VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

WEBOVÁ APLIKÁCIA PRE GRAFICKÉ ZADÁVANIE A SPÚŠŤANIE SPARK ÚLOH

WEB APPLICATION FOR GRAPHICAL DESCRIPTION AND EXECUTION OF SPARK TASKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JOZEF HMEĽÁR**
AUTHOR

VEDOUCÍ PRÁCE **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/20640/2017/xhmela00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů

Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Hmeľár Jozef, Bc.**

Obor: Počítačové sítě a komunikace

Téma: **Webová aplikace pro grafické zadávání a spouštění Spark úloh**
Web Application for Graphical Description and Execution of Spark Tasks

Kategorie: Uživatelská rozhraní

Pokyny:

1. Seznamte se s prostředím Apache Spark pro distribuované zpracování Big data a s projektem Apache Livy pro vzdálené zadávání a spouštění Spark úloh přes REST rozhraní.
2. Prozkoumejte možnosti a existující projekty pro grafické znázornění úloh zpracování Big data (např. v rámci projektu Apache Hue) a samotných Big data, tedy vstupů a výstupů zmiňovaných úloh.
3. Navrhněte webovou aplikaci s grafickým editorem úloh zpracování Big data, která bude umožňovat interaktivní spuštění úloh prostředí Spark. Umožněte skládání úloh z předdefinovaných komponent a tvorbu takových komponent.
4. Po konzultaci s vedoucím aplikaci implementujte. Definujte typové komponenty úloh zpracování Big data a demonstrujte jejich použití na několika ukázkových úlohách.
5. Výsledek zdokumentujte, zhodnoťte a zveřejněte jako open-source.

Literatura:

- Holden Karau. *Learning Spark*. First edition, 256 pp. ISBN 978-1-449-35862-4
- Databricks Spark Reference Applications. 2017. [https://databricks.gitbooks.io/databricks-spark-reference-applications]
- Hue - Hadoop User Experience - The Apache Hadoop UI. 2017. [http://gethue.com/]
- Apache Livy: An REST Service for Apache Spark. 2017. [https://livy.incubator.apache.org/]

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT

Datum zadání: 1. listopadu 2017

Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto diplomová práca sa zaoberá problematikou spracovania Big data v distribuovanom systéme Apache Spark pomocou nástrojov, ktoré umožňujú vzdialené zadávanie a spúšťanie Spark úloh cez webové rozhranie. Autor v prvej časti opisuje prostredie Apache Spark, v ďalšej sa zameriava na projekt Apache Livy, ktorý ponúka REST API pre spúšťanie Spark úloh. Následne sú predstavené súčasné riešenia, ktoré umožňujú interaktívnu analýzu dát. Autor ďalej popisuje vlastný návrh aplikácie pre interaktívne zadávanie a spúšťanie Spark úloh, pomocou grafovej reprezentácie úlohy. Autor v návrhu popisuje nielen webovú časť aplikácie, ale aj serverovú časť aplikácie. Ďalej implementáciu oboch častí a v neposlednom rade demonštráciu dosiahnutého výsledku na typickej úlohe. Vytvorená aplikácia poskytuje intuitívne rozhranie pre pohodlnú prácu s prostredím Apache Spark, vytváranie vlastných komponent a k tomu aj radu ďalších možností, ktoré sú v dnešnom svete webových aplikácií štandardom.

Abstract

This master's thesis deals with Big data processing in distributed system Apache Spark using tools, which allow remotely entry and execution of Spark tasks through web interface. Author describes the environment of Spark in the first part, in the next he focuses on the Apache Livy project, which offers REST API to run Spark tasks. Contemporary solutions that allow interactive data analysis are presented. Author further describes his own application design for interactive entry and launch of Spark tasks using graph representation of them. Author further describes the web part of the application as well as the server part of the application. In next section author presents the implementation of both parts and, last but not least, the demonstration of the result achieved on a typical task. The created application provides an intuitive interface for comfortable working with the Apache Spark environment, creating custom components, and also a number of other options that are standard in today's web applications.

Kľúčové slová

webová aplikácia, Apache Spark, Apache Livy, REST API, Spark úlohy, big data, analýza dát, Zeppelin, Jupyter, Angular 2, Python

Keywords

web application, Apache Spark, Apache Livy, REST API, Spark tasks, big data, data analysis, Zeppelin, Jupyter, Angular 2, Python

Citácia

HMEĽÁR, Jozef. *Webová aplikácia pre grafické zadávanie a spúšťanie Spark úloh.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Webová aplikácia pre grafické zadávanie a spúšťanie Spark úloh

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána RNDr. Mareka Rychlého, Ph.D. . Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

....
Jozef Hmeľár
21. mája 2018

Poďakovanie

Rád by som sa poďakoval pánovi RNDr. Marekovi Rychlému, Ph.D. za poskytnuté konzultácie, ľudský prístup a odborné vedenie tejto práce.

Obsah

1 Úvod **4**
1.1 Štruktúra práce 4

2 Prostredie Apache Spark **6**
2.1 Popis Apache Spark 6
2.2 Architektúra Apache Spark 7
2.3 Stack Apache Spark 7
2.4 Spark aplikácie 9
2.5 RDD – Resilient Distributed Dataset 10

3 Projekt Apache Livy **12**
3.1 Popis Livy 12
3.2 Architektúra Livy 12
3.3 Základná funkcionalita 13
3.4 Webové služby 14
3.4.1 REST 15
3.4.2 REST serveru Livy 19

4 Súčasné riešenia **21**
4.1 Projekt Apache Zeppelin 21
4.2 Projekt Jupyter 22

5 Návrh aplikácie **25**
5.1 Návrh architektúry 25
5.2 Externé služby 26

6 Návrh webovej aplikácie **27**
6.1 Diagram prípadov použití 27
6.2 Návrh užívateľského rozhrania 27
6.2.1 Prihlasovacia obrazovka 27
6.2.2 Registrácia 29
6.2.3 Hlavná obrazovka 30
6.2.4 Sledovanie stavu 34
6.2.5 Správa profilu 35
6.2.6 Dátové štruktúry 36

