje štruktúru tejto syntaxe.

---

```

[annotations]
[expression_declarations]
[context context_name]
[into table table_name]
[insert into insert_into_def]
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]

```

---

Výpis 2.3: Vzor EPL syntaxe [11]

Ako môžeme vidieť v tomto výpise, každý EPL príkaz musí obsahovať minimálne klauzuly SELECT a FROM. Ďalej je možné filtrovať pomocou klauzuly WHERE, spájať prúdy udalostí pomocou JOIN alebo využiť relačnú databázu ako zdroj udalostí. Nasledujúci popis rozoberá základné EPL klauzuly [11].

**Select** Klauzula SELECT je povinná v každom EPL príkaze. Je v nej možné využiť náhradný znak \* alebo vymenovať všetky požadované položky. Ak položka nemá unikátne meno, musí sa použiť predpona s názvom zdroja dát.

V prípade použitia znaku \* v JOIN príkaze nebude výsledná udalosť obsahovať všetky položky oboch zdrojov. Namiesto toho bude pozostávať z položiek reprezentujúcich objekty daných udalostí pomenované podľa zdrojov.

Syntax select klauzuly je znázornená vo výpise 2.4. Môže obsahovať aj nepovinné parametre istream (input), irstream (input & remove) a rstream (remove), ktoré definujú na ktoré udalosti príkaz reaguje. Prednastavené je použitie parametru istream.

---

```

select [istream | irstream | rstream] [distinct] * | expression_list

```

---

Výpis 2.4: Syntax SELECT klauzuly [11]

Obsah klauzuly select tiež definuje typ udalostí vyprodukovaných daným príkazom.

**FROM** FROM klauzula špecifikuje jeden alebo viac zdrojov, pomenovaných okien alebo tabuliek (od verzie Esper 5.1). Tie môžu byť pomenované klauzulou AS. Pre join je potrebné definovať viacero zdrojov dát. Syntax from klauzuly je vo výpise 2.5.

---

```
from stream_def [as name] [unidirectional]
    [retain-union | retain-intersection]
[, stream_def [as stream_name]] [, ...]
```

---

Výpis 2.5: Syntax FROM klauzuly [11]

Podporovaný je tiež join s relačnou databázou ako zdrojom dát. To je možné využiť napríklad na prístup k historickým dátam.

**WHERE** Where klauzula je nepovinná časť príkazu, ktorá špecifikuje vyhľadávacie parametre. Zvyčajne obsahuje výrazy pozostávajúce z porovnávacích operátorov =, <, >, >=, <=, !=, <>, exists, is null a ich kombinácie pomocou kľúčových slov AND a OR.

---

```
where exists (
    select orderId from Settlement.win:time(1 min)
    where settlement.orderId = order.orderId
)
```

---

Výpis 2.6: Syntax WHERE klauzuly [11]

Klauzula where môže obsahovať tiež vnorené výrazy, ako je to znázornené vo výpise 2.6.

**JOIN** Klauzula FROM môže obsahovať viacero zdrojov dát. V tom prípade sú dátové zdroje spojené pomocou JOIN. Predvolene je použitý inner join, ktorý produkuje udalosti len v prípade výskytu vyhovujúcej udalosti vo všetkých zdrojoch. V prípade použitia outer join sa chýbajúce udalosti nahradia hodnotou null.

K dispozícii sú tiež varianty left outer join, right outer join a full outer join. Výpis 2.7 znázorňuje pravidlá syntaxe pri použití join.

---

```
...from stream_def [as name]
((left|right|full outer) | inner) join stream_def
[on property = property [and property = property ...] ]
[ ((left|right|full outer) | inner) join stream_def [on ...]]...
```

---

Výpis 2.7: Syntax JOIN klauzuly [11]

Každý z prúdov dát definovaný pomocou join klauzuly obsahuje vstupný a výstupný stream. Join tak môže byť realizovaný pri prijatí udalosti v ktoromkoľvek z týchto prúdov. Join je teda viacsmerový, prípadne dvojsmerný pri použití dvoch prúdov dát. EPL definuje kľúčové slovo *unidirectional*, ktoré umožňuje identifikovať jediný prúd dát poskytujúci udalosti ktoré spustia join. Všetky ostatné prúdy sa stanú pasívnymi. Keď je prijatá udalosť pasívnym prúdom dát, negeneruje join novú udalosť.

**OUTPUT** Output klauzula umožňuje kontrolovať rýchlosť ktorou sú produkované udalosti. Zvyčajne sa používa spolu s určením časového údaju. Tým môže byť napríklad výstup každých *n* sekúnd, *n* udalostí alebo v daný čas dňa. Časy výstupu je možné definovať tiež vo formáte *cronu*. Jeden zo spôsobov zápisu zobrazuje výpis 2.8.

---

```
output [after suppression_def]
[[all | first | last | snapshot] every output_rate [seconds | events]]
```

---

Výpis 2.8: Syntax OUTPUT klauzuly [11]

V tomto výpise vidíme aj možnosť definovania toho čo sa má vyprodukovať. Je možné si vybrať produkovanie všetkých udalostí, prvej, poslednej alebo snímky. Snímka sa používa spolu s agregáčnymi funkciami a produkuje jedinou udalosť s hodnotou agregáčnej funkcie.

Esper navyše umožňuje presmerovávať toku udalostí, prípadne ich za behu upravovať nasledujúcimi klauzulami:

**INSERT INTO** Túto klauzulu je možné použiť pre vloženie výsledkov príkazu do pomenovaného okna alebo tabuľky. Tiež umožňuje presmerovať tieto výsledky ako vstupný tok pre iný príkaz. Syntax pre použitie klauzuly *insert into* je zobrazená vo výpise 2.9. Príklad použitia je dostupný v nasledujúcom texte.

---

```
insert [istream | istream | rstream] into event_stream_name [
    (property_name [, property_name] ) ]
```

---

Výpis 2.9: Syntax INSERT INTO klauzuly [11]

**UPDATE** Klauzula UPDATE slúži na úpravu vlastností udalosti a je aplikovaná pred spracovaním príkazu.

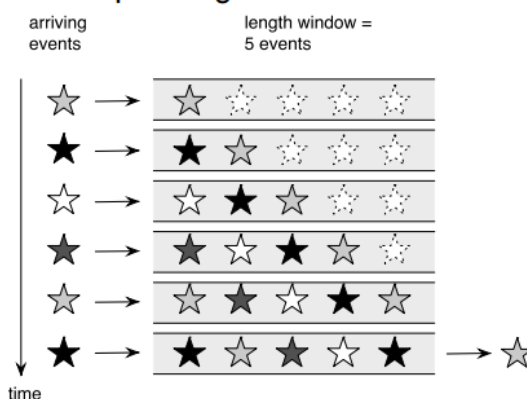
EPL tiež umožňuje definovať náhľady, ktoré predstavujú istú obdobu náhľadov (view) ako ich poznáme z databázových prostredí. CEP sa ale zaoberá prácou s tokmi dát a nie statickým pohľadom na ne, preto rozširuje tieto náhľady o viacero funkcií. Tými sú napríklad tvorenie štatistík z vlastností udalostí, ich zoskupovanie či funkcie pre umožnenie výberu udalostí, ktoré bude dátové okno obsahovať. Náhľady môžu byť reťazené. Názorný príklad fungovania náhľadov je na obrázku 2.1.

#### Example: time window

e.g.:  $(A \rightarrow B) [\text{win:time:5 min}] \rightarrow C$

- an instance of event type A is followed by an instance of type B within 5 minutes; then an event instance of C follows

#### Example: length window



Obr. 2.1: Príklad dátového okna obmedzeného časom a počtom udalostí [7]

Esper rozdeľuje náhľady do menných priestorov. V nasledujúcom zhrnutí sú predstavené tie základné [11].

**Náhľady do dátových okien** definujú kľzavé okná a nájdeme ich v mennom priestore win.

win:length - náhľad rozširuje okno o definovaný počet udalostí do minulosti. Nové udalosti vytlačajú tie, ktoré sa do okna už nezmestia, čo vytvára štruktúru podobnú konceptu fifo.

win:length\_batch - náhľad funguje podobne ako win:length, avšak udalosti odstraňuje nárazovo pri zaplnení okna o definovanej veľkosti.

win:time - náhľad rozširuje dátové okno o minulé udalosti obmedzené časovou značkou.

win:keepall - na rozdiel od predchádzajúcich náhľadov, ktoré udalosti nevyhovujúce podmienke z dátového okna odstránia, tento náhľad udržiava všetky prijaté

udalosti. Keďže sú všetky udalosti udržiavané v pamäti je nutné dať si pozor aby náhľad nezabral všetku dostupnú operačnú pamäť.

`win:firstlength` - je podobný `win:length` v tom že obmedzuje počet udalostí. Naproti nemu však v dátovom okne udržiava len prvých `n` udalostí.

`win:firsttime` - je ekvivalentný s `win:firstlength`, avšak udalosti nie sú obmedzené počtom, ale časom.

**Štandardné náhľady** Ostatné bežne používané náhľady sú dostupné s predponou `std`.

`std:unique` - tento náhľad uchováva len najaktuálnejšiu udalosť v prípade prijatia duplicitnej udalosti.

`std:size` - náhľad poskytuje prístup k premennej `size`, ktorá obsahuje počet udalostí prijatých daným príkazom. Náhľad vytvára novú udalosť len v prípade zmeny premennej `size`.

`std:firstevent` - náhľad udržiava len prvú prijatú udalosť. Všetky udalosti prijaté po nej sú ignorované. Toto správanie spôsobuje že je jeho forma podobná ako `win:length` o veľkosti 1.

**Štatistické náhľady** Štatistické náhľady pokrýva menný priestor `stat`.

`stat:uni` - umožňuje pristupovať k štatistickým hodnotám, napríklad priemeru, smerodajnej odchýlky, súčtu či rozptylu.

`stat:correl` - počíta korelačnú hodnotu. Funkcia vyžaduje minimálne dva parametre, ktorých korelačnú hodnotu počíta.

`stat:weighted_avg` - ako názov napovedá, náhľad umožňuje vypočítať vážený priemer. Podobne ako `stat:correl` vyžaduje aspoň dva parametra, prvý udáva údaj z ktorého sa počíta priemer a druhý jeho váhu.

EPL jazyk tiež umožňuje definovať vzory. *Patterny* sú výrazy, ktoré hľadajú zhodu podľa definovaného vzoru. Je možné ich definovať ako samostatný výraz alebo ako súčasť príkazu. Môžu sa vyskytovať kdekoľvek v klauzule `FROM`, vrátane `join`. Vďaka tomu ich je možné použiť v kombinácii s klauzulami `WHERE`, `GROUP BY`, `HAVING` a `INSERT INTO`.

V nasledujúcich výpisoch sú príklady príkazov, zobrazujúcich príklady syntaxe popísanej v predchádzajúcom texte.



---

```
select * from TweetEvent.win:time(60 sec) where message='happy'
```

---

### Výpis 2.10: Jednoduchý EPL príkaz

---

```
select sum(price) from OrderEvent.win:time(30 min)
output snapshot every 60 seconds
```

---

### Výpis 2.11: EPL príkaz s výstupom každých 60 sekúnd

---

```
select * from TickEvent.std:unique(symbol) as t,
NewsEvent.std:unique(symbol) as n
where t.symbol = n.symbol
```

---

### Výpis 2.12: Jednoduchý EPL príkaz použitím join

---

```
select a.custId, sum(a.price + b.price)
from pattern [every a=ServiceOrder ->
b=ProductOrder(custId = a.custId) where timer:within(1 min)].win:time
(2 hour)
where a.name in ('Repair', b.name)
group by a.custId
having sum(a.price + b.price) > 100
```

---

### Výpis 2.13: EPL príkaz s použitím vzoru [11]

---

```
insert into CombinedEvent
select A.customerId as custId, A.timestamp - B.timestamp as latency
from EventA.win:time(30 min) A, EventB.win:time(30 min) B
where A.txnId = B.txnId
```

---

### Výpis 2.14: Príklad použitia klauzuly INSERT INTO [11]

---

Počíta priemernú cenu akcie z udalostí prijatých v posledných 30 sekundách

```
select sum(price) from StockTickEvent(symbol='GE').win:time(30 sec)
```

Počíta počet udalostí StockTickEvent prijatých počas poslednej minúty

```
select size from StockTickEvent.win:time(1 min).std:size()
```

Počíta smerodajnú odchýlku z posledných 10 prijatých udalostí

```
select stddev from StockTickEvent.win:length(10).stat:uni(price)
```

---

### Výpis 2.15: Príklady použitia dátových náhľadov

---

```
update istream AlertEvent
set severity = 'High'
where severity = 'Medium' and reason like '%withdrawal limit%'
```

---

Výpis 2.16: Príklady úpravy udalosti pred spracovaním [11]

## Objektový model

Objektový model je sada tried poskytujúcich objektovú reprezentáciu príkazu alebo vzoru. Tá umožňuje zostrojiť, zmeniť alebo získať údaje z EPL príkazov a vzorov na vyššom stupni ako pri práci s textovou reprezentáciou. Objektový model pozostáva z objektového grafu, ktorého prvky je jednoducho prístupné. Objektový model umožňuje plný export do textovej formy a naopak.

Príkazy vo výpise 2.17 a 2.18 sú ekvivalentné. Podobným spôsobom je možné vytvárať príkazy, vzory, definovať premenné alebo vytvárať dátové okná.