7 Návrh serverovej časti aplikácie **39**
7.1 Pre-posielanie požiadaviek 39

7.2 Dátový model 39
7.3 REST API serveru 40

8 Implementácia **42**
8.1 Implementácia webovej časti aplikácie 42
8.1.1 Grafová reprezentácia úlohy 43
8.1.2 Vytváranie dátovej štruktúry 43
8.1.3 Vytvorenie užívateľských komponent 43
8.1.4 Generovanie kódu 44
8.1.5 Pripojenie na Livy server 45
8.2 Implementácia serverovej časti aplikácie 45
8.2.1 Python Flask 46
8.3 Databáza 47
8.4 Autentizácia, autorizácia 48
8.5 Zdieľanie komponent 48

9 Demonštrácia výsledkov **50**
9.1 Wordcount 50
9.1.1 Vytvorenie a spustenie úlohy 50
9.1.2 Uloženie práce 51
9.1.3 Vytvorenie vlastnej komponenty 52
9.1.4 Zmazanie úlohy 52
9.1.5 Zdieľanie komponenty 52

10 Záver **53**

Literatúra **54**

Prílohy **57**
Zoznam príloh 58

A Návrh hlavnej obrazovky **59**

B Dátové štruktúry webovej časti **60**
B.1 Spark komponenta 60
B.2 Užívateľom definovaná komponenta 61

C Manuál **63**

D Obsah priloženého pamäťového média **66**

Zoznam obrázkov

2.1 Spark architektúra (Zdroj: https://jaceklaskowski.gitbooks.io) 8
2.2 Stack Sparku (Zdroj: [14]) 8
2.3 RDD partície na rôznych uzloch (Zdroj: https://www.jbssolutions.com/) . . 11

3.1 Architektúra Livy 13
3.2 Princíp ako funguje REST (Zdroj: https://www.hongkiat.com) 17

4.1 Intepretácia dát pomocou grafu (Zdroj: https://zeppelin.apache.org) 23
4.2 Ukážka vstupu a výstupu v rozhraní Jupyter (Zdroj: http://jupyter.org/) . 24

5.1 Architektúra aplikácie 25
5.2 Architektúra HDFS (Zdroj: [21]) 26

6.1 Diagram prípadov použitia 28
6.2 Pokus o prihlásenie. 29
6.3 Formulár pre registráciu používateľa. 29
6.4 Navigačné menu. 30
6.5 Používateľské menu. 30
6.6 Výber komponent. 31
6.7 Plocha pre vytvorenie Spark úlohy s nastavením Spark komponenty. 32
6.8 Skontrolovaný a vygenerovaný python kód. 32
6.9 Nastavenie užívateľskej komponenty. 33
6.10 Nastavenie komponenty s vlastným kódom. 34
6.11 Modálne okno pri vytvorení užívateľskej komponenty. 34
6.12 Obrazovka sledovania stavu úloh a spojení. 35
6.13 Stránka profilu - zmena hesla. 36
6.14 Stránka profilu – správa používateľových súborov. 36
6.15 Stránka profilu – správa používateľom vytvorených komponent. 37

7.1 Dátový model aplikácie. 40

8.1 Ako funguje WSGI aplikácia (Zdroj: https://www.fullstackpython.com/wsgi-servers.html) 46
8.2 Grafický popis implementácie JWT. (Zdroj: https://jwt.io/) 49

9.1 Spark úloha vytvorená zo Spark komponent 51

A.1 Hlavná obrazovka aplikácie. 59

Kapitola 1

Úvod

Dnešná doba má množstvo prívlastkov. Jedným z nich je prívlastok informačná. Niet sa čomu čudovať, keď takmer každý z nás vlastní moderné zariadenie, ktoré je schopné nejakým spôsobom prijímať, vysielať, spracovávať informácie. S trendom zvyšujúceho sa výkonu integrovaných obvodov a klesajúcou cenou za internetové pripojenie, sme dospeli do bodu, kedy denne dokážeme internetom pretlačiť niekoľko terabajtov dát.

Čísla hovoria za všetko, podľa [26] každú minútu sa na sociálne siete zaregistruje 840 používateľov, používatelia YouTube sledujú 4,146,600 videí každú minútu, používatelia Instagramu nahrajú 46,740 miliónov nových príspevkov každú minútu, 4,3 miliárd správ sa pošle denne cez Facebook Messenger atď., väčšina z nás používa emaily, internetové bankovníctvo, nakupovanie cez internet a podobne.

Celkovo sa odhaduje, že v digitálnom svete existuje momentálne niečo cez 2,7 zettabajtov dát (1 ZB $= 10^9$ TB $= 10^{21}$ B), číslo samozrejme rastie každú sekundu, pričom sa odhaduje, že do roku 2025 ich bude 180 zettabajtov.

Celé toto viedlo k vzniku nového fenoménu „Big data“. Podľa [10] by sa Big data dali definovať ako výsledok zhromažďovania informácií na vysokej úrovni granularity. Ide v podstate o to, čo dokážete získať zo systému, ktorý zostrojíte a uchováte všetky dáta, ktoré ste schopný z vášho systému zozbierať. Uchovávaniu týchto dát pomohol aj fakt, že cena uloženia jedného terabajtu dát klesne zhruba dvojnásobne za štrnásť mesiacov, čo znamená, že uloženie veľkého množstva dát nie je ani pre malé spoločnosti príliš finančne náročné.

V dôsledku toho vznikol nový obor `Data Science` [22], ktorý sa zaoberá analýzou týchto dát. Dátový inžinieri vo svojej práci kombinujú štatistiku, matematiku, programovanie, riešenie problémov, zber dát, schopnosť pozerať na veci inak kvôli hľadaniu vzorov v dátach, spolu s aktivitami ako filtrácia, príprava a zarovnanie dát.

Konečným výsledkom je svet, v ktorom sa zhromažďovanie údajov stalo, hlavne pre business sféru, mimoriadne dôležité [9]. Niektoré organizácie považujú za nedbanlivé, keď nezaznamenávajú údaje, ktoré by mali mať pre ich podnikanie význam.

Je na mieste konštatovať, že softvér navrhnutý tradičnými programovacími technikami nebude dostatočný pre spracovanie takýchto dát. Tým vzniká potreba pre nové programovacie modely a nástroje, ktoré dokážu poskytnúť potrebné výsledky v reálnom čase.