---

```
select line, avg(age) as avgAge
from ReadyEvent(line in (1, 8, 10)).win:time(10) as RE
where RE.waverId != null
group by line
having avg(age) < 0
order by line
```

---

Výpis 2.17: EPL príkaz bez použitia objektového modelu [11]

---

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setSelectClause(SelectClause.create()
    .add("line")
    .add(Expressions.avg("age"), "avgAge"));
Filter filter = Filter.create("com.chipmaker.ReadyEvent", Expressions.in
    ("line", 1, 8, 10));
model.setFromClause(FromClause.create(
    FilterStream.create(filter, "RE").addView("win", "time", 10));
model.setWhereClause(Expressions.isNull("RE.waverId"));
model.setGroupByClause(GroupByClause.create("line"));
model.setHavingClause(Expressions.lt(Expressions.avg("age"), Expressions.
    constant(0)));
model.setOrderByClause(OrderByClause.create("line"));
```

---

Výpis 2.18: EPL príkaz s použitím objektového modelu [11]

Rovnako ako v textovej reprezentácii sú v objektovej reprezentácii klauzuly SELECT a FROM povinné. Pomocou objektového modelu je tiež možné skontrolovať syntax príkazu pred pridaním do Esper engine.

### 2.1.3 Api

Esper pre svoje ovládanie neposkytuje grafické rozhranie. Na komunikáciu používa api, ktorá definuje tieto primárne rozhrania:

- Rozhranie udalostí a ich typov
- Administrátorské rozhranie na vytváranie a správu EPL príkazov a vzorov a definovanie konfigurácie Esperu
- Runtime rozhranie, ktoré slúži na posielanie udalostí do Esperu, definovanie premenných a spúšťanie on-demand výrazov.

#### EP Service Provider

EPServiceProvider reprezentuje konkrétnu inštanciu Esperu. Každá takáto inštancia je nezávislá od ostatných a má svoje vlastné administrátorské a runtime rozhranie. Pri prístupe umožňuje rozhranie voľbu “getDefaultProvider” bez parametrov, ktorá vráti predvolenú inštanciu, alebo “getProvider” s textovým parametrom URI identifikujúcim konkrétnu inštanciu. Tá je vytvorená ak ešte neexistuje. Opakované volania s rovnakým URI vracajú stále rovnakú inštanciu. Túto funkcionality je možné využiť napríklad pre oddelenie pracovného prostredia viacerých užívateľov.

#### EP Administrator

EPAdministrator umožňuje registrovanie EPL príkazov, vzorov alebo ich objektovej reprezentácie do Esperu a to metódami createPattern, createStatement a create pre objektový model. Tieto funkcie poskytujú voliteľné parametre umožňujúce definovať meno príkazu a užívateľský objekt, ktorý je v implementačnej časti tejto práce využitý na ukladanie

dodatočných informácií o príkaze - napríklad definovanie TTL pri ukladaní výsledkov do databázy.

Po registrácii nového EPL výrazu rozhranie vracia inštanciu vytvoreného EPStatement, pomocou ktorej môžeme ovládať už vytvorený príkaz alebo pristupovať k výsledkom. Praktická časť práce z ovládacích funkcií využíva stop() a start(), ktoré definujú, či je príkaz aktívny.

Esper poskytuje tri možnosti ako pristupovať k výsledkom konkrétného príkazu. Tieto je možné rôzne kombinovať. Možnosti sú predstavené v nasledujúcom výpise:

**Listener** V prvom prípade aplikácia poskytuje implementáciu rozhraní UpdateListener alebo StatementAwareUpdateListener vytváranému príkazu. Takýto listener bude následne notifikovaný pri výskyte novej udalosti a metóde update bude predaná inštancia EventBean, ktorá obsahuje udalosť produkovanú niektorým z príkazov.

**Subscriber** Týmto spôsobom Esper posíla výsledky na definovaný subscriber. Je to najrýchlejšia možnosť, pretože Esper predáva typované výsledky priamo do objektov aplikácie, nemusí teba zostavovať inštancie EventBean ako v predošlom prípade. Nevýhodou je že príkaz môže mať registrovaný maximálne jeden subscriber, naproti predošlému spôsobu, kde umožňoval definovať viacero listenerov.

**Pull Api** Týmto spôsobom aplikácia pristupuje k výsledkom on-demand spôsobom, kde jednorázovou žiadosťou o výsledky daného príkazu získa zoznam EventBean prístupný pomocou iterátora. Toto je využiteľné v prípade kedy aplikácia nevyžaduje nepretržité spracovanie nových výsledkov v real-time.

V tomto projekte bol použitý prvý spôsob listenera. Aplikácia v tomto prípade použije implementáciu rozhrania StatementAwareUpdateListener, ktorá je registrovaná pri vytváraní nového príkazu metódou addListener. Vďaka použitiu rozhrania StatementAwareUpdateListener a nie UpdateListener získava aplikácia prístup k príkazu, ktorý konkrétnu udalosť vyprodukoval, pre všetky príkazy môže byť preto definovaný jediný spoločný listener.

Esper podporuje tiež spracovanie udalostí, ktoré nevyhoveli žiadnemu statementu. Tieto výsledky získame registrovaním implementácie rozhrania UnmatchedListener.

## EP Runtime

EPRuntime rozhranie slúži na odosielanie nových udalostí do Esperu k spracovaniu, nastavenie a prístup k hodnotám premenných a spúšťanie on-demand EPL výrazov. Na odosielanie nových udalostí slúži metóda `sendEvent`, ktorá je preťažená. Typ parametra tejto metódy indikuje typ udalosti odosielanej do Esperu. Tieto typy boli bližšie popísané v predchádzajúcich sekciách.

V prípade použitia XML definície typov udalostí sa pri spracovaní prichádzajúcej udalosti skontroluje, že meno koreňového elementu prichádzajúcej udalosti je zhodné s menom typu udalosti definovanej XML schémou.

Ak aplikácia nepozná EPL výrazy dopredu alebo nevyžaduje streamovanie výsledkov, je možné prostredníctvom EPRuntime spúšťať jednorázové výrazy. Tieto nie sú permanentné, po ich vykonaní je výsledok okamžite predaný aplikácii na spracovanie. Použitie nachádzajú napríklad v spojení s pomenovanými oknami a tabuľkami, ktoré je možné indexovať pre zrýchlenie prístupu.

## 2.2 Cassandra

Cassandra je databázový projekt, ktorý pôvodne vznikol vo firme Facebook. Neskôr bol zverejnený ako open-source a v roku 2009 bol prijatý do Apache inkubátora. V roku 2010 získal top prioritu a je naďalej vyvíjaný a dostupný pod Apache 2.0 licenciou [12].

Cassandra je distribuovaná databáza, ktorá umožňuje spracovanie a uchovávanie veľkého množstva dát rozložených na veľkom počte menej výkonných serverov, ako je to znázornené na obrázku 2.2. Táto architektúra zároveň poskytuje vysokú dostupnosť dát pri zabezpečení proti strate dát pri výpadku niektorého zo serverov. Cassandra je navrhnutá na použitie veľkého počtu počítačov (v ráde stoviek) podľa možností rozložených v rôznych častiach sveta.

Cassandra bola navrhnutá pre beh na cenovo dostupnom hardware a podporuje rýchly zápis veľkého množstva dát pri zachovaní efektívnosti prístupu k nim. Týmto pomáha znižovať firemné náklady na hardware.

Vďaka týmto vlastnostiam je Cassandra využívaná množstvom známych firiem, medzi ktoré patrí napríklad CERN, eBay, GitHub, Netflix, Twitter alebo Cisco. Veľké produkčné nasadenia obsahujú stovky TB dát v klastroch zložených zo stoviek serverov. Pri porov-



Obr. 2.2: Škálovanie databázy Cassandra s rastúcou záťažou [13]

naní výkonnosti s ostatnými NoSQL databázami Cassandra získava výborné výkonnostné výsledky aj vďaka svojej jednoduchej architektúre.

Cassandra je dostupná z dvoch zdrojov, prvým z nich stránka projektu Apache Cassandra. Tá je základnou verziou a obsahuje databázový engine a cqlsh nástroj slúžiaci ako vývojový shell. Pre jeho spustenie je nutné mať nainštalované interpretátor jazyka python.

Druhou je nadstavba tretej strany DataStax Cassandra, ktorá odlišuje komerčnú a nekomerčnú verziu. V nekomerčnej verzii nájdeme rovnako ako v Apache balíku databázový engine a cqlsh. Navyše na svojich stránkach DataStax poskytuje zdarma rozšírenia, ktoré zjednodušujú prácu s databázou, a to:

**OpsCenter** je grafický nástroj na správu databázy. Poskytuje prehľadné rozhranie pre administrátorov a vývojárov v ktorom je možné vidieť jednotlivé časti klastru. Umožňuje monitorovať stav, aktuálnu záťaž, pridávať a odoberať servery do konfigurácie klastru, nastaviť zálohovanie či generovať štatistiky. Pomocou neho je možné prehľadne spravovať klastre zložené zo stoviek serverov.

**DevCenter** Pre úpravu štruktúry a údajov databázy slúži nástroj DevCenter. Grafické rozhranie umožňuje po pripojení na databázový engine vytvárať a spúšťať dotazy v CQL jazyku. Pri vytváraní dotazov je automaticky kontrolovaná syntax a sú zvýraznené chyby s popisom. Obsahuje tiež interaktívne pomôcky na vytvorenie keyspace alebo tabuliek, export výsledkov alebo ukladanie dotazov. Tento nástroj sa dá zjednodušene vnímať ako grafická verzia cqlsh.

**Java driver** Pre použitie databázy v programe napísanom v jave je potrebný ovládač,

ktorý je možné stiahnuť práve na stránkach DataStax. Nájdeme tu tiež ovládače pre ďalšie vývojové platformy.

Aj keď v mnohom pripomína Cassandra relačnú databázu, nepodporuje plne relačný model. Namiesto toho poskytuje klientom jednoduchší dátový model a prináša návod ako niektoré chýbajúce funkcie nahradiť. Nasledujúci text rozoberá niektoré základné odlišnosti cassandry a relačných databáz.

### 2.2.1 Keyspace

Keyspace je možné prirovnať k schéme relačnej databázy. Slúži ako kontajner pre dáta, ktoré zdieľajú určité vlastnosti. Pri vytvorení keyspace je nutné určiť spôsob replikácie a počet kópií dát. Tieto kópie slúžia na zachovanie dátovej integrity v prípade výpadku niektorého zo serverov. Syntax pre vytvorenie nového keyspace je zobrazená vo výpise 2.19.

---

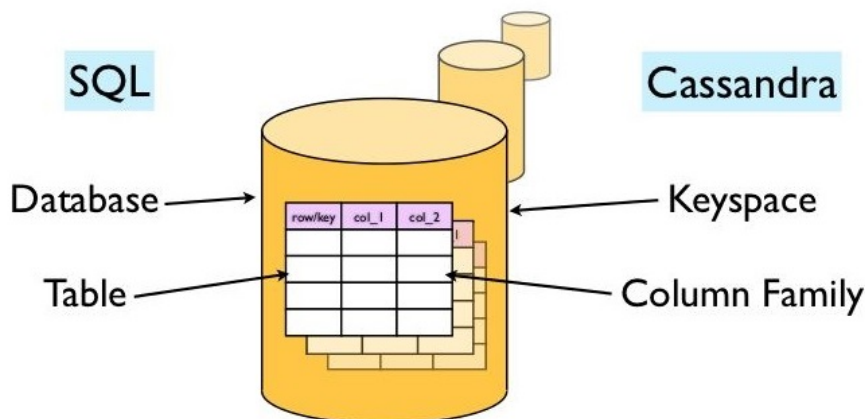
```
CREATE KEYSPACE <identifier> WITH <properties>
```

---

Výpis 2.19: Syntax pre vytvorenie nového keyspace

Po vytvorení keyspace je možné v ňom vytvárať tabuľky, ktoré už obsahujú samotné dáta. Tabuľka patrí do keyspace podobne ako v relačnom pojatí je tabuľka obsiahnutá v databáze.

## Keyspaces and Column Families



Obr. 2.3: Porovnanie štruktúry SQL a NoSQL databázy [14]

V predchádzajúcich verziách CQL bolo možné sa stretnúť so skupinami stĺpcov (column family). Tento pojem bol používaný v spojení s dynamickým modelom databázy, kedy stĺpce nebolo nutné definovať, ich definíciou sa vynucoval typ obsahu. V najnovšej verzii CQL sa však od tohoto prístupu upustilo, a dnes v aktuálnej verzii nájdeme tabuľky podobne ako ich poznáme z relačných databáz. Klauzula COLUMN FAMILY ostala ako synonym klauzuly TABLE. Porovnanie štruktúry SQL a NoSQL databázy je na obrázku 2.3.

### 2.2.2 Primárny kľúč

Podobne ako v relačnej databáze musí mať každý riadok unikátny identifikátor - primárny kľúč. Ten môže byť tvorený jediným údajom, alebo môže byť zložený z viacerých stĺpcov. Naproti relačnej databáze má primárny kľúč aj dodatočný význam.

---

```
create table thesis (  
    key_one text ,  
    key_two int ,  
    key_three int ,  
    data text ,  
    PRIMARY KEY(key_one , key_two , key_three)  
);
```

---

Výpis 2.20: Tvorenie primárneho kľúča v CQL

Vo výpise 2.20 je zobrazené názorné vytvorenie tabuľky so zloženým primárnym kľúčom. Tento kľúč má dve zložky:

**Kľúč partície** (Partition key) určuje na ktorých uzloch sa uložia dáta. Je tiež zodpovedný za distribúciu naprieč jednotlivé uzly. V príklade je to key\_one.

**Zoskupovací kľúč** (Clustering key) určuje zoskupovanie dát, čo je proces, ktorý usporadúva dáta v rámci daného úseku. V príklade je použitý zložený kľúč pozostávajúci z key\_two a key\_three.

Keďže Cassandra je distribuovaná databáza, tieto kľúče slúžia k vyhľadaniu miesta, kde sú uložené vyhľadávané záznamy. A práve toto vyhľadávanie je veľmi ovplyvnené architektúrou databázy.

Práve tu vzniká jedna z najväčších odlišností od relačných databáz, ktorých užívatelia sú zvyknutí filtrovať údaje podľa ľubovoľného stĺpca tabuľky. Vyhľadávanie v klauzule



WHERE je možné len podľa stĺpcov z ktorých sa primárny kľúč skladá a to v poradí v ktorom sú definované. Vo výpise 2.21 vidíme možnosti filtrovania v dotazoch do tabuľky definovanej výpisom 2.20.

---

Príklad validných dotazov:

```
select * from thesis where key_one='kluc' and key_two=11
select * from thesis where key_one='kluc' and key_two=11 and key_three=13
```

Príklad nevalidných dotazov:

```
select * from thesis where key_two=11
select * from thesis where key_three=11 and key_one='kluc'
```

---

Výpis 2.21: Príklad vyhľadávania v tabuľke thesis

Kvôli tomuto obmedzeniu je nutné prispôbovať návrh tabuliek dotazom, ktoré ich budú využívať. To nutne vedie k duplicitám dát vo viacerých tabuľkách. Toto je ďalšia dôležitá odlišnosť a jeden z problémov v zmene myslenia pri prechode zo sveta relačných databáz, kde je správnym prístupom normalizácia dát. Pre dnešné dátové úložiská však nepredstavuje taký problém množstvo dát ako rýchlosť prístupu k nim. Preto sú duplicitné dáta prijateľným kompromisom v porovnaní s výhodami ktoré NoSQL databázy prinášajú.

### 2.2.3 Dátové typy

Ďalším z rozdielov oproti bežným relačným databázam sú dátové typy, ktoré je možné využívať. Nasledujúci zoznam zhŕňa tie, s ktorými sa v relačných databázach bežne nestretneme.

**Kolekcie** Použitím dátového typu kolekcie môžeme definovať map, set a list. Pri definícii je nutné uviesť dátový typ, ktorý bude daná kolekcia obsahovať. Kolekcie sú vhodné na ukladanie relatívne malého množstva dát, napríklad telefónnych čísel daného užívateľa alebo popisov výrobku.

**TimeUUID** Typ UUID je používaný na predchádzanie kolíziám. Jeho rozšírenie TimeUUID obsahuje navyše časovú značku, čo je možné využiť v aplikáciách na vytvorenie jedinečného časového identifikátora.

Na výpočet TimeUUID je použitá MAC adresa, časová značka a sekvenčné číslo. Z vygenerovaného TimeUUID je možné spätne získať časovú značku, takže funguje ako časový identifikátor, ktorý je zároveň jedinečný.

**Tuple** umožňuje udržiavať stanovený počet vopred definovaných typov dát v jednom dátovom poli.

Výpis 2.22 zobrazuje príklad použitia týchto dátových typov. V prvom kroku je vytvorená tabuľka incidentov, ktorá používa ako unikátny identifikátor riadku časovú značku a obsahuje dva stĺpce - data uchovávajúce informácie o incidente a notified, ktorý obsahuje zoznam osôb ktoré majú byť upozornené na incident. V druhom kroku je do tabuľky vložený vzorový incident.

---

```
CREATE TABLE incident (
    tid timeuuid primary key,
    data <tuple<int , text , float >>,
    notified set<text>,
);

INSERT INTO incident (tid , data , notified) VALUES(
    now() ,
    (31, 'Sunny', 77.5) ,
    {'f@baggins.com', 'baggins@gmail.com'})
);
```

---

Výpis 2.22: Príklad použitia dátových typov databázy Cassandra

## 2.2.4 Offset, Join, Order, Count

V relačných databázach je samozrejmosťou výsledky radiť podľa potreby či spájať tabuľky. Cassandra má však rozdielnu architektúru, preto niektoré z typických klauzúl nenájdeme vôbec, prípadne je ich implementácia odlišná. Nasledujúci popis rozoberá hlavné rozdiely medzi Cassandrou a relačnými databázami z pohľadu práce s týmito klauzulami.

**OFFSET** Klauzuly limit a offset sú bežne používané v spojení so stránkovaním výsledkov vyhľadávania. Kvôli svojej distribuovanej architektúre však Cassandra neimplementuje klauzulu offset. Pri stránkovaní teda nejde jednoducho preskočiť na konkrétnu stranu. Implementácia teda zvyčajne spočíva v zobrazení výsledkov vo vzťahu ku konkrétnemu záznamu.

**JOIN** Klauzula JOIN je nahradená ústupom od normalizácie. Údaje sú uchovávané v tabuľkách duplicitne a ich štruktúra je prispôbena požiadavkám na operácie, ktoré

s nimi budú vykonávané. Potrebné join sa teda musia plánovať už pri návrhu tabuľky, prípadne majú za následok vznik nových tabuliek.

**ORDER BY** Klauzulu ORDER BY tiež nie je možné použiť ľubovoľne na každom definovanom stĺpci. V prípade že tabuľka definuje zoskupovací stĺpec, je možné ho použiť pre zoradenie výsledkov. V opačnom prípade je možné použiť len stĺpce definované v zoskupovacom kľúči.

**COUNT** Klauzula COUNT je implementovaná rozdielne ako v relačných databázach. Operácia pre zápis v databáze Cassandra prebieha bez nutnosti čítania už existujúcich záznamov. To zvyšuje rýchlosť zápisu, avšak databáza neudržiava informáciu o aktuálnom počte záznamov. Operácia count preto musí pri zavolaní spočítať existujúce záznamy, čo je časovo náročná operácia. Preto sa táto klauzula zvyčajne používa spolu s klauzulou limit, ktorá zamedzí zbytočnému zahlteniu databázy [15].

Tieto rozdiely sú pri prechode z relačných databáz prekvapením, návrh databáz je totiž z veľkej časti obmedzujúci. Obmedzenia vyplývajú predovšetkým z architektúry dátového úložiska databázy, avšak práve táto architektúra ponúka aj mnoho výhod. Preto je použitie NoSQL databázy nutné dôkladne zvážiť a prispôbiť potrebám konkrétnej aplikácie.

## 2.3 Frameworky

Pre uľahčenie vývoja projektov je dnes bežné použitie frameworku. Rozsahom malé aplikácie často vyžadujú použitie funkcionality (prihlasovanie, odosielanie mailov, práca s databázou, zjednotenie grafického zobrazenia pre rôzne platformy, správu závislostí), ktorej implementácia je zdĺhavá a v málo prípadoch lepšia ako pri opätovnom použití riešenia na to stavaného. Pri vývoji jednotlivých častí tohoto projektu boli použité voľne dostupné frameworky, ktorých krátkym popisom sa zaoberá táto sekcia.

### 2.3.1 Spring

Spring je framework napísaný v jave, distribuovaný pod Apache License verzie 2.0. Spring pozostáva z viacerých projektov zameraných na riešenie konkrétnych problémov vývoja aplikácie. V praktickej časti tejto aplikácie boli použité nasledujúce spring komponenty:

**Spring Framework** svojou funkcionalitou rieši základné oblasti vývoja java aplikácií.

Obsahuje základnú podporu pre injektovanie závislostí, správu transakcií, vývoj webových aplikácií alebo prístup k dátam. Táto funkcionalita je rozdelená do komponentov, v praktickej časti sú použité spring-core, spring-jdbc a spring-webmvc.

**Spring Boot** je spôsob ako urýchliť vývoj spring aplikácií. Rovnako ako Ruby On Rails sa prikláňa k prístupu convention over configuration. Z využitej funkcionality je dôležitý hlavne webový server Tomcat, ktorý tento komponent obsahuje. Vďaka nemu nie je nutné robiť deploy war súborov serverovej časti aplikácie, stačí jednoducho zdrojové kódy preložiť a spustiť. Je možné tiež využiť komponenty obsiahnuté v rodičovskom pom súbore. Spring boot automaticky nakonfiguruje spring aplikáciu bez nutnosti XML konfiguračných súborov s predvolenými nastaveniami.

### 2.3.2 Ruby On Rails

Mottom frameworku Ruby On Rails je snaha o dosiahnutie spokojnosti programátora a udržateľnú produktivitu. Framework uprednostňuje prístup convention over configuration, čo znamená že konfigurácia je preddefinovaná a ak programátor používa predom dohodnuté konvencie postačujú minimálne úpravy na dosiahnutie požadovaného výsledku. Konfiguráciu je samozrejme možné podľa potreby upraviť. Framework je voľne dostupný a distribuovaný pod MIT licenciou.

Vytvorenie a spustenie nového projektu pozostáva z jednoduchej série príkazov zobrazených vo výpise 2.23. Všetky závislosti sú automaticky stiahnuté pri vytvorení projektu nástrojom bundler.

---

```
rails new myweb
cd myweb
rails s
```

---

Výpis 2.23: Príklad vytvorenia a spustenia projektu v Ruby On Rails

Adresárová štruktúra projektu je prispôsobená MVC architektúre. Jednotlivé komponenty nájdeme v adresároch model, view a controller.

**Model** je vrstva reprezentujúca údaje a ich logiku. Umožňuje definovať objekty, ktorých dáta vyžadujú uloženie v databáze. Vlastnosti týchto objektov sú mapované na reálne dáta. Model je možné definovať pomocou migrácií, ktoré po spustení upravujú

štruktúru databázy a umožňujú rollback k predchádzajúcemu stavu. V modeloch je možné definovať kontroly vstupných dát, asociácie, funkcie na rozšírenie prístupu k údajom alebo spätné volania v závislosti na vykonávanej akcii.

**View** adresár je ďalej rozdelený do podadresárov reprezentujúcich jednotlivé controllery.

Dôležitým je tiež adresár layout, ktorý ako názov napovedá obsahuje jednotný layout aplikácie. Prípona `.erb` súborov umožňuje použitie ruby kódu. V tomto adresári sa stretneme tiež so systémom vkladania čiastkových elementov stránky príkazom `partial`. Adresár `helpers` umožňuje definovať funkcie použiteľné v `erb` súboroch, ktoré sprehľadňujú štruktúru kódu.

**Controller** je zodpovedný za obsluhu požiadavky a vyprodukovanie odpovedi. Jeho úloha zvyčajne pozostáva z prijatia požiadavky, získania alebo uloženia dát do databázy, definovania premenných pre zobrazenia potrebného view súboru. Controller poskytuje prístup k `request` a `response` objektom a definuje premennú `flash`, ktorá nesie správu o úspechu alebo chybe danej akcie.

Konfigurácia projektu je rozdelená do troch súborov podľa aktuálneho prostredia - `test`, `development` a `production`. Rovnako je možné podľa prostredia definovať rôzne databázy v súbore `database.yml`. V konfigurácii framework nájdeme tiež súbory `initializers`, ktoré sú spustené pri štarte projektu a inicializujú jednotlivé komponenty a súbor `routes` obsahujúci smerovanie prichádzajúcich požiadaviek na jednotlivé controllery. Závislosti a použité komponenty sú definované v súbore `Gemfile`.

### 2.3.3 Sinatra

Sinatra je jazyk použiteľný pre rýchly vývoj jednoduchých webových aplikácií postavených na ruby. Framework zapuzdruje jednoduchý webový server. Umožňuje definovať takzvané `route`, predstavujúce URL, ktoré aplikácia rozoznáva. Obsahom bloku definujúceho `route` je telo metódy, ktorá sa má vykonať pri zavolaní danej URL. Tieto bloky akceptujú vstupné parametre, predstavujúce `GET` a `POST` parametre. Príklad takejto routy zobrazuje nasledujúci výpis.

---

```
get '/hello/:name' do |n|  
  "Ahoj #{n}!"  
end
```

---

Výpis 2.24: Príklad definovania GET route vo frameworku Sinatra

V tejto aplikácii bola Sinatra použitá pri demonštrácii jedného z usecase - webu kontaktov. Celá logika aplikácie pozostáva vďaka použitiu tohoto frameworku z 32 riadkov.

### 2.3.4 Bootstrap

Bootstrap framework je najpopulárnejším HTML a CSS frameworkom pre vývoj webových projektov. Umožňuje rýchly a jednoduchý vývoj front-end aplikácie. Je dostupný vo forme css súborov ako aj sass pre jednoduché použitie v rails projektoch. Distribuovaný je pod MIT licenciou.

.col-md-8		.col-md-4
.col-md-4	.col-md-4	.col-md-4

Obr. 2.4: Ukážka Bootstrap grid systému [19]

Bootstrap poskytuje triedy upravujúce zobrazovanie základných HTML komponentov. Aplikovaný je pomocou premennej class na jednotlivých komponentoch. Distribúcia tiež zahŕňa sadu ikon, písem a javascript, ktorý je zodpovedný napríklad za zobrazenie vyskakovacích okien alebo pomocných textov.

---