1.1 Štruktúra práce

Táto práca sa zaoberá novými technológiami pre spracovanie Big data v distribuovaných systémoch. V druhej kapitole 2 je predstavený systém Apache Spark, ktorý predstavuje

systém pre distribuované výpočty. Sú predstavené jeho časti 2.3, ako funguje a aké rozhranie poskytuje. Ďalej je uvedený server Livy 3, ktorý umožňuje vzdialene cez REST API spúšťať Spark úlohy.

V kapitole 4 sú spomenuté niektoré zo súčasných nástrojov, ktoré využívajú ako Livy tak Spark, pre zadávanie úloh a získavanie výsledkov.

V kapitole 5 je navrhnutá architektúra aplikácie so všetkými jej časťami, ktorá umožní interaktívne vytváranie a spúšťanie Spark úloh z vopred definovaných komponent, poprípade užívateľom vytvorených komponent. Ďalej je v časti 6 vytvorený diagram prípadov použitia na základe ktorého sú potom navrhnuté jednotlivé časti webovej časti aplikácie. Z diagramu prípadu použitia a taktiež dátových požiadaviek na webovú aplikáciu je v časti 7 navrhnutá serverová časť aplikácie.

Následne sú v časti 8 popísané ako boli implementované jednotlivé časti, použité technológie a podobne.

V časti 9 je popísané vytvorenie Spark úlohy v implementovanom nástroji, ktoré je sprevádzané video ukážkou, ktorá je dostupná na priloženom DVD.

V závere 10 sú zhrnuté výsledky tejto práce, čo bolo potrebné k vytvoreniu aplikácie, spomenuté sú aj obmedzenia, ktoré sa objavili neskôr vo vývoji a sú popísané aj ďalšie vylepšenia, kam by sa mohla práca posunúť.

Kapitola 2

Prostredie Apache Spark

Táto kapitola predstaví čitateľovi projekt Apache Spark, jeho hlavné ciele, výhody oproti predchádzajúcim projektom. Ďalej je opísaná architektúra, popísané jednotlivé komponenty (stack), ktoré Spark tvoria a v neposlednom rade ukážka toho, pre koho je Spark určený a za akým účelom ho je možné použiť.

2.1 Popis Apache Spark

Jedná sa o platformu na distribuované výpočty v clustroch, navrhnutú pre rýchle a obecné použitie. Spark vznikol ako následník populárneho *MapReduce*. Narozdiel od neho, je pre Spark typické spracovanie výpočtov v pamäti. Spark je však rýchlejší aj pri komplexnejších úlohách, ktoré vyžadujú na spracovanie aj pevný disk.

Spark ponúka efektívnejšiu podporu pre rôzne typy úloh. Bol navrhnutý tak aby pokrýval široké spektrum spracovania, ktoré si predtým vyžadovalo ďalšie distribuované systémy. Ponúka tak dávkové spracovanie, interaktívne algoritmy, interaktívne query a streamové spracovanie.

Tieto atribúty, ktoré ponúka jeden engine z neho robia jednoduché a lacné riešenie pri kombinovaní rôznych procesov spracovania, ktoré sú často potrebné v produkčných dátových analýzach. Okrem toho sa znižuje administratívne zaťaženie pri udržiavaní samotných nástrojov. Pre podnik je lacnejšie udržiavať chod jedného distribuované systému ako niekoľko ďalších. Nehľadiac na fakt, že každý z nich by vyžadoval prítomnosť odborníka na daný systém, čo stojí ďalšie finančné prostriedky.

Spark je navrhnutý tak, aby bol vysoko dostupný. Ponúka jednoduché rozhranie pre Python, Javu, Scalu, SQL a ďalšie vstavané knižnice. Je taktiež úzko integrovaný s ďalšími nástrojmi na spracovanie Big Data, napríklad Spark môžeme spustiť v Hadoop clustri a pristupovať k rôznym Hadoop dátovým zdrojom, vrátane databázy Cassandra.

Komponenty prostredia

Spark tvorí niekoľko úzko integrovaných komponent. Jadrom Sparku je „výpočtový engine“, ktorý sa stará o plánovanie, distribúciu a monitorovanie aplikácií pozostávajúcich z množstva výpočtových úloh cez množstvo uzlov, výpočtových strojov a výpočtových clustrov.

Pretože výpočtové jadro Sparku je rýchle a všeobecné, hlavným účelom je napájanie viacerých komponent na rôzne úlohy, ako napríklad SQL alebo strojové učenie. Tieto komponenty sú navrhnuté tak, aby mohli vzájomne spolupracovať a aby ich užívateľ mohol v programe používať ako knižnice.

Filozofia jednotného stack-u má niekoľko výhod. Prvou je, že všetky knižnice a vyššie komponenty ťažia zo zlepšenia nižšej vrstvy. Napríklad, ak Spark jadro pridá nejakú optimalizáciu, automaticky sa zvýši rýchlosť knižníc SQL a knižníc pre strojové učenie.

Druhou výhodou je, že náklady spojené so spustením jedného stack-u sú minimálne, oproti spustení 5–10 nezávislých. Tieto náklady zahŕňajú nasadenie, údržbu, podporu a ďalšie. To tiež znamená, že ak Spark pridá novú komponentu do stack-u, tak každá organizácia, ktorá využíva Spark bude schopná si tú komponentu hneď vyskúšať. Tým sa menia náklady na vyskúšanie nového typu analýzy údajov od sťahovania, nasadenia a učenia nového softvérového projektu pre vylepšenie Sparku.

Hlavnou výhodou úzkej integrácie je vytvorenie aplikácií, ktoré bezproblémovo kombinujú rôzne modely spracovania. Napríklad, v Sparku môžeme napísať aplikáciu, ktorá bude využívať strojové učenie pre klasifikáciu dát v reálnom čase, ktoré sa načítavajú prúdovo (stream). Súčasne sa môže analytik pýtať na výsledky, taktiež v reálnom čase pomocou SQL. Navyše, sofistikovaní dátový inžinieri môžu pristúpiť k rovnakým dátam cez Python shell pre ad-hoc analýzu. Ďalší môžu spracovať dáta v samostatnej aplikácií.

2.2 Architektúra Apache Spark

Spark využíva architektúru *master/worker*. *Driver* vyberie master uzol, ktorý sa stará o workerov, v ktorých bežia *executors*. Executor je distribuovaný agent, ktorý zodpovedá za vykonávanie úloh [18].