```

<div class="row">
  <div class="col-md-8">.col-md-8</div>
  <div class="col-md-4">.col-md-4</div>
</div>
<div class="row">
  <div class="col-md-4">.col-md-4</div>
  <div class="col-md-4">.col-md-4</div>
  <div class="col-md-4">.col-md-4</div>
</div>

```

---

## Výpis 2.25: Príklad použitia Bootstrap grid systému [19]

Jednou z jeho základných súčastí je mriežkový systém rozloženia stránky. Pred jeho použitím je potrebné pridať elementom, ktoré obaľujú stránku triedu container. Mriežkový

system poskytuje responzívne rozhranie pozostávajúce z riadka obsahujúceho 12 stĺpcov, ktoré sa prispôsobujú podľa veľkosti obrazovky cieľového zariadenia. Tento systém umožňuje stránku rozdeliť na sekcie, ktoré je možné odsadiť, zarovnať, alebo rovnomerne rozložiť podľa potreby. Každý stĺpec funguje ako samostatná jednotka, ktorá môže obsahovať nový riadok s 12 stĺpcami. Stĺpce je možné spájať so skupín. Príklad použitia takéhoto rozloženia je vo výpise 2.25. Rozloženie stránky vytvorenej týmto kódom zobrazuje obrázok 2.4.

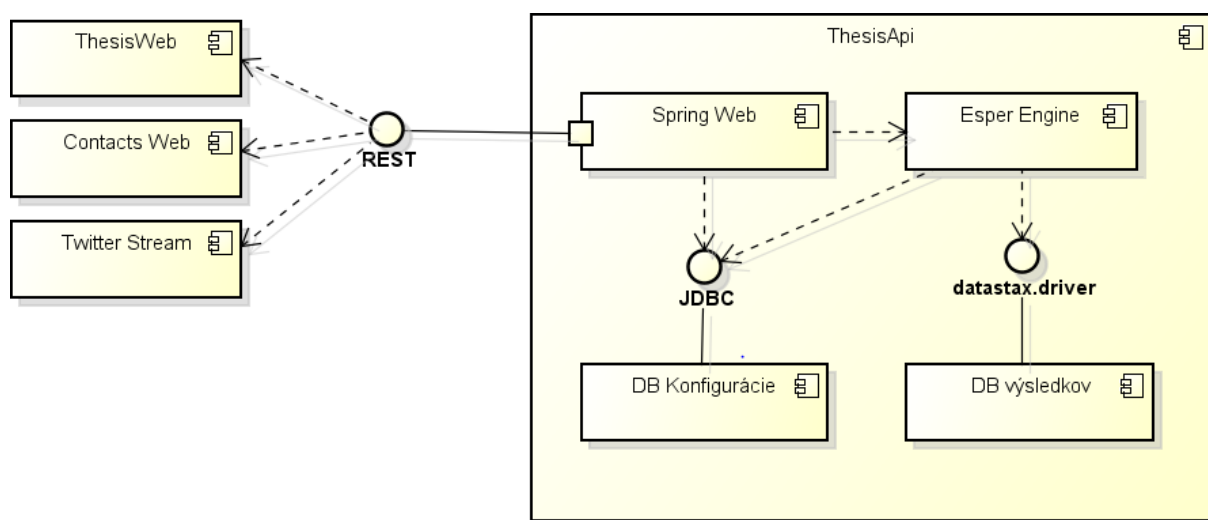
V tejto aplikácii bol bootstrap použitý pri vytváraní front-end administratívneho rozhrania ThesisWeb a usecase webu klientov.

Využitie frameworky a nástroje značne uľahčili vývoj projektu a v kapitole 4 je bližšie popísaný spôsob ich inštalácie a použitia. Všetky použité nástroje sú voľne dostupné.

# Kapitola 3

## Návrh a Implementácia

Táto kapitola popisuje implementačné detaily projektu. Kompletný projekt obsahuje štyri časti, serverovú api, jej administratívne rozhranie a dva príklady použitia. Administratívne rozhranie ako aj oba príklady použitia sú závislé na serverovej časti, na ktorej restové rozhranie sa pripájajú. Serverová api je nezávislá, jedinou prerekvizitou je databáza Cassandra, ktorá musí byť spustená ako prvá. Závislosti komponentov aplikácie zobrazuje diagram 3.1.



Obr. 3.1: Diagram komponentov aplikácie

Jednotlivé komponenty nájdeme vo výslednom projekte v samostatných adresároch. Projekt navyše obsahuje adresár so vzorovými schémami a udalosťami.



## 3.1 ThesisApi

Esper je voľne dostupný pre dve vývojové platformy - javu a .net. Pre serverovú časť aplikácie som zvolil javu s použitím spring frameworku.

Vo finálnej verzii som ponechal aj súbory v balíčku sample, ktoré slúžia na testovanie aplikácie. Ich použitím je možné jednorázovo spustiť serverovú časť aplikácie, načítať schémy a definovanými pravidlami vyhodnotiť udalosti uložené v csv súboroch v adresári resources. Takéto jednorázové spustenie uľahčuje hľadanie chýb, prípadne testovanie novej funkcionality aplikácie.

Súčasťou restovej api sú funkcie pre prístup k výsledkom v dvoch podobách, a to getAll a exportAll. V implementácii týchto funkcií sú dve hlavné rozdiely. Prvý je ten že funkcia getAll vracia aj metadáta obsahujúce napríklad informácie o stránkovaní. Druhou dôležitou odlišnosťou je streamovaný výstup funkcie exportAll, ktorý umožňuje získanie veľkého objemu dát bez nutnosti ich udržiavať v operačnej pamäti servera. Takto môžeme stiahnuť všetky výsledky daného príkazu, naproti funkcii getAll, ktorá vracia len časť dát obmedzenú parametrami požiadavku.

### 3.1.1 Konfiguračné súbory

Aplikácie využíva konfiguračné súbory umiestnené v adresári resources, ktoré sú načítané pri spustení. Najdôležitejšími z nich sú konfigurácie spring frameworku a databázy:

**application-config.xml:** Konfiguračný súbor spring frameworku, ktorého najzaujímavejšou časťou je konfigurácia jdbc. Tá špecifikuje adaptér pripojenia k databáze konfigurácie, konkrétne parametre pripojenia sú načítané zo súboru database.properties.

**database.properties:** Súbor obsahuje nastavenie pripojenia k databázi. Jeho úpravou môžeme jednoducho nahradiť použitý databázový systém iným.

Ďalej v tomto adresári nájdeme vzorové schémy udalostí a samotné udalosti použité pre testovanie alebo konfiguráciu logovania.

### 3.1.2 Databáza

Aplikácia využíva dve databázy, jednu na ukladanie konfiguračných dát a druhú na udalosti nájdené Esperom. Je to tak kvôli predpokladu že Esper bude produkovať veľké množs-

tvo dát, preto je vhodné použitie databázy optimalizovanej pre zápis. Správa konfigurácia na druhú stranu nezaťažuje databázový server, preto bolo pre ňu použité jednoduchšie riešenie.

Prístup k databázam zabezpečujú triedy v balíčku DAO, ktoré sú pre použitie injektované do ostatných objektov aplikácie. Balíček obsahuje tri triedy, dva z nich slúžia na správu schém a príkazov a využívajú konfiguráciu načítanú zo súboru `database.properties`. Tretia trieda slúži na prácu s výsledkami príkazov o jej konfiguráciu sa stará trieda `Constants.java`. Implementácia triedy obsluhujúcej výsledky je riešená trochu odlišne, kde sú všetky dotazy špecifikované na začiatku a v konštruktoze inicializované do objektu `PreparedStatement`. Je to takto z dôvodu optimalizácie rýchlosti zápisu.

Pred použitím aplikácie je nutné databázy vytvoriť a špecifikovať cestu k databáze konfigurácie v nastaveniach aplikácie. Tieto kroky sú bližšie popísané v kapitole 4.

### Databáza konfigurácie

Ako databázu pre ukladanie stavu Esperu a jednotlivých konfiguračných položiek som použil `HSQLDB`<sup>1</sup>. Táto databáza je napísaná v jave a pre ukladanie tabuliek ponúka pamäťový aj diskový mód. Použil som ju, pretože je jednoduchá, nenáročná na systém a je možné ju stiahnuť ako jednu zo závislostí aplikácie pomocou maven. Databáza je jednoducho nahraditeľná iným riešením, keďže k nej je pristupované pomocou `jdbc`. Pre zmenu je potrebné upraviť konfiguračný súbor `application-config.xml` a zmeniť položku `dataSource`.

Databáza obsahuje tabuľku schém a príkazov, ktoré sú načítavané pri štarte Api. Jej štruktúru vidíme na obrázku 3.2.

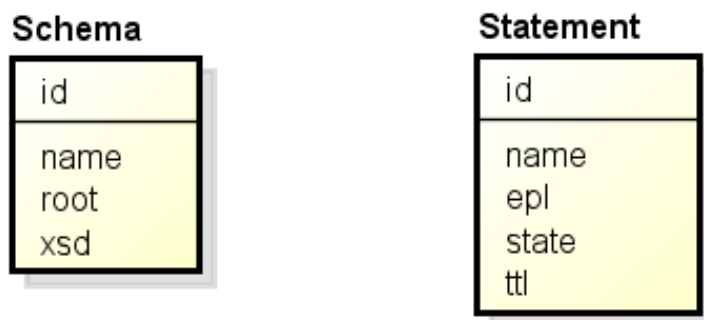
Pri ukončení aplikácie je nutné sa od databázy odpojiť aby boli zmeny dočasne uložené v pamäti presunuté na disk. Toto odpojenie zaobstaráva spúšťač súbor aplikácie, volaním príkazu “SHUTDOWN”. Odpojenie je nutné vykonať aj pri nastavení automatického ukončenia relácie v parametroch pripojenia k databáze.

### Databáza udalostí

Databáza udalostí slúži na ukladanie výsledkov vyhovujúcich niektorému z definovaných príkazov. Ako konkrétne riešenie som použil databázu `Cassandra`. `Cassandra` je jedným z

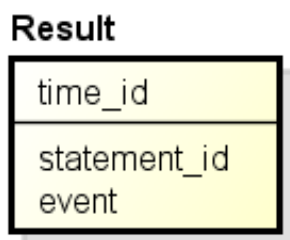
---

<sup>1</sup>HyperSQL DataBase



Obr. 3.2: Model databázy konfigurácie

predstaviteľov NoSQL databáz. Jednou z jej výhod je možnosť rozširiteľnosti v prípade veľkého objemu dát a optimalizácia pre zápis. Použitie tejto databázy je v rámci “proof of concept” princípu, kde by bolo jednoduchšie pracovať napríklad s HSQLDB ako v prípade konfigurácie, avšak kvôli možnosti generovania veľkého množstva udalostí Esperom je použitá databáza na to vhodná. Model tabuľky je na obrázku 3.3.



Obr. 3.3: Model databázy udalostí

Ako unikátny identifikátor je použitý time\_id, ktorý je typu TimeUUID. Tento typ je zložený z kombinácie časovej značky, mac adresy a sekvenčného čísla. Tá zaručuje unikátnosť a súčasne umožňuje zistiť čas vytvorenia záznamu. Stĺpec event obsahuje samotnú udalosť v JSON alebo XML formáte a statement\_id identifikátor príkazu, ktorý danú udalosť vyprodukoval.

---

```
CREATE KEYSPACE thesis WITH REPLICATION = { 'class' : 'SimpleStrategy', '
    replication_factor' : 1 };
```

```
create table statement_results(
    time_id timeuuid,
    statement_id int,
```

```

event text ,
primary key (statement_id , time_id))
with clustering order by (time_id desc);

```

---

### Výpis 3.1: Vytvorenie keyspace a tabuľky databázy udalostí

Vo výpise 3.1 je zobrazený spôsob vytvorenia tabuľky databázy udalostí. Keyspace bolo vytvorené na testovacie účely, preto bol replikačný faktor nastavený na 1 - teda bez zálohy.

Dôležitým prvkom je primárny kľúč. Ten je zložený z dvoch častí, kľúča partície a zoskupovacieho kľúča. Takéto zloženie primárneho kľúča je potrebné kvôli možnosti vyhľadávania v tabuľke. Každý výsledok patrí nejakému príkazu, preto pri pristupovaní k výsledku musí byť špecifikovaný identifikátor daného príkazu. Zoskupovací kľúč bol použitý preto, aby bolo možné vyhľadať konkrétny výsledok, pretože pre použitie v klauzule WHERE sú prípustné len polia definované v primárnom kľúči a aj to v danom poradí.

Vďaka takto definovanému primárnemu kľúču tabuľka umožňuje dotazy vo výpise 3.2, ktoré sú použité v aplikácii.

---

Vybrať jeden výsledok

```
select * from thesis where statement_id = ? and time_id = ?
```

Vybrať všetky výsledky s obmedzením počtu

```
select * from thesis where statement_id = ? and time_id <= ? limit ?
```

---

### Výpis 3.2: Príklad príkazov vyhovujúcich definovanému primárnemu kľúču

Ďalším rozdielom oproti relačnej databáze je spôsob implementácie stránkovania výsledkov, pretože v implementácii cassandry nenájdeme klauzulu OFFSET. Stránkovanie teda pozostáva z možnosti zobrazíť prvú a poslednú stranu a posúvať sa z aktuálneho zobrazenia na ďalšie alebo predošlé výsledky. Táto funkcionality bola riešená nasledovne: [16]

**Prvá stránka** zobrazuje údaje jednoducho získané usporiadaním výsledkov podľa časovej značky vytvorenia. Tento čas je obsiahnutý v identifikátore time\_id.

**Predchádzajúca stránka** žiadosť o zobrazenie predchádzajúcej stránky je vždy odosielaná spolu s identifikátorom začiatku tejto stránky. Takto server načíta výsledky, ktoré začínajú daným identifikátorom.

**Nasledujúca stránka** podobne ako pri zobrazení predchádzajúcej stránky aj nasledujúca stránka vyžaduje identifikátor začiatku požadovaných údajov.

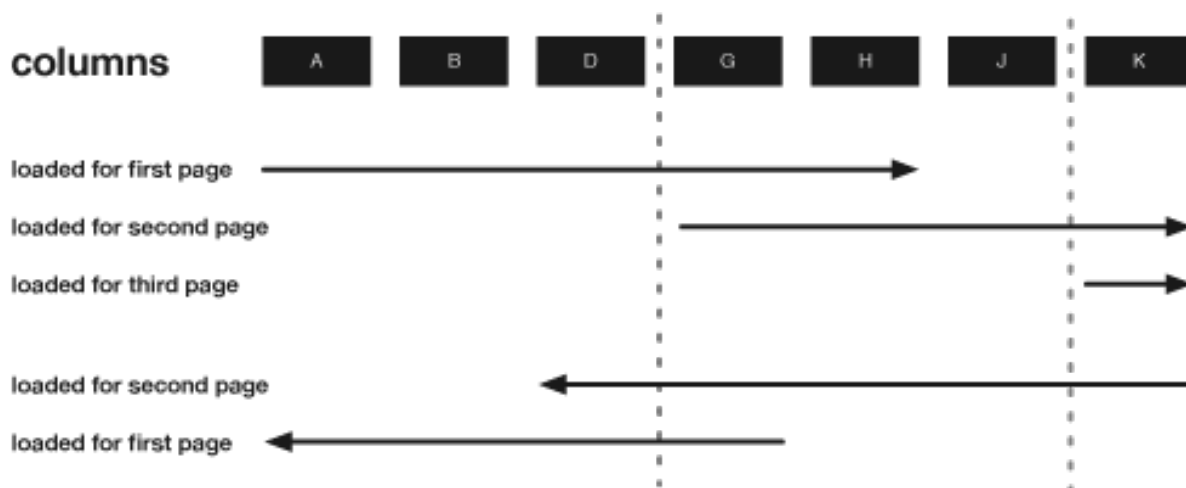
**Posledná stránka** údaje pre poslednú stránku sú získavané podobne ako pre prvú, avšak záznamy sú usporiadané v opačnom poradí.

Aplikácia pri požiadavku o zobrazenie výsledkov vždy načíta 2 údaje navyše - zmení parameter klauzuly limit. Taktiež funkcia očakáva identifikátor začiatku (alebo konca) stránky, ktorá sa bude načítavať.

**Posun ďalej** Prijatý identifikátor je použitý ako informácia o pozícii predchádzajúcej stránky. Údaj načítaný navyše slúži ako identifikátor nasledujúcej stránky. Ak tento údaj neexistuje, značí to že bude zobrazená posledná stránka.

**Posun späť** Predchádzajúca stránka je dostupná pomocou údajov načítaných navyše. Tie tiež slúžia na zistenie existencie predchádzajúcej stránky. Pre informáciu o nasledujúcej stránke sa použije prijatý identifikátor.

Tento postup je pre názornosť vysvetlený na obrázku 3.4. Údaje o nasledujúcej a predchádzajúcej stránke sú odosielané klientovi spolu s výsledkami. Funkcia pre export neprodukuje metadáta o stránkovaní.



Obr. 3.4: Model stránkovania bez použitia klauzuly OFFSET [16]

Pre zobrazenie výsledkov je nutné definovať offset vo forme TimeUUID. Ako alternatívu je možné použiť kľúčové slová first a last, ktoré sa premietnu do minimálneho a maximálneho možného TimeUUID.

Pri funkcii na preposlanie historických udalostí je možné dáta obmedziť časovými značkami. Pre ich porovnanie s údajmi v databáze bolo nutné použiť funkcie CQL `maxTimeuuid` a `minTimeuuid`, ktoré slúžia na porovnávanie časových značiek s `TimeUUID` identifikátorom.

### 3.1.3 Spring framework

Ako základ serverovej časti aplikácie som použil Spring framework. Jeho hlavnou úlohou je injektovanie závislostí vo väčšine tried, no využil som aj rozšírenia pre databázu, webový prístup a pribaleny webový server tomcat. Spring Framework zároveň poskytuje v aplikácii podporu pre správu transakcií a prístup k dátam pomocou jdbc. Serverová časť aplikácie využíva nasledujúce knižnice:

**spring-core:** Pomocou anotácie `@Autowired` sú v aplikácii riešené závislosti väčšiny komponent.

**spring-jdbc:** Prístup do databázy je realizovaný pomocou spring triedy `NamedParameterJdbcTemplate`, ktorá oproti jdbc pridáva možnosť prístupu k vygenerovanému id nového záznamu.

**spring-webmvc:** Prístup k dátam a ovládaniu serverovej časti aplikácie je umožnený pomocou restovej api.

**spring-boot-starter-web:** Táto závislosť umožňuje použitie pribaleného tomcat serveru. Ten sa spustí pri štarte aplikácie a sprístupní restovú api.

Súčasťou spring frameworku je webové rozšírenie umožňujúce tvorbu restových služieb. V tejto implementácii som tieto služby rozdelil do štyroch zdrojových súborov podľa oblastí, ktoré obsluhujú. Prvé dva `SchemaController` a `StatementsController` poskytujú prístup k správe schém a príkazov, `ResultController` umožňuje prácu s výsledkami príkazov a `EsperController` sa stará o spracovanie prichádzajúcich udalostí. Každá trieda je anotovaná pomocou `@RestController` a každá metóda definuje unikátnu cestu a HTTP metódu, pomocou ktorej sa vzdialene volá. Za zmienku tiež stojí závislosť `ResultController` na `StatementsController`, kde cesta k volaniu jednotlivých metód je vnorená. Výpis 3.3 zobrazuje cesty ku controllerom, pri ktorých si môžeme všimnúť že akceptujú vstup

vo forme JSON. V nasledujúcich kapitolách budú tiež detailne špecifikované prístupové cesty k jednotlivým REST metódam.

---

```
@RequestMapping(value = "Esper", produces = "application/json")
@RequestMapping(value = "schemas", produces = "application/json")
@RequestMapping(value = "statements", produces = "application/json")
@RequestMapping(value = "/statements/{statement_id}/results", produces =
    "application/json")
```

---

### Výpis 3.3: Definícia ciest REST API

Spustenie aplikácie je implementované v triede `Application.java`, v ktorej je načítaná konfigurácia. Pomocou rozšírenia `spring boot` sú použité prednastavené (nekonfigurované) hodnoty podľa prístupu `convention-over-configuration`. Následne je automaticky spustený aplikačný server `tomcat`, ktorý je súčasťou distribúcie.

#### 3.1.4 Esper engine

Esper je v aplikácii obsluhovaný triedou `EsperManager`, ktorá poskytuje prístup ku konfiguračným nástrojom a sprístupňuje handler prichádzajúcich udalostí. Pri spustení aplikácie načítava konfiguračné údaje z databázy a inicializuje Esper. Najdôležitejšou úlohou tejto triedy je správa schém a príkazov. `EsperManager` tiež implementuje službu `ping`, ktorá umožní detekovať či je engine spustený.

**Schémy** sú reprezentované XML (`org.w3c.dom.Node`) dokumentom. Oproti POJO reprezentácii umožňuje tento formát vytváranie schém počas behu čo je z pohľadu užívateľa nutnosťou. V prípade potreby by bolo možné použiť udalosti reprezentované pomocou `java.util.Map` alebo objektovým poľom, avšak to vyžaduje definovanie formátu prenosu a následné spracovanie do požadovaného formátu klientom, čo nie je veľmi intuitívne. Použitie XML reprezentácie klientovi umožní komunikovať priamo s Esperom.

**Udalosti** ktoré engine spracováva musia byť v XML formáte. Toto obmedzenie je zapríčinené XML reprezentáciou schém. Ako zdroje udalostí týmto eliminujeme adaptéry CSV a HTTP z `Esperio` knižnice. Na prijímanie XML udalostí slúži rest služba, ktorá spracováva vstupný stream dvoma spôsobmi:

- Ako jednotlivé udalosti, kde koreňový element udalosti reprezentuje názov schémy reprezentujúcej udalosť.
- Ako stream udalostí, kde je koreňový element nazvaný “events” a obaľuje jednotlivé udalosti definované v predošlom bode. Koreňový element je v tomto prípade ignorovaný.

**Príkazy** pridávané do Esper engine môžu obsahovať len meno a epl výraz. Preto je každému príkazu priradený aj užívateľský objekt - `statementBean` s dodatočnými informáciami:

- ID ktoré unikátne identifikuje príkaz v rámci celej aplikácie, nie len daného Esper provideru
- TTL hovoriaci ako dlho má byť nájdená udalosť perzistentná.
- STATE udávajúci či je konkrétny príkaz spustený alebo zastavený

Pri pridávaní príkazu do Esper engine je kontrolovaná unikátnosť mena v rámci daného Esper providera. V prípade že existuje príkaz s rovnakým menom je meno doplnené o “-N”, kde N značí najbližšie voľné celé číslo. Takto upravený príkaz je následne uložený.

Pri pridávaní XML schém udalostí je nutné ich konvertovať do typu `Document`. To zaobstaráva trieda `Helpers.java`, ktorá sa zároveň stará napríklad o konvertovanie typov udalostí do JSON formátu. Dôležitou súčasťou tejto triedy je funkcia `event-TypeToJSON`, ktorá rekurzívne skonvertuje typ udalosti a jej dedičností do JSON objektu vhodného k odoslaniu klientovi.

Aplikácia definuje jeden globálny listener, ktorý je priradený všetkým príkazom. Ten v prípade výskytu udalosti vyhovujúcej niektorému z definovaných príkazov uloží udalosť vo forme obsahu json objektu do databázy.

Esper umožňuje export výsledkov pomocou `JSONRenderer` a `XMLRenderer`, avšak tieto triedy produkujú formátovaný výstup, čo je v serverovej časti aplikácie neželané. Preto je konverzia realizovaná upravenými verziami týchto tried, ktoré nepridávajú formátovacie znaky a produkujú validné XML a JSON dokumenty. Predvolene je použitý upravený `JSONRenderer`.



Pri ukladaní týchto výsledkov do databázy je nastavený TTL, ktorý bol definovaný pri vytvorení príkazu. Ak TTL pri vytvorení príkazu nebolo definované použije sa predvolené nastavenie, kde sú výsledky persistentné až kým ich užívateľ manuálne neodstráni.

### 3.1.5 Maven

Zostavenie projektu je jednou z nevyhnutných súčastí tvorby java aplikácií. Na uľahčenie tohoto procesu je možné použiť viacero nástrojov, ktorých hlavnými predstaviteľmi sú maven, gradle a ant. Api komponent tejto aplikácie je zostavený pomocou nástroja maven.

Maven uľahčuje prácu vo viacerých oblastiach, a to [17]:

- Uľahčenie prekladu aplikácie
- Poskytnutie jednotného riešenia pre zostavenie aplikácie
- Poskytnutie informácií o projekte
- Poskytnutie vzorov pre vývoj aplikácií
- Umožnenie transparentnej migrácie nových vlastností programu

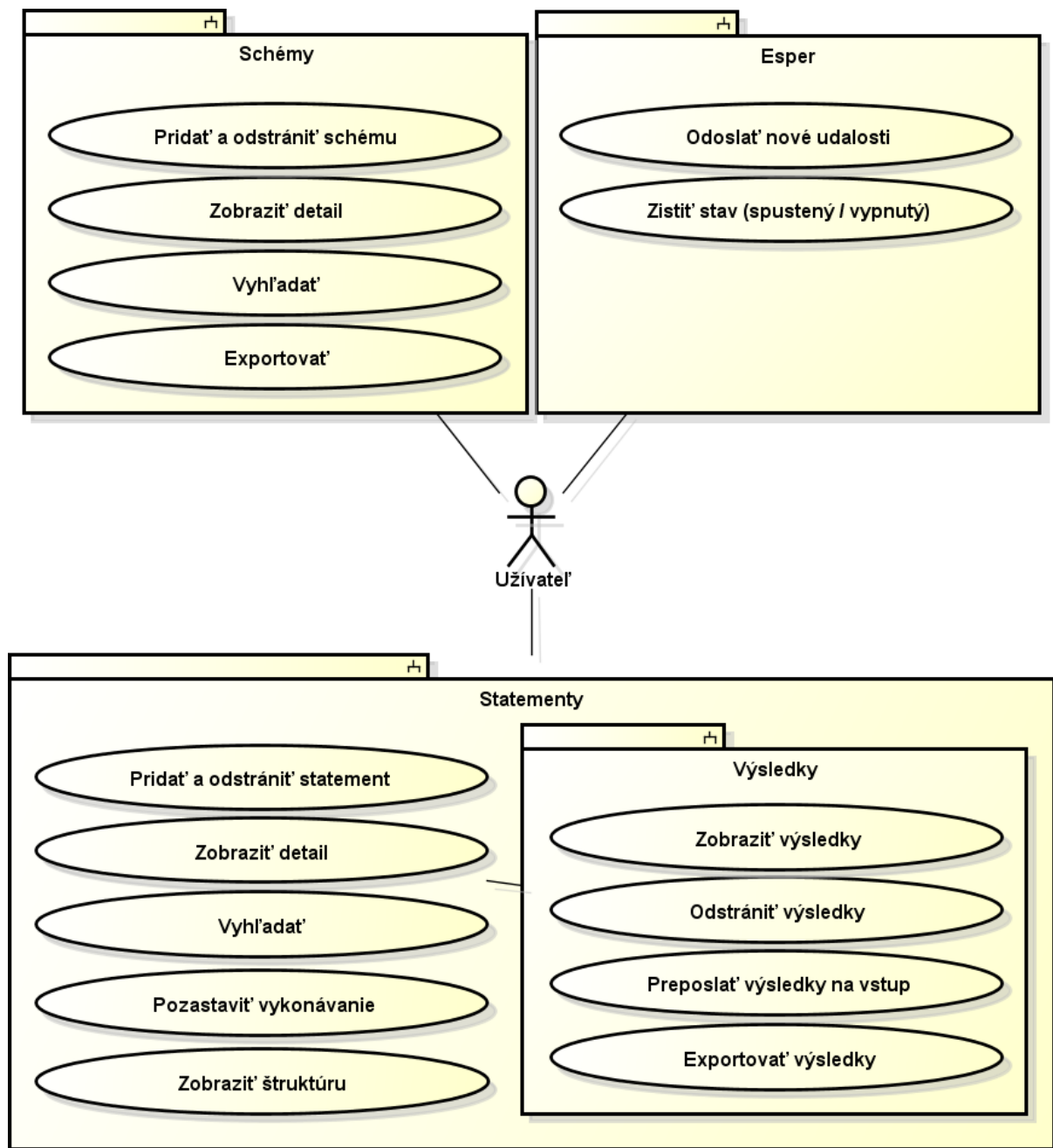
Pre túto prácu je najdôležitejšia prvá oblasť. Vďaka mavenu nemusí distribúcia projektu obsahovať všetky knižnice závislostí, ani nemusia byť jednotlivito sťahované pri zostavovaní projektu. Všetky potrebné závislosti sú definované v súbore pom.xml a pri preklade automaticky stiahnuté z internetu. Tento súbor zároveň definuje vlastnosti projektu ako názov, verziu, verziu javy použitú pre zostavenie, spôsob generovania dokumentácie a iné detaily.

Maven tiež umožňuje vytvárať v pom.xml súboroch závislosti a odkazovať sa na externé konfigurácie. Táto funkcionality je v projekte využitá pri spring závislostiach, kde je verzia niektorých komponent definovaná v externom rodičovskom súbore. Rodičom spring závislostí projektu je spring-boot-starter-parent.

## 3.2 ThesisWeb

Web časť projektu slúži ako administratívne rozhranie. Je riešená formou webovej aplikácie, ktorej základ je framework Ruby On Rails. Ten umožňuje rýchle vytváranie stránok, kde

prevažuje prístup convention over configuration. Dôležitou súčasťou frameworku sú gemy, ktoré reprezentujú závislosti projektu. Jedným z najpodstatnejších v tomto projekte je gem her, ktorý zabezpečuje komunikáciu s Api časťou projektu.



Obr. 3.5: UseCase diagram

Grafická stránka tohoto projektu sa opiera o framework Bootstrap. Ten v základnej konfigurácii poskytuje html komponenty s vylepšeným UI. Tiež opravuje niektoré chyby kompatibility pri zobrazovaní stránok v rôznych prehliadačoch.

Web nevyužíva žiadne persistentné úložisko dát. Všetky realizované zmeny sú posielané na restovú api, ktorá zmeny spracuje a uloží. Zobrazované údaje sú tiež získavané zo vzdialeného zdroja. Pre prípad nedostupnosti serverovej časti je a rodičovskom controlleri odchyťovaná výnimka `Faraday::ConnectionFailed`, ktorá klienta presmeruje na chybovú stránku. Po pri znovu načítaní stránky sa na kontrolu dostupnosti používa vzdialené volanie funkcie `ping`.

Jediná zmena v prípade úpravy IP adresy serverovej časti je potrebná v konfiguračnom súbore iniciátora `her.rb`. V týchto súboroch tiež nájdeme úpravu konfigurácie stránkovanie, umožňujúcu pomocou gemu `will_paginate` stránkovať polia.

Administračné rozhranie sa skladá zo štyroch hlavných častí. Prvými dvoma sú správa schém a príkazov, od ktorej sa odvíja správa výsledkov nájdených jednotlivými príkazmi. Na odosielanie udalostí je využitá súčasť pre komunikáciu s Esper engine. Funkcionalitu jednotlivých častí zobrazuje diagram 3.5. Je z neho tiež vidieť, že aplikácia nerozlišuje role užívateľa.

Výsledky sú vždy výsledkom vyhľadávania konkrétneho príkazu, preto sú v usecase diagrame zobrazená ako podsystém príkazov. Dostať sa na stránku s výsledkami je tiež možné len zo stránky detailu jednotlivých príkazov.

## 3.3 UseCase

Na predvedenie možností použitia aplikácie som vytvoril 2 jednoduché programy. Prvý Twitter Stream demonštruje možnosť zasielania udalostí na server a Contacts Web predstavuje možnosť alternatívneho použitia Esperu.

### 3.3.1 Twitter Stream

Twitter Stream predstavuje možnosť zasielanie streamu udalostí na REST rozhranie serveru. Skript je napísaný v Ruby a demonštruje ako sa je možné pomocou necelých 30 riadkov kódu napojiť na Twitter Api a presmerovať vzorku tweetov na server. Prijatý tweet je vo forme objektu, ktorý je nutné pred preposlaním formátovať do XML dokumentu. Skript pred spustením odošle na server požiadavku pre registrovanie schémy tweetu, ktorá sa nachádza v samostatnom súbore.

Podobne ako je predvedené v Twitter Stream je možné naprogramovať ďalšie gene-

rátory udalostí. Jedinou podmienkou je definovanie schémy (čo je možné aj pomocou administratívneho rozhrania) pred spustením generátora a streamovanie udalostí vo forme XML dokumentov na URL definovanú v administratívnom rozhraní.

### 3.3.2 Contacts Web

Contacts Web je webová aplikácia, zostavená pomocou frameworku Sinatra a Bootstrap. Aplikácia obsahuje prezentačnú a aplikačnú vrstvu, databázová vrstva je riešená pomocou REST volaní na server. Vďaka tomu je možné logiku aplikácie zmestiť do necelých 30 riadkov kódu.

Logika aplikácie pozostáva z obsluhy požiadaviek na vytvorenie nového kontaktu, zmazanie existujúceho a výpis existujúcich kontaktov. Všetky tieto požiadavky sú posielané na server.

## 3.4 Možnosti rozšírenia

Keďže nástroj Esper je aktívne vyvíjaný a obsahuje rozsiahlu funkcionálnu nebolo možné v rámci tejto diplomovej práce pokryť všetky jeho oblasti. V tejto časti zhrniem niektoré z oblastí vhodné k dodatočnej implementácii. Diplomová práca je voľne dostupná z úložiska github, vďaka čomu si môže túto aplikáciu upraviť ktokoľvek podľa svojich požiadaviek. Časti aplikácie vhodné k dodatočnej implementácii sú nasledovné:

**Užívateľské účty** Ako prvú oblasť, ktorú implementuje väčšina aplikácií je oddelenie prístupu rôznych užívateľov. Táto funkcionálna však nie je nevyhnutná, preto bola ponechaná v tejto časti práce. Užívateľské účty predstavujú možnosť prihlásenia a správu užívateľov v administratívnom rozhraní, čo zahŕňa aj rozdelenie práv minimálne na role návštevníka, prihláseného užívateľa a správcu systému. V serverovej časti sa tieto zmeny prejavujú nutným oddelením pracovného prostredia jednotlivých užívateľov. To je možné docieľiť pomocou EP Service Provider v ktorom sa dá vytvoriť samostatné prostredie definovaním unikátnej URL, napríklad zahrnutím emailu užívateľa. Zmeny sa prejavujú aj v databázovej časti, kde bude nutné upraviť štruktúru tabuľky výsledkov.

Spoločne s oddelením pracovného prostredia užívateľov je vhodné implementovať autorizáciu volaní v serverovej časti aplikácie.

**Rozšírenie prístupu k Esperu** Prístup k Esper engine je v tejto verzii limitovaný na správu schém udalostí a príkazov, možnosť spracovania nových udalostí a zobrazenie výsledkov jednotlivých príkazov. Esper však obsahuje viacero prvkov, ktoré sú skryté na pozadí. Vhodným rozšírením v tejto oblasti by bola implementácia možnosti správy premenných a vzorov alebo vytvárania a prehľadu pomenovaných okien alebo tabuliek.

**Definícia typov udalostí v rôznych formátoch** Esper podporuje štyri základné možnosti definície typu udalosti. Pomocou POJO objektu, triedy `java.util.Map`, definovaním poľa objektov a ich typov a XML dokumentom. V tejto aplikácii je implementovaná posledná možnosť, ktorá umožňuje užívateľovi komunikovať priamo s Esperom.

Definícia pomocou XML dokumentu však neumožňuje využitie rozšírenia `EsperIO`, ktorá slúži na spracovanie udalostí z rôznych vstupov, pretože Esper pri definícii schémy vo forme XML dokumentu vyžaduje aj prichádzajúce udalosti v tomto formáte.

**Napojenie na relačnú databázu** Esper umožňuje spracovať údaje z relačnej databázy ako vstupné udalosti. Túto funkcionality by tiež bolo možné sprístupniť v administratívnom rozhraní. Aplikácia by tak získala nový zdroj spracovania dát bez nutnosti programovania a konfigurácie na serverovej časti.

**Štatistiky** Pri rozšírení aplikácie o správu užívateľov je vhodné mať k dispozícii aspoň základné štatistiky využitia Esperu. Je tak možné predísť napríklad zahľteniu systému jediným užívateľom.

Ďalšie využitie štatistík je možné v prípade poskytovania výpočtového výkonu Esperu ako služby. Pri takomto využití by bolo nutné sledovať využitie jednotlivých užívateľov a vznikla by tak možnosť spoplatniť tieto služby.

**Notifikácie** Esper je v mnohých firmách využívaný ako nástroj, ktorý vyhľadáva v prichádzajúcich udalostiach definované vzory a na ich základe vykonáva nejaké akcie. Takouto akciou by mohlo byť odoslanie emailu alebo sms, ktorá by upozornila na výskyt hľadanej udalosti.

Tieto rozšírenia nie sú potrebné pre základnú prácu s Esperom, na čo sa orientuje táto diplomová práca. CEP je možné využiť vo viacerých oblastiach a každá z nich potrebuje špecifické sady nástrojov. Preto by mal byť ďalší vývoj odvodený od konkrétnej potreby využitia tohoto nástroja. Napríklad v prípade reálneho nasadenia je potrebnější notifikačná časť a pre uľahčenie práce programátora rozšírenie prístupu k Esperu.

# Kapitola 4

## Inštalácia & Použitie

Táto kapitola sa bude zaoberať názorným postupom, ktorý je potrebný k spusteniu aplikácie. Postup zahŕňa kroky od prípravy systému, inštalácie aplikácie až po popis administratívneho rozhrania a názorné príklady použitia.

Program	Verzia	Zdroj
Windows 8	Professional N 64-bit	Microsoft DreamSpark
Java JDK	1.8.0_31	<a href="http://www.oracle.com">www.oracle.com</a>
Maven	3.2.3	<a href="http://maven.apache.org">maven.apache.org</a>
Cassandra	2.1.2	<a href="http://Cassandra.apache.org">Cassandra.apache.org</a>
HSQLDB	2.3.2	<a href="http://hsqldb.org">hsqldb.org</a>
DevCenter	1.2.0	<a href="http://www.datastax.com">www.datastax.com</a>
RailsInstaller	3.1.0	<a href="http://railsinstaller.org">railsinstaller.org</a>
Node.js	0.10.31	<a href="http://nodejs.org">nodejs.org</a>

Tabuľka 4.1: Verzie programov použitých pri inštalácii

### 4.1 Príprava systému

Aplikácia je k dispozícii vo forme zdrojových kódov. Pred použitím ju preto musíme skompilovať, nastaviť databázu a premenné prostredia a stiahnuť závislé knižnice. Táto sekcia sa zaoberá prípravou systému a prostredia ku spusteniu aplikácie.

## Operačný systém

Ako operačný systém bol použitý Windows 8 Professional N 64-bit. Táto verzia je pre študentov FIS dostupná prostredníctvom projektu DreamSpark - software spoločnosti Microsoft licencovaného pre akademické inštitúcie. Windows 8 bol nainštalovaný s predvolenými nastaveniami. Vzhľadom na minimálne požiadavky databázového systému Cassandra 2GB RAM je vhodné použiť systém s minimálne 4GB RAM.

## Java

Ako programovací jazyk serverovej časti aplikácie bola použitá java. Demonštračná kompilácia využíva aktuálnu verziu jdk1.8.0\_31. Je samozrejme možné použitie iných verzií, avšak knižnica Esper správne pracuje len s niektorými z nich. Konkrétne nastáva problém pri pridaní schémy definovanej XML dokumentom do Esper konfigurácie, kde vzniká výnimka:

---