Driver a executors bežia vo vlastných Java procesoch. Môžu sa spustiť všetky na jednom uzle (*horizontálny cluster*) alebo na oddelených uzloch (*vertical cluster*). Obrázok 2.1 popisuje architektúru Sparku.

2.3 Stack Apache Spark

V časti 2.1 boli popísané výhody úzkej integrácie jednotlivých Spark komponent a jadra Sparku. Ďalej sú bližšie popísané jednotlivé komponenty stack-u podľa obrázka 2.2. Jedná sa o základné komponenty, nie sú zahrnuté všetky.

Spark Core

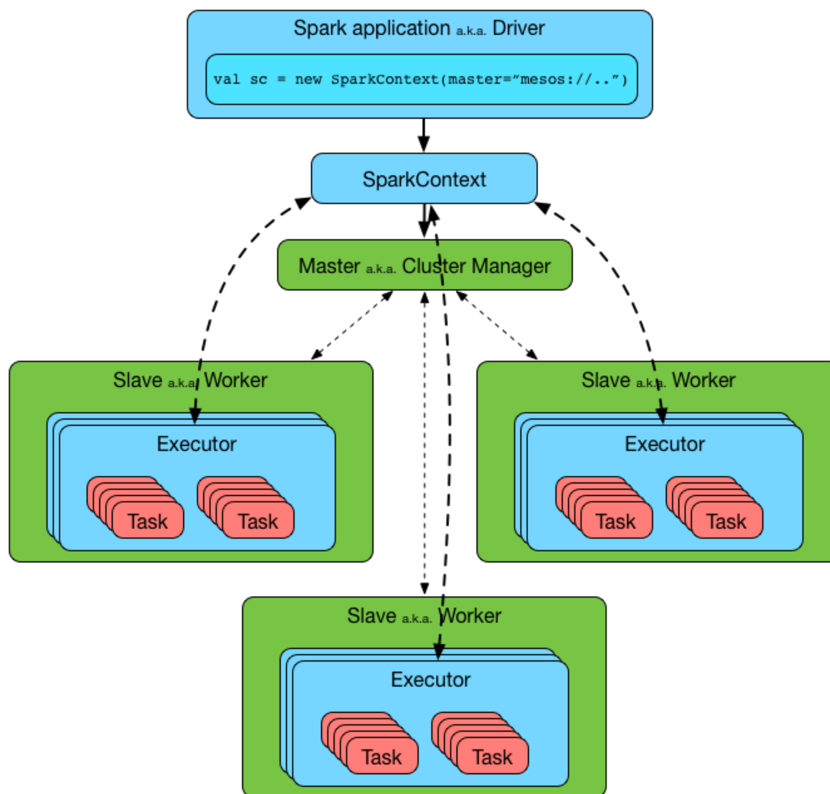
Spark Core obsahuje základnú funkcionalitu Sparku, ktorá zahŕňa komponentu pre plánovanie úloh, manažment pamäte, opravu chýb, komunikácia s úložným systémom a ďalšie. Spark Core je tiež domovom API, ktoré definuje Resilient Distributed Datasets (RDD) 2.5, čo je hlavná programová abstrakcia. RDD reprezentuje zbierku položiek distribuovanú v mnohých výpočtových uzloch, ktoré môžu byť paralelne manipulované. Spark Core poskytuje mnoho API volaní pre vývoj a manipuláciu týchto kolekcií.

Spark SQL

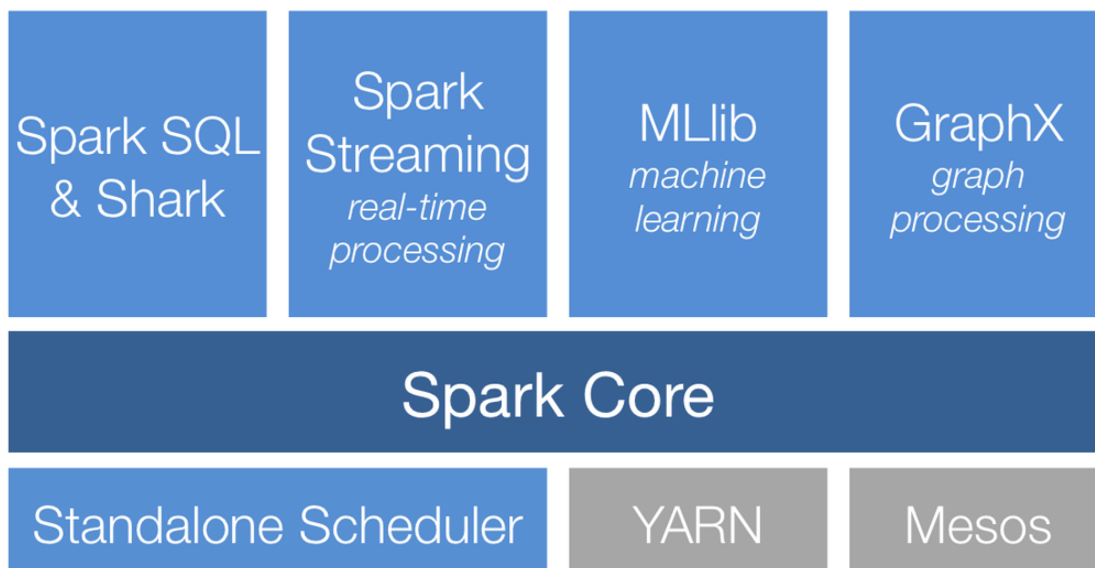
Spark SQL poskytuje podporu pre interakcie medzi Sparkom cez SQL, rovnako ako Apache Hive varianta SQL, nazývaná tiež *Hive Query Language* (HiveQL [1]).

Spark SQL reprezentuje databázové tabuľky ako Spark RDD a prekladá SQL dotazy na Spark operácie. Okrem poskytnutia rozhrania pre Spark, umožňuje vývojárom manipulovať

[1]https://docs.treasuredata.com/articles/hive



Obr. 2.1: Spark architektúra (Zdroj: https://jaceklaskowski.gitbooks.io)



Obr. 2.2: Stack Sparku (Zdroj: [14])

s dátami programovo pomocou RDD v Pythone, Jave a Scale, všetko v rámci jednej aplikácie. Táto úzka integrácia s bohatým výpočtovým prostredím Sparku, tvorí hlavný rozdiel medzi inými open-source nástrojmi pre distribuované spracovanie dát.