```
com.Espertech.Esper.client.ConfigurationException: Failed to read schema  
via URL 'null'
```

---

Výpis 4.1: Výnimka pri definovaní XML schémy v niektorých verziách javy

Testované verzie javy a ich kompatibilita je v nasledujúcej tabuľke:

Verzia	Stav
jdk1.7.0_25	kompatibilná
jdk1.7.0_71	nekompatibilná
jdk1.7.0_72	kompatibilná
jdk1.8.0_25	nekompatibilná
jdk1.8.0_31	kompatibilná

Tabuľka 4.2: Kompatibilita verzií javy s Esper xml schémami

Java bola nainštalovaná použitím predvolených nastavení, súčasťou ktorých je aj inštalácia JRE. Po nainštalovaní definujeme premennú prostredia JAVA\_HOME nasledovne:

---

```
JAVA_HOME – C:\Program Files\Java\jdk1.8.0_31
```

---

Výpis 4.2: Definícia premennej prostredia JAVA\_HOME

Toto nastavenie je vyžadované nástrojom Maven, ktorý popisuje nasledujúca sekcia.



## Maven

Na zostavenie projektu bol použitý Maven vo verzii 3.2.3. Inštalácia pozostáva zo stiahnutia zip archívu (binárnej distribúcie) zo stránky projektu a rozbalenia do cieľového adresára. Pre jednoduchšie použitie je vhodné rozšíriť PATH o cestu k binárnym súborom nástroja, v tomto prípade:

---

```
PATH = %PATH%;C:\Apache\apache-maven-3.2.3\bin
```

---

Výpis 4.3: Rozšírenie premennej prostredia PATH o nástroj Maven

## RubyOnRails

Inštalácia frameworku RubyOnRails na windows platforme ja náročnejšia, pretože obsahuje viacero komponentov, preto použijeme program RailsInstaller. Ten v použitej verzii obsahuje nasledujúce komponenty použité v projekte:

- Ruby 2.1.5 - interpretátor jazyka Ruby
- Rails 4.1 - webový framework pre administratívne rozhranie
- Bundler - manažér závislostí projektu, funkciou podobný nástroju Maven
- Git - verzovací nástroj použitý pre stiahnutie projektu z verejného repozitára
- DevKit - nástroj pre zostrojenie (build) natívnych C/C++ rozšírení na Windowse

Pri inštalácii použijeme predvolené nastavenia. Pri prvom použití nástroja bundler však narazíme na problém s certifikátmi:

---

```
Error:SSL\._connect returned=1 errno=0 state=SSLv3 read server certificate  
B: certificate verify failed.
```

---

Výpis 4.4: Problém s certifikátom v Ruby On Rails

Riešenie pozostáva zo stiahnutia súboru obsahujúceho certifikáty certifikačných autorít. Ten následne sprístupníme pre ruby definovaním premennej prostredia [18].

---

```
ruby "%USERPROFILE%\Desktop\win_fetch_cacerts.rb"  
SSL_CERT_FILE = C:\RailsInstaller\cacert.pem
```

---

Výpis 4.5: Riešenie problému s certifikátmi

Problém s certifikátmi sa týmto vyriešil. Pri vytvorení a spustení prvej aplikácie ale narazíme na ďalšiu výnimku:

---

```
ExecJS::ProgramError – TypeError: Object doesn't support this property or method
```

---

Výpis 4.6: Problém s javascript runtime

Táto je spôsobená nekompatibilitou predvoleného windows javascript runtime a rails prostredia pri spracovaní assets (css a js) súborov. Jedným z riešení tohoto problému je inštalácia platformy Node.js. Pre potreby projektu postačuje predvolená inštalácia.

## 4.2 Vytvorenie databázy

Kód potrebný na vytvorenie databáz je v súbore database.sql v adresári ThesisApi. Pred použitím aplikácie je nutné vytvoriť ako databázu konfigurácie tak databázu udalostí.

### 4.2.1 Cassandra

Cassandra je okrem binárnej verzie distribuovaná aj ako spustiteľný MSI inštalátor vďaka DataStax komunite. Tento spôsob inštalácie je pohodlnejší, avšak aktuálna verzia 2.1.2 pre operačný systém Windows po inštalácii nepracovala správne. Prejavili sa problémy ako chyby pri spustení spôsobené chybnou verzou knižnice jamm použitej pri kompilácii či poruchy Windows agenta.

Pre projekt som kvôli týmto dôvodom použil binárnu distribúciu. Inštalácia tejto verzie je veľmi podobná inštalácii Mavenu, preberanému v predošlej sekcii. Na stránkach projektu stiahneme archív a rozbalíme ho do cieľového adresára. Tiež rozšírime systémovú premennú PATH o cestu k spúšťacím súborom takto:

---