Spark Streaming

Spark Streaming je komponenta, ktorá poskytuje prúdové spracovanie dát. Príkladom môžu byť logy generované z produkčného webového servera alebo fronty správ, ktoré obsahujú aktualizácie stavu uverejnené používateľmi webovej služby.

Spark Streaming poskytuje API volania pre manipuláciu s prúdovými dátami, ktoré sú podobné Spark Core RDD volaniam. To uľahčuje programátorom učenie, pre vývoj aplikácií v tomto prostredí a migráciu medzi aplikáciami, ktoré manipulujú s dátami uloženými v pamäti alebo na disku.

Rozhranie Spark Streaming poskytuje rovnaký stupeň chybovej tolerancie (*fault tolerance*), priepustnosti (*throughput*) a škálovateľnosti (*scalability*) ako Spark Core.

MLlib

Spark poskytuje knižnicu *MLlib* pre jednoduchý *machine learning* (ML). MLlib poskytuje niekoľko typov *machine learning* algoritmov, vrátane binárnej klasifikácie, regresie, clustering atď., rovnako ako podpora vyhodnotenia modelov a import dát.

Tak isto poskytuje niektoré nízko úrovňové ML primitíva vrátane generického hľadania. Všetky tieto metódy sú navrhnuté tak, aby sa distribuovali cez cluster.

GraphX

GraphX je knižnica, ktorá poskytuje API pre manipuláciu s grafmi, vykonávanie grafových výpočtov paralelne. Napríklad autori [16] využili túto knižnicu pre zostrojenie grafu priateľov na sociálnych sieťach. Tak ako Spark Streaming a Spark SQL, GraphX rozširuje Spark RDD API, dovoľuje vytvoriť graf priamo s ľubovoľnými vlastnosťami jednotlivých uzlov a hrán grafu. GraphX disponuje aj rôznymi grafovými algoritmami, z ktorých mnohé sú pridané od samotných používateľov [4].

Cluster Managers

Spark je navrhnutý tak, aby sa jednoducho zväčšoval z jedného až na niekoľko tisíc výpočtových uzlov. Ak chceme aby Spark dosiahol maximálnu flexibilitu, môže byť spustený cez rôznych cluster manažérov, ako je napríklad *Hadoop YARN* [2], *Apache Mesos* [3] a tiež jednoduchý cluster manažér, ktorý je implementovaný v Sparku samotnom, *Standalone scheduler* [4], ktorý predstavuje jednoduchšiu cestu ako začať.

2.4 Spark aplikácie

Každá Spark aplikácia začína vytvorením *SparkContext-u*. Bez vytvorenia kontextu nie je možné začať vykonávať Spark úlohu. Môžeme povedať, že Spark aplikácie sú inštancie Spark contextu, inými slovami Spark context predstavuje Spark aplikáciu. Každá aplikácia je jednoznačne identifikovaná na základe *applicationId* a *applicationAttemptIds*.

Vytvorenie Spark aplikácie môžeme zhrnúť do týchto piatich bodov:

1. Vytvorenie Spark konfigurácie

[2] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[3] http://mesos.apache.org/
[4] https://spark.apache.org/docs/1.2.0/job-scheduling.html

2. Pripojenie na Master URL
3. Vytvorenie Spark kontextu
4. Vytvorenie RDD
5. Vykonanie samotnej aplikácie

Ukážka kódu 2.1 jednoduchej Spark aplikácie znázorňuje algoritmus pre výpočet počtu slov v súbore.

```
from pyspark.streaming import StreamingContext
from pyspark import SparkContext
from pyspark import SparkConf

# configuration (1)
APP_NAME = 'words count with python lambda expression'
conf = SparkConf().setAppName(APP_NAME)
# connect to Master (2)
conf = conf.setMaster('localhost')
# create Spark Context (3)
sc = SparkContext(conf=conf)
# create RDD (4)
lines = sc.textFile("hdfs://...")
# execute count (5)
words = lines.flatMap(lambda x: x.split(' '))
pairs = words.map(lambda x: (x,1))
count = pairs.reduceByKey(lambda x,y: x+y)
count.saveAsTextFile("hdfs://...")
```

Výpis 2.1: Aplikácia WordCount v Pythone

2.5 RDD – Resilient Distributed Dataset

Resilient Distributed Dataset (RDD), je hlavnou dátovou abstrakciou v Sparku. Základ tohto konceptu bol vytvorený na univerzite v Berkley [20]. Autori definujú RDD ako abstrakciu distribuovanej pamäte, ktorá umožňuje programátorom vykonávať výpočty vo veľkých clustroch spôsobom tolerantným voči chybám.

Vlastnostni RDD sú:

- `Resilient` — pružnosť, odolnosť voči chybám (*fault–tolerant*), RDD je schopné dopočítať chybné alebo chýbajúce partície.
- `Distributed` — dáta sú distribuované na viacerých uzloch v clustri.
- `Dataset` — jedná sa o kolekciu particiovaných dát s primitívnymi hodnotami alebo hodnotami hodnôt, napríklad n-tice alebo iné objekty.

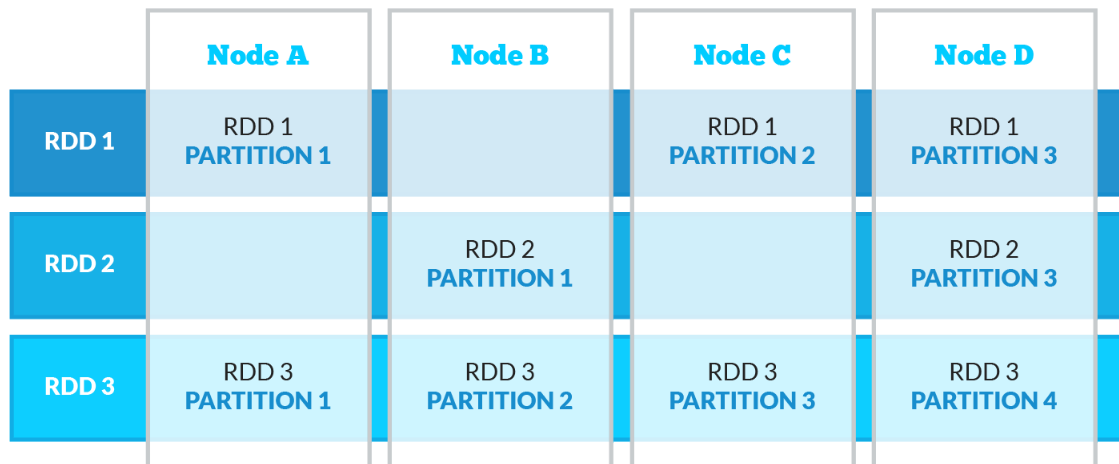
Okrem týchto vlastností má RDD ďalšie, ktoré robia z tohto konceptu výkonné riešenie pre distribuované výpočty:

- `In-Memory` — dáta, ktoré nesie RDD sú uložené v pamäti tak dlho, ako je možné
- `Immutable` alebo *Read-Only* — dáta sú nemenné, zostávajú také, aké boli pri vytvorení RDD. Môžu byť však transformované pomocou transformačných funkcií.

- `Lazy evaluated` — dáta v RDD nie sú prístupné, až kým nie je ukončená úloha, ktorá vykonávanie spustila.
- `Cacheable` — dáta môžu byť trvalo uchované v pamäti, alebo na disku (neodporúča sa kvôli rýchlosti).
- `Parallel` — umožňuje spracovávať dáta paralelne.
- `Typed` — RDD záznamy sú typované, napríklad *long* v *RDD[long]* alebo *(Int, String)* v *RDD[(Int, String)]*
- `Partitioned` — záznamy sú particiované a distribuované na uzly v clustri.
- `Location-Stickiness` — môže definovať predvoľby umiestnenia na výpočet particií (čo najbližšie k záznamom).

Výpočet particií v RDD je navrhnutý ako distribuovaný proces na dosiahnutie rovnomerného rozloženia ako aj využitia dátovej lokality (nie ako v distribuovaných systémoch ako HDFS a Cassandra, kde je daný fixný počet particií). Každá partícia pozostáva zo záznamov.

Partície sú jednotkami paralelizmu. Programátor môže kontrolovať počet particií v RDD pomocou transformácií. Spark sa snaží byť čo najbližšie k záznamom ako je možné, preto vytvára toľko particií, koľko je potrebných. Tým sa optimalizuje prístup k dátam. Vzniká tzv. *one-to-one* mapovanie medzi fyzickými dátami a distribuovaným dátovým úložiskom. Rozdelené RDD na partície, ktoré sú rozdistribuované medzi jednotlivé uzly sú znázornené na obrázku 2.3.



Obr. 2.3: RDD partície na rôznych uzloch (Zdroj: https://www.jbssolutions.com/)

Kapitola 3

Projekt Apache Livy

Hlavnou motiváciou vzniku projektu Livy bolo zredukovať počet úkonov pre vytvorenie Spark aplikácie, čo najviac ako je možné [2]. Aby programátor nemusel riešiť celý mechanizmus zadávanie úlohy, správu Spark kontextu a získavanie výsledkov ale zameral sa len na aplikačnú logiku a tým sa urýchlil proces vývoja. Toto všetko však musí byť splnené za predpokladu, že celé API Sparku je stále dostupné.

3.1 Popis Livy

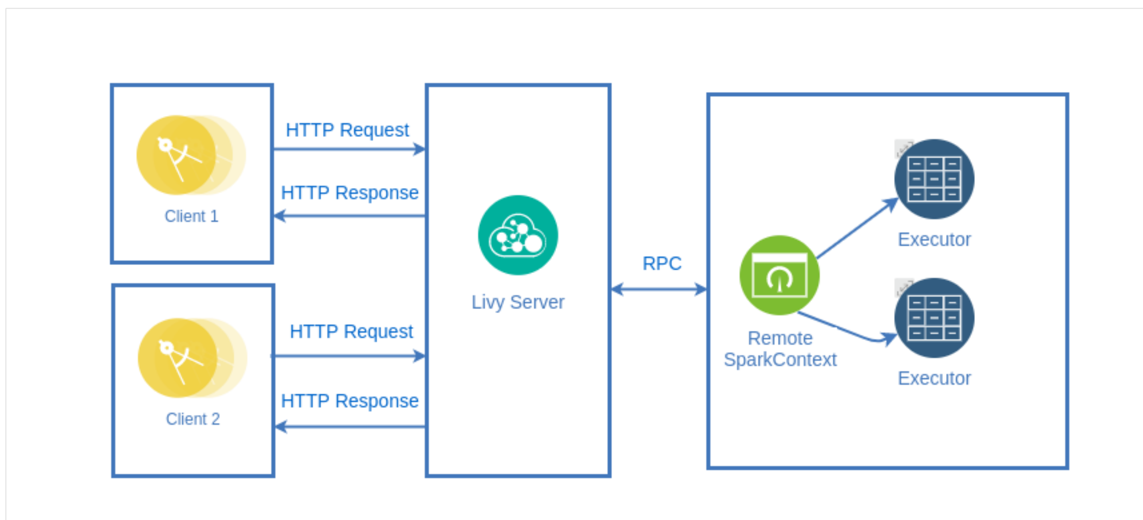
Livy je *open-source REST* rozhranie pre komunikáciu s Apache Spark z akéhokoľvek miesta [2]. Podporuje vykonávanie časti kódu alebo celých aplikácií v Spark kontexte lokálne, alebo v Apache Hadoop YARN. Livy má nasledujúce vlastnosti:

- Interaktívny Scala, Python a R *shell*.
- Dávkové zadávanie úloh pomocou programovacích jazykov Scala, Python, Java.
- Niekoľko používateľov môže zdieľať jeden server.
- Zadávanie Spark úloh cez REST z akéhokoľvek miesta.
- Nevyžaduje žiadnu zmenu v súčasných zdrojových kódoch Spark aplikácií.

3.2 Architektúra Livy

Architektúru popisuje diagram 3.1. Klientská časť vytvorí pri inicializácií vzdialený Spark kontext a pošle úlohu na Livy [27]. Livy server odbaví požiadavok s tým, že obalí úlohu a pošle ju do Sparku. Na strane Livy sa nevykonáva žiadna úloha, ktorú zadal programátor, všetko rieši Spark. Na výsledok sa čaká synchrónne alebo asynchrónne, záleží na klientovi. Aké sú možnosti zadania úlohy cez Livy do Sparku si popíšeme v časti 3.3.