```
PATH = %PATH%;C:\Apache\apache-Cassandra-2.1.2\bin
```

---

Výpis 4.7: Rozšírenie premennej prostredia PATH o Cassandra

Databázu spustíme z adresára bin. Následne stiahneme nástroj DevCenter, ktorý umožní vytvorenie štruktúry databázy. Po spustení sa nástroj pripojí k databáze. Skopírujeme a vykonáme skript pre vytvorenie databázy výsledkov, ktorý nájdeme v priečinku ThesisApi/database.sql.

### 4.2.2 HSQLDB

Súčasťou závislostí definovaných v pom súbore serverovej časti aplikácie je databáza HSQLDB. Pre jej použitím je však nutné vytvoriť tabuľky schém a príkazov. Preto je nutné si zo stránky projektu stiahnuť archív, ktorý obsahuje rozhranie pre prácu s touto databázou.

Stiahnutý archív rozbalíme do ľubovoľného adresára a spustíme `runManagerSwing.bat`, ktorý sa nachádza v adresári `bin`. Z ponuky typu databázy vyberieme `HSQL Database Engine Standalone`, špecifikujeme cestu k dátovému súboru `jdbc:hsqldb:file:hsqldb` a pripojíme sa k databáze.

Následne do panelu pre zadávanie príkazov vložíme skript pre vytvorenie databázy konfigurácie zo súboru `database.sql` a skript spustíme. Týmto krokom sme vytvorili potrebné tabuľky. Následne je nutné v konfigurácii serverovej časti aplikácie nastaviť cestu k databáze. V súbore `database.properties` upravíme cestu k novo vytvorenej databáze, ktorá sa nachádza v adresári `data`. Vykonané kroky sú znázornené vo výpise 4.8.

---

```
Spustiť: c:/hsqldb/bin/runManagerSwing.bat
```

```
Type: HSQL Database Engine Standalone
```

```
URL: jdbc:hsqldb:file:hsqldb
```

```
Vykonať skript pre vytvorenie tabuliek konfigurácie ,  
ktorý nájdeme v súbore ThesisApi/database.sql
```

```
Upraviť resources/database.properties
```

```
jdbc.url=jdbc:hsqldb:file:c:/hsqldb/data/hsqldb;shutdown=true;
```

```
write_delay=false;
```

---

Výpis 4.8: Vytvorenie štruktúry databázy konfigurácie

Týmto krokmi sme pripravili prostredie pre spustenie aplikácie. Podrobný postup spustenia je popísaný v nasledujúcej sekcii.

## 4.3 Spustenie aplikácie

Pred samotným spustením aplikácie je potrebné ju stiahnuť a skompilovať. Ak bol dodržaný postup inštalácie popísaný v predchádzajúcej sekcii tak sú všetky potrebné nástroje k dispozícii. Pokračujeme teda stiahnutím zdrojových kódov kompletného projektu

z git repozitára. V príkazovom riadku:

---

```
git clone https://github.com/kravciak/thesis-diploma-code.git
```

---

#### Výpis 4.9: Stiahnutie zdrojových súborov projektu

Kompletný projekt obsahuje 4 samostatné komponenty. Každá z nich sa spúšťa samostatne, avšak všetky sú závislé na Api a pred použitím je nutné nakonfigurovať schémy a príkazy - na čo slúži komponenta Web. Postup spustenia je preto nasledovný:

### Api

Predtým než je spustená Api je nutné spustiť Cassandra databázu. Tá je spustiteľná z bin priečinka inštalácie súborom Cassandra.bat. Pre spustenie vyžaduje administrátorské oprávnenia. Cassandra je tiež možné nainštalovať ako service pridaním parametra “install”. Pri správnom spustení sa na konzole zobrazí text:

---

```
Starting listening for CQL clients on localhost/127.0.0.1:9042...
Binding thrift service to localhost/127.0.0.1:9160
Listening for thrift clients...
```

---

#### Výpis 4.10: Spustenie databázy výsledkov

Api nájdeme v priečinku ThesisApi. Je závislá na viacerých knižniciach, ktoré sú automaticky stiahnuté nástrojom Maven pri zostavovaní. Po úspešnom zostavení aplikáciu spustíme. Zmienené kroky dosiahneme použitím dvoch príkazov:

---

```
mvn install
mvn exec:java
```

---

#### Výpis 4.11: Spustenie serverovej časti aplikácie

Pri úspešnom spustení Api inicializuje spojenie s databázou výsledkov, spustí webový server Apache a načíta schémy a príkazy z databázy konfigurácie.

### Web

Administračné rozhranie sa nachádza v priečinku ThesisWeb. Podobne ako Api aj Web je závislý na viacerých gemoch. Tie sú stiahnuteľné pomocou bundleru, ktorého inštalácia je popísaná v predchádzajúcej sekcii. Po stiahnutí závislostí môžeme web spustiť.

---

```
bundle install
```

---

```
rails s
```

---

#### Výpis 4.12: Spustenie administračného rozhrania

Po spustení je webové rozhranie dostupné na adrese `http://localhost:3000/`.

### Twitter Stream

Twitter stream závisí na dvoch gemoch, builder a twitter gem. Twitter gem slúži na prijatie vzorky tweetov a builder na transformáciu tweetu do formy xml. Obe nainštalujeme pomocou príkazu `gem install`, ktorý je dostupný ako súčasť ruby.

---

```
gem install twitter builder
ruby generator.rb
```

---

#### Výpis 4.13: Spustenie vzorového streamu udalostí

Kedže twitter stream sa pred spustením preposielania tweetov pripojí na Api (a ak je Api offline vyhlási výnimku) je nutné ich spustiť v tomto poradí.

### Contacts Web

Závislosti contacts webu nainštalujeme rovnako ako v prípade twitter streamu.

---

```
gem install sinatra faraday gyoku json
ruby client.rb
```

---

#### Výpis 4.14: Spustenie vzorovej webovej aplikácie

Pre používanie je potrebné mať spustené Api.

## 4.4 Použitie

### 4.4.1 ThesisApi

Serverová časť aplikácie je prístupná prostredníctvom restovej api počúvajúcej na porte 80. Jej použitie teda spočíva z zavolaní správnej URL so správnymi parametrami. Po spustení je aplikácia predvolene prístupná na adrese `http://localhost:8080/`. Vo výpise 4.15 sú zobrazené príklady volaní, ktoré server akceptuje.

---