Z diagramu architektúry 3.1 je vidieť, že ak Livy stratí spojenie so Sparkom, všetky pripojené clustre ostanú nedostupné, to znamená, že užívateľ stratí prehľad o spustených úlohách a dočasných dátach. Livy však tento stav rieši svojím mechanizmom pred obnovu sessions. Po reštarte servera sa dokáže pripojiť na užívateľom vytvorené spojenia a vrátiť ho do stavu pred pádom servera.



Obr. 3.1: Architektúra Livy

3.3 Základná funkcionalita

Livy poskytuje svoje zabezpečenie *multi-user* prostredia. Okrem autentizácie pomocou serveru Kerberos [1] poskytuje aj šifrovanie.

REST API je zabezpečené SPNEGO autentizáciou, ktorá vyžaduje od používateľa overenie voči Kerberos serveru. Aj Livy server sa autentizuje sám seba voči serveru Kerberos. RPC komunikácia medzi Livy serverom a Spark kontextom je šifrovaná pomocou SASL.

Okrem toho Livy poskytuje aj autorizáciu používateľa, čo znamená, že na základe užívateľských práv vie prijať alebo odmietnuť request od používateľa.

Livy ponúka tri spôsoby spúšťania Spark aplikácií. Jednotlivý spôsoby budú popísané v ďalších častiach.

Použitie programového API

Livy poskytuje podobné rozhranie[2], ktoré ponúka samotný Spark. Existujú však dva hlavné rozdiely medzi týmito rozhraniamia a to:

1. Keď používame Spark API, vstupný bod, teda Spark kontext vytvára užívateľ sám, kde to pri písaní kódu s API Livy sa o vytvorenie kontextu stará Livy sám.

2. Kód je poslaný cez REST API z klienta na Livy server, ten zase pošle kód na Spark cluster, kde bude vykonaný.

Programové API je možné používať v Jave, Scale a Python. Programové API slúži ako obal pre REST rozhranie. Programátor nerieši ako dáta poslať alebo získať späť zo servera. O tom ako komunikovať s Livy priamo cez REST si povieme v ďalšej časti.

[1] https://databricks.com/session/secured-kerberos-based-spark-notebook-for-data-science
[2] https://livy.incubator.apache.org/docs/latest/programmatic-api.html

Interaktívny session pomocou REST API

Spustenie interaktívnej session cez REST API je podobné tomu, ktoré ponúka Spark ako Spark-shell alebo PySpark. Rozdiel je však v tom, že shell nebeží lokálne, ale namiesto toho beží v clustri a dáta sa prenášajú pomocou siete.

Ako vyzerá komunikácia s Livy pomocou REST API je znázornené na ukážke 3.1

```
import json, requests
host = 'http://localhost:8998' # local livy server
data = {'kind': 'python'} # type of session
headers = {'Content-Type': 'application/json'} # set headers
r = requests.post(host + '/sessions', data=json.dumps(data), headers=headers) # POST data to Livy for
      session start
r.json()
# response from Livy server
{u'state': u'starting', u'id': 0, u'kind': u'spark'}

# session URL, entry point to Livy
session_url = host + r.headers['location']
# statement URL, now is possible send and run code to Spark via Livy
statements_url = session_url + '/statements'

# easy code for Spark
data = {'code': 'print 1 + 1'}
# send code to execute
r = requests.post(statements_url, data=json.dumps(data), headers=headers)
# response from Livy with result from Spark plus information about state
{u'id': 1,
u'output': {u'data': {u'text/plain': u'2'}, # result
          u'execution_count': 1,
          u'status': u'ok'},
u'state': u'running'}
```

Výpis 3.1: Komunikácia s Livy pomocou REST API

Spúšťanie Spark aplikácie cez REST API

Spark poskytuje *spark-submit* príkaz pre spúšťanie Spark aplikácií. Livy poskytuje ekvivalentné volanie cez REST 3.4.1 API, ktoré spustí Spark aplikáciu v clustri. Svojím spôsobom ide o podobný princíp ako bol popísaný v časti 3.3 s tým rozdielom, že v JSON dátach, ktoré sa posielajú cez REST programátor špecifikuje, ktorú triedu (Java) alebo skript (Python) sa má spustiť [1], poprípade špecifikuje ďalšie parametre ako maximálne množstvo pamäte, ktoré môže aplikácia využiť, počet uzlov, poslať ďalšie parametre a podobne. Viac položiek špecifikuje dokumentácia [3].

3.4 Webové služby

Aby bolo možné spúšťať aplikácie vzdialene, je potrebné definovať rozhranie, cez ktoré budú jednotlivé celky vzájomne komunikovať. Jedným z takýchto rozhraní sú aj webové služby.

Existujú dva konkurenčné architektonické štýly pri budovaní webových služieb: REST[4] a SOAP[5]. Oba spôsoby majú oddelené základne nasledovateľov [24], ale mnohé rozdiely

[3] https://livy.incubator.apache.org/docs/latest/rest-api.html#get-batches
[4] Representational State Transfer
[5] Simple Object Access Protocol

medzi nimi sú skôr ideologické ako praktické. S cieľom podporovať zdravý rast webových služieb, je dôležité rozlišovať argumenty pre implementačné postupy nad abstraktnými konceptmi reprezentovanými týmito štýlmi a dôkladne vyhodnocovať výhody RESTful a SOAP.

Podstata týchto rozdielov je dôležitá pre vývoj podnikových systémov, pretože v tejto oblasti preferovali dodávatelia nástrojov SOAP spôsob so zanedbaním RESTful riešenia. Ide o to, že REST je len akýsi súhrn pravidiel a obmedzení, SOAP je skutočným protokolom. Architektúry, ktoré využívajú SOAP sú komplexnejšie, vyžaduje si to však viac úkonov pre odoslanie/spracovanie, oproti RESTu. SOAP využíva pre transport dát formát XML, ktorý je oproti JSON formátu, ktorý využíva REST, pamäťovo náročnejší, čím sa zvyšuje aj čas spracovania správy.