```
Výpis prvých 100 schém
http://localhost:8080/schemas?offset=0&limit=100
```

Výpis prvých 100 príkazov

`http://localhost:8080/statements?offset=0&limit=100`

Zobrazenie výsledkov

`http://localhost:8080/statements/53/results`

---

#### Výpis 4.15: Príklad volaní REST API

Niektoré z volaní vracajú súčasne s dátami aj metadáta poskytujúce dodatočné informácie, napríklad o stránkovaní, schéme udalostí či počte výsledkov. Tieto informácie sú spracované na strane grafického rozhrania a zjednodušujú užívateľovi prácu so systémom.

Každá metóda má definovaný spôsob prístupu, teda jedinečnú kombináciu URL a metódy volania (GET, POST, DELETE). Tabuľka 4.3 poskytuje kompletný prehľad mapovania http metódy a URL na funkciu, ktorá obsluhuje dané volanie. Funkcie sú rozdelené do štyroch hlavných skupín podľa oblasti, ktorú obsluhujú. Takisto ich mapovanie zodpovedá konkrétnej oblasti.

Obrázok 4.1 predstavuje parametre funkcií REST rozhrania. Rovnako ako tabuľka 4.3 je rozdelený do štyroch skupín predstavujúcich triedy obsahujúcu konkrétnu funkciu. Ich vzájomnou kombináciou sme schopní identifikovať všetky REST volania, ktoré umožňuje serverová časť aplikácia spracovať.

### 4.4.2 ThesisWeb

Hlavná obrazovka zobrazuje tri oblasti Esperu, ktoré je možné pomocou aplikácie obsluhovať - správu schém a príkazov a formulár pre odosielanie udalostí. Po kliknutí na jednu z nich sa zobrazí obrazovka s rozšíreným menu 4.2, ktoré navyše obsahuje možnosť prechodu na zobrazenie výsledkov nájdených konkrétnym príkazom.

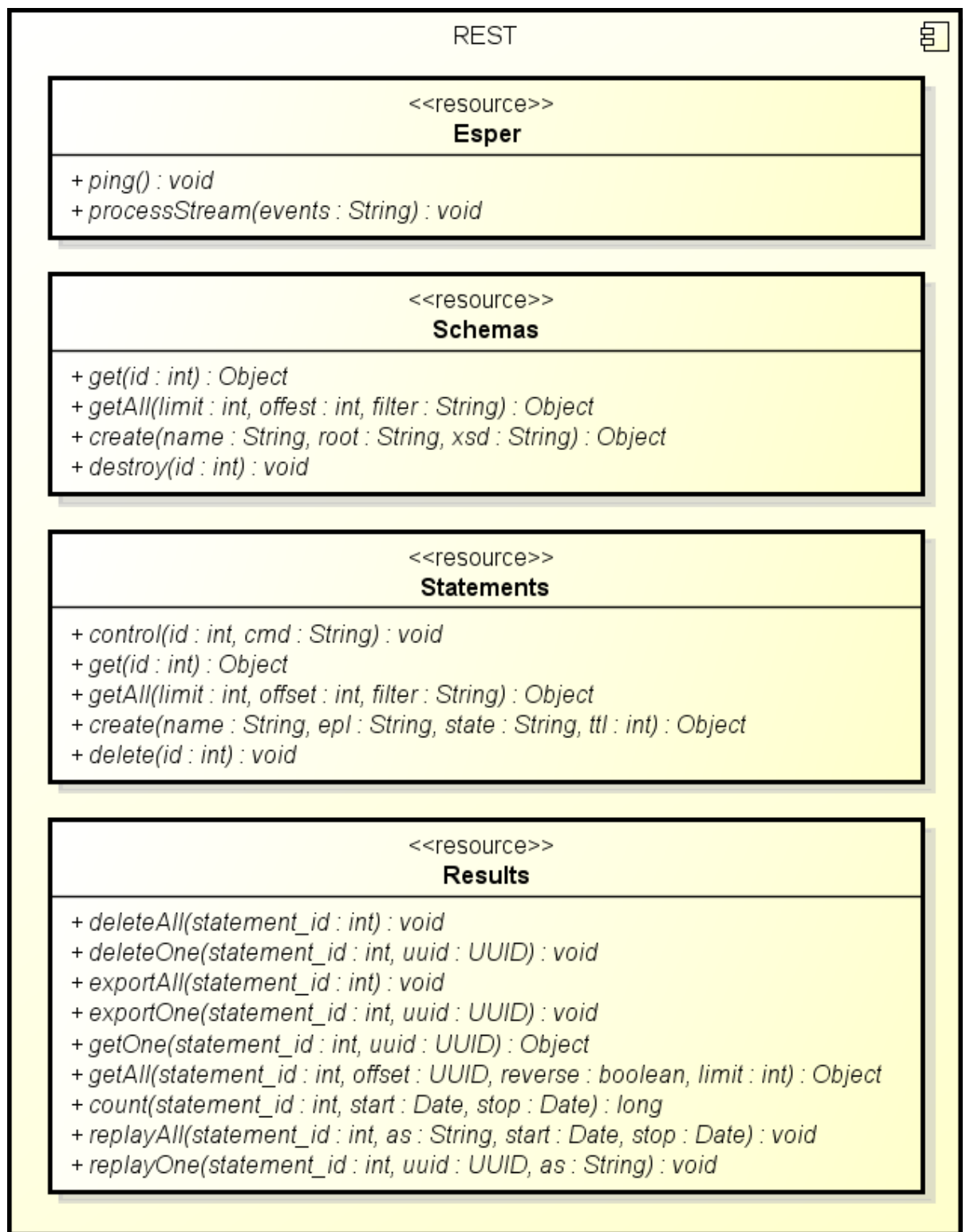
Pod týmto menu sa nachádza navigačný panel s informáciou o ceste k práve zobrazovanej stránke 4.3. Po jeho pravej strane sú ovládacie tlačidlá na pridávanie, mazanie, prípadne export elementov na stránke. Súčasťou navigačného panelu je na stránke schém a príkazov vyhľadávanie, ktoré umožňuje filtrovať pomocou mena schémy alebo príkazu.

Pod navigačným menu sa nachádza samotný obsah stránky. Nasledujúci text popisuje každú zo štyroch hlavných sekcií.

**Schémy:** Na hlavnej obrazovke je možné vidieť zoznam schém a pri niektorých z nich

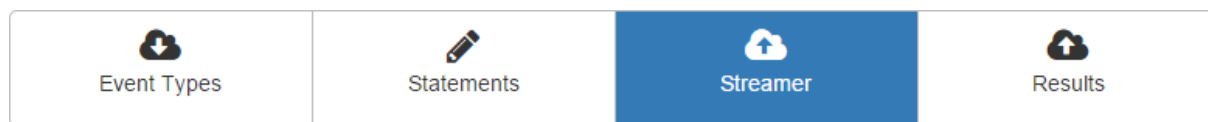
HTTP metóda	Funkcia	URL
GET	ping()	/Esper/ping
POST	processStream()	/Esper/events
GET	get()	/schemas/{id}
GET	getAll()	/schemas
POST	create()	/schemas
DELETE	delete()	/schemas/{id}
GET	control()	/statements/{id}/control
GET	get()	/statements/{id}
GET	getAll()	/statements
POST	create()	/statements
DELETE	delete()	/statements/{id}
DELETE	deleteAll()	/statements/{statement_id}/results
DELETE	deleteOne()	/statements/{statement_id}/results/{uuid}
GET	exportOne()	/statements/{statement_id}/results/{uuid}/export
GET	exportAll()	/statements/{statement_id}/results/export
GET	getOne()	/statements/{statement_id}/results/{uuid}
GET	getAll()	/statements/{statement_id}/results
GET	count()	/statements/{statement_id}/results/count
GET	replayAll()	/statements/{statement_id}/results/replay
GET	replayOne()	/statements/{statement_id}/results/{uuid}/replay

Tabuľka 4.3: Mapovanie funkcií na REST URL

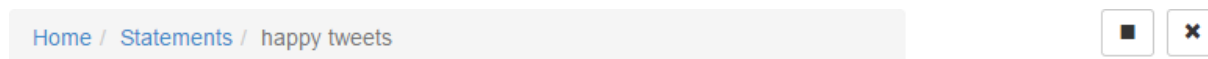


Obr. 4.1: Parametre funkcií REST rozhrania





Obr. 4.2: Menu aplikácie rozšírené o zobrazenie výsledkov



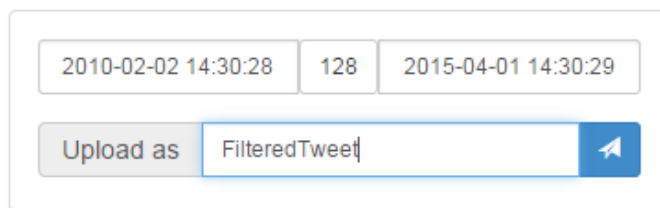
Obr. 4.3: Navigačný panel s ovládacími prvkami

číslo. Toto číslo reprezentuje počet príkazov závislých na konkrétnej schéme. Po rozkliknutí schémy sa zobrazí jej detail. Na ľavej strane sú zobrazené dodatočné informácie vrátane zoznamu príkazov závislých na danej schéme. Na pravej strane obrazovky je konkrétna XML schéma. Pomocou ovládacích tlačidiel je možné túto schému exportovať alebo zmazať v prípade že neobsahuje žiaden na nej závislý príkaz.

Pri vytváraní novej schémy je nutné vyplniť jej meno, koreňový element umožňujúci Esper engine jedinečne identifikovať schému (zhodný s koreňovým elementu definície schémy) a XML definíciu schémy.

**Príkazy:** Podobne ako v prípade schém nájdeme na hlavnej obrazovke zoznam príkazov a pri každom z nich počet udalostí, ktoré mu vyhovujú. Po rozkliknutí sa zobrazí detail príkazu. Na tejto stránke je zaujímavá v popise príkazu na ľavej strane URL adresa, pomocou ktorej sú výsledky dostupné priamo zo serverovej API a odkaz pre zobrazenie výsledkov. Na pravej strane je zobrazená schéma príkazu s menami polí a ich dátovými typmi. Nad ňou je panel pre opätovné preposlanie uložených výsledkov do Esper engine. Pri preposlaní je možné zvoliť si počiatočný a konečný dátum ohraničujúci ktoré výsledky sa majú preposlať a meno udalosti pod ktorým sa majú odoslať. Tlačidlo medzi poliami ohraničujúcimi časový úsek umožňuje spočítať vyhovujúce udalosti. Ovládací panel príkazu umožňuje spustiť, pozastaviť alebo zmazať daný príkaz. Pozastavené príkazy nevyhľadávajú nové udalosti.

Pri vytváraní novej schémy je potrebné zadať jej meno, umožňujúce jedinečne identifikovať príkaz v Esper engine. V prípade že už existuje príkaz s rovnakým menom je na jeho koniec automaticky pridané číslo vo forme “-n”. Voliteľne je možné za-



Obr. 4.4: Formulár preposielania historických udalostí na Esper engine

dať TTL, ktoré špecifikuje ako dlho majú byť výsledky príkazu držané v databáze. Posledné textové pole obsahuje text vytváraného príkazu. Novo vytvorený príkaz je predvolene v spustenom stave.

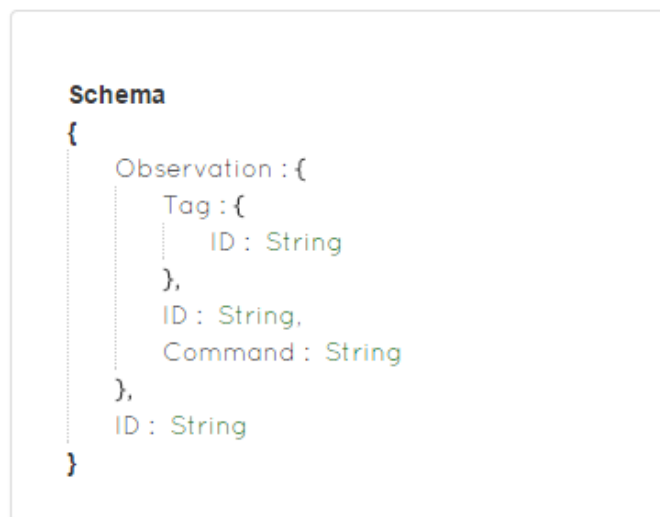
**Streamer:** Pre posielanie nových udalostí existuje viacero spôsobov, jedným z nich je odoslanie udalostí pomocou administratívneho rozhrania na tejto stránke. V hornej časti je URL, ktorú je možné použiť na odosielanie udalostí priamo na API serverovej časti. Štruktúra odosielanej udalosti je bližšie popísaná v použití serverovej časti aplikácie a Esper dokumentácii. Pre odoslanie udalostí je možné použiť formulár alebo udalosti načítať zo súboru. Obe možnosti sa dajú použiť súčasne. Pri načítaní zo súboru sú odosielané udalosti streamované, čo umožňuje posielať veľké objemy dát.

V ovládacích prvkoch stránky nájdeme tlačidlo pre odoslanie vzorky dát. Tieto sú načítané zo súboru a vyhovujú prednastaveným schémam a príkazom. Ich použitie uľahčuje pochopenie fungovania aplikácie.

**Výsledky** Pre zobrazenie výsledkov musíme najprv vybrať príkaz, ktorého výsledky chceme vidieť. Na stránku výsledkov je možné sa dostať z detailu príkazu kliknutím na odkaz s počtom výsledkov alebo na poslednú položku z menu. Na základnej stránke je tabuľka so zoznamom výsledkov. Tieto je možné stránkovať, avšak z dôvodu použitia NoSQL databázy nie je možné preskočiť na konkrétnu stránku. Je to zapríčinené tým, že použitá databáza nepozná klauzulu OFFSET a k výsledkom je tak možné pristupovať len špecifikáciou ich primárneho kľúča, alebo vzťahu k inému primárnemu kľúču. Po rozkliknutí konkrétneho výsledku sa na ľavej strane zobrazí jeho detailný popis, v ktorom je možné vidieť napríklad príkaz ktorému výsledok patrí alebo čas za ktorý expiruje. Pod týmito metadátami je daný výsledok zobrazený v JSON formáte.

Podobne ako na stránke príkazu, kde je možné preposlať všetky výsledky je možné preposlať jeden konkrétny výsledok pomocou panelu v pravej časti stránky detailu výsledku.

Vďaka ovládacím prvkom je možné exportovať alebo zmazať všetky výsledky na hlavnej stránke, alebo konkrétny výsledok pri zobrazení jeho detailu. V navigačnom paneli je zobrazené aj UUID výsledku.



Obr. 4.5: Zobrazenie schémy udalosti s vnorenými typmi

Pre zobrazenie detailu schémy bolo použité javascript rozšírenie pretty-json. Jeho výhodou je možnosť zbaliť a rozbaľiť jednotlivé časti schémy a zároveň prehľadne zobraziť celkovú štruktúru, čo je výhodné predovšetkým pri rozsiahlejších schémach. Príklad schémy s vnorenými typmi je na obrázku 4.5.

Grafické rozhranie je jednoduché, niektoré stránky zámerne obsahujú prázdne miesto - napríklad na zozname schém a príkazov. Tieto časti sú vhodné pre doplnenie funkcionality v prípade dodatočného rozšírenia projektu. Možnosti rozšírenia sú zhrnuté v kapitole popisujúcej návrh aplikácie. Výsledné grafické rozhranie pokrýva základnú funkcionality potrebnú k práci s Esper engine.

# Záver

V úvode tejto práce boli stanovené tri ciele. Pre ich zdokumentovanie bolo nutné čitateľa najprv uviesť do problematiky spracovania komplexných udalostí a práce s nosql databázami, ktorými sa zaoberajú kapitoly 1 a 2.

Po oboznámení čitateľa s použitými technológiami nasleduje kapitola 3, v ktorej je popísaný návrh a implementácia konkrétneho riešenia. Práve v tejto časti sú realizované ciele diplomovej práce.

Prvým z nich bolo vytvorenie administračného rozhrania k Esper engine. Táto časť bola implementovaná pomocou frameworku Ruby On Rails. Administračné rozhranie pre prístup k Esper engine využíva serverovú časť, na ktorú sa napája pomocou restovej api. Toto oddelenie spĺňa podmienku druhého cieľa, a umožňuje vývoj ďalších aplikácií využívajúcich prístup k Esper engine. Serverová časť bola implementovaná v jave za pomoci spring frameworku.

Tretím cieľom práce bolo umožnenie preposielania historických udalostí. Táto časť zadania bola tiež splnená, keďže administračné rozhrania obsahuje v detaile príkazu a udalosti formulár, umožňujúci presmerovať uložené výsledky na vstup Esper engine. Rovnako ako všetky operácie je preposielanie udalostí možné realizovať aj bez administračného rozhrania a to volaním definovanej URL v restovej api.

Databázová časť aplikácie bola rozdelená na dve samostatné časti, jednu SQL databázu, ktorá slúži na uchovávanie konfigurácie serverovej časti aplikácie a druhú NoSQL databázu na ukladanie výsledkov Esperu. Administračné rozhranie nevyužíva žiadne úložisko dát, všetky údaje sú dynamicky získavané pomocou restovej api.

Keďže pri implementácii a inštalácii celkového riešenia som narazil na viacero problémov kompatibility či už s java knižnicami, certifikátmi alebo javascript engine a ich riešenie nepovažujem za triviálne je súčasťou práce aj kapitola zaoberajúca sa inštaláciou, spustením a použitím aplikácie.

Zdrojový kód aplikácie je voľne dostupný z úložiska github, čo umožňuje ľubovoľné úpravy a rozšírenia, napríklad v rámci ďalšej diplomovej práce. Možné rozšírenia vidím v pridaní správy premenných, vzorov, pomenovaných okien, tabuliek či vylepšení grafického rozhrania.

# Literatúra

- [1] Štefan Repček, “Cep portál pro simulaci,” Master’s thesis, Masarykova Univerzita, 2013.
- [2] J. Demo, “Metódy analýzy a návrhu cep aplikácií,” Master’s thesis, Masarykova Univerzita, 2011.
- [3] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. ADDISON WESLEY Publishing Company Incorporated, 2013.
- [4] B. K. Alejandro Buchmann, “Complex event processing.” <http://nirvana.informatik.uni-halle.de/~molitor/webcms/pdf/it0905.pdf>, 2009.
- [5] P. Vincent, “Cep versus esp.” <http://www.tibco.com/blog/2009/08/21/cep-versus-esp-an-essay-or-maybe-a-rant/>, 2009.
- [6] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Pubs Co Series, Manning, 2011.
- [7] P. D. R. Bruns, “Event-driven architecture.” [http://eda.inform.fh-hannover.de/buch/material/M\\_SWA\\_SS12\\_Kap6\\_EDA\\_Teil1.pdf](http://eda.inform.fh-hannover.de/buch/material/M_SWA_SS12_Kap6_EDA_Teil1.pdf), 2015.
- [8] F. B. Michael Eckert, “Complex event processing (cep).” <http://core.ac.uk/download/pdf/12175099.pdf>, 2009.
- [9] COUCHBASE, “What is nosql.” <http://www.couchbase.com/nosql-resources/what-is-no-sql>, 2015.
- [10] A. L. David Schubmehl, “Applying cognitive computing to the challenges of discovery.” [http://www.idc.com/prodserv/custom\\_solutions/download/IDC\\_1779.pdf](http://www.idc.com/prodserv/custom_solutions/download/IDC_1779.pdf), 2014.

- [11] E. Team and E. Inc, “Esper reference.” [http://esper.codehaus.org/esper-5.1.0/doc/reference/en-US/pdf/esper\\_reference.pdf](http://esper.codehaus.org/esper-5.1.0/doc/reference/en-US/pdf/esper_reference.pdf), 2015.
- [12] P. Cassandra, “What is apache cassandra?.” <http://planetcassandra.org/what-is-apache-cassandra/>, 2015.
- [13] I. DataStax, “About apache cassandra.” <http://docs.datastax.com/en/cassandra/2.1/cassandra/gettingStartedCassandraIntro.html>, 2015.
- [14] A. Byde, “Cassandra: Two data centers and great performance.” <http://www.slideshare.net/Dataiversity/thu-1400-bydeandrewcolor>, 2011.
- [15] R. Low, “Counting keys in cassandra.” <http://planetcassandra.org/blog/counting-key-in-cassandra/>, 2013.
- [16] Theo, “Implementing column pagination in cassandra.” <http://architecturalatrocities.com/post/13918146722/implementing-column-pagination-in-cassandra>, 2011.
- [17] T. A. S. Foundation, “Maven – introduction.” <http://maven.apache.org/what-is-maven.html>, 2015.
- [18] F. Nichol, “Download a cacert.pem for railsinstaller.” <https://gist.github.com/fnichol/867550>, 2011.
- [19] Bootstrap, “Css · bootstrap.” <http://getbootstrap.com/css/>, 2015.

# Zoznam obrázkov

1.1	Model spracovania udalostí . . . . .	5
1.2	Škálovanie aplikácie vzhľadom na jej cenu . . . . .	8
2.1	Príklad dátového okna obmedzeného časom a počtom udalostí . . . . .	16
2.2	Škálovanie databázy Cassandra s rastúcou záťažou . . . . .	23
2.3	Porovnanie štruktúry SQL a NoSQL databázy . . . . .	24
2.4	Bootstrap grid systém . . . . .	31
3.1	Diagram komponentov aplikácie . . . . .	33
3.2	Model databázy konfigurácie . . . . .	36
3.3	Model databázy udalostí . . . . .	36
3.4	Model stránkovania bez použitia klauzuly OFFSET . . . . .	38
3.5	UseCase diagram . . . . .	43
4.1	Parametre funkcií REST rozhrania . . . . .	57
4.2	Menu aplikácie . . . . .	58
4.3	Navigačný panel s ovládacími prvkami . . . . .	58
4.4	Formulár preposielania historických udalostí . . . . .	59
4.5	Zobrazenie schémy udalosti . . . . .	60



# Zoznam tabuliek

2.1	Možnosti definície typu udalosti . . . . .	10
4.1	Verzie programov použitých pri inštalácii . . . . .	48
4.2	Kompatibilita verzií javy s Esper xml schémami . . . . .	49
4.3	Mapovanie funkcií na REST URL . . . . .	56