Vzhľadom k tomu, že Livy server poskytuje webové rozhranie pomocou RESTu, bude ďalej opísané ako sa definuje práve toto rozhranie a ďalej aj samotné REST rozhranie, ktoré poskytuje Livy.

3.4.1 REST

Aplikačné rozhranie REST *Representational State Transfer* prezentoval vo svojej dizertačnej práci [11] študent Kalifornskej Univerzity Roy Fielding vo už v roku 2000.

V rámci svojej pracovnej skupiny mal za úlohu definovať existujúci HTTP/1.0[6] navrhnúť rozšírenia pre verziu 1.1 a URI[7], autor uznal potrebu nového modelu ako by mal WWW[8] fungovať.

Takto vznikla myšlienka pre model interakcií, v rámci celkovej webovej aplikácie, nazývaný REST a stal sa základom modernej webovej architektúry, poskytujúc hlavné princípy, pomocou ktorých by bolo možné identifikovať chyby v existujúcej architektúre a rozšírenia architektúry validovať pred samotným nasadením.

Kľúčovými vlastnosťami RESTu sú:

- `Výkonnosť` — interakcia medzi komponentami môže byť dominantným faktorom určujúcim vnímanie aplikácie z pohľadu používateľa a efektívnosti siete.
- `Škálovateľnosť` — určuje schopnosť architektúry podporovať veľké množstvo komponent, alebo interakcie medzi komponentami vrámci aktívnej konfigurácie.
- `Jednoduchosť` — hlavný spôsob ako architektonické štýly prinášajú jednoduchosť je princíp rozdelenia zodpovednosti vrámci komponent, ak sú funkcie rozdelené tak, že komponenty sú menej zložité, potom budú ľahšie pochopiteľné a implementovateľné.
- `Modifikovateľnosť` — spočíva v jednoduchosti s akou môže byť aplikačná architektúra zmenená. Môže byť ďalej rozdelená na vyvíjateľnosť, rozšíriteľnosť, prispôsobivosť, konfigurovateľnosť a opakovateľnosť.
- `Viditeľnosť` — vzťahuje sa na schopnosť komponenty monitorovať alebo sprostredkovať interakciu medzi ďalšími komponentami. Môže umožniť zlepšenie výkonu prostredníctvom zdieľaného ukladania do vyrovnávacej pamäte, škálovateľnosť pomocou vrstvových služieb, spoľahlivosť pomocou reflexného monitorovania a bezpečnosť tým, že dovolí aby interakcie boli kontrolované.

[6]Hypertext Transfer Protocol
[7]Uniform Resource Identifiers
[8]World Wide Web

- `Prenositeľnosť` — schopnosť behu v rôznych prostrediach.
- `Spoľahlivosť` — z pohľadu aplikačnej architektúry sa môže jednať o stupeň, v ktorom je architektúra v prítomnosti, na systémovej úrovni, náchylná k zlyhaniu, čiastkovým poruchám v rámci komponent, konektorov alebo dát.

Existujú dve základné perspektívy pre architektonický dizajn, či už ide o stavanie domu alebo softvéru. Prvý je, že designér začína z ničoho (hovorovo „na zelenej lúke") a začína vytvárať architektúru z už známych komponent, až kým nie sú splnené požiadavky pre daný systém.

Druhý je, že designér začína s požiadavkami systému ako celku, bez obmedzení a postupne identifikuje a pridáva obmedzenia na elementy systému, aby sa rozlíšil dizajnový priestor a umožnil tak ovplyvňovanie systému prirodzene, s harmóniou systému.

Takto vznikol REST, ktorý následne môžeme popísať týmito obmedzeniami:

Klient - server — oddelenie serverovej a klientskej časti. Oddelením užívateľského rozhrania od dátového úložiska, zvýšime prenositeľnosť klientskej časti na viacero platforiem a zlepšíme škálovateľnosť zjednodušením komponent servera. Jednoducho klient sa nestará o uloženie dát do databázy, server sa nestará o užívateľské rozhranie. Server a klient môžu byť vymenené a vyvíjané nezávisle, až kým sa nezmení rozhranie.

Bezstavovosť (ang. *stateless***)** — z celého návzu RESTu vyplýva, že bezstavovosť je jedna z jeho hlavných atribútov. V podstate to znamená, že potrebný stav pre odbavenie požiadavku je obsiahnutý v samotnej požiadavku, či už ako súčasť URI, parametrov query-stringu, tela alebo hlavičiek. URI jednoznačne identifikuje zdroj a telo obsahuje stav (alebo zmeny stavu) tohoto zdroja. Každá požiadavka na server musí obsahovať tieto dáta, aby server žiadosti rozumel. Stav komunikácie je preto iba na klientskej časti.

Potom ako server požiadavku spracuje, príslušný stav, alebo časti stavu, na ktorom záleží, sú klientovi sprostredkované prostredníctvom hlavičiek, návratového kódu a tela odpovede. Týmto spôsobom sa zvyšuje hlavne škálovateľnosť a spoľahlivosť. Škálovateľnosť preto, lebo server si nemusí pamätať interný stav jednotlivých žiadostí a následne tento stav zohľadňovať pri ďalších.

Bezstavovosť však prináša aj svoje nevýhody. Napríklad znižovanie výkonu siete zasielaním opakujúcich sa dát posielaných v správach za sebou, pretože ich nemôžeme uchovať na serveri. Ďalšou nevýhodou je to, že sa musíme spoľahnúť na sémantickú správnosť správ, ktoré prichádzajú na server od klienta a opačne.

ang. Cacheable — toto obmedzenie vyžaduje, že dáta ktoré prišli v odpovedi musia byť implicitne alebo explicitne označené ako (`ang. cacheable`) alebo (`ang. non-cacheable`). Ak je odpoveď označená ako cacheable, klient má právo si ju uložiť „u seba" a použiť túto odpoveď na neskôr. Ak sa bude klient dotazovať rovnako, tak sa požiadavka neposiela na server, ale použije sa odpoveď z klientovej pamäte.

Týmto spôsobom sa dá redukovať počet požiadaviek na server, čím sa môže zvýšiť výkon (rýchlosť) klienta ale aj znížiť spotreba dát. To sa môže hodiť hlavne pre mobilné pripojenie do internetu, kde mobilné dáta sú finančne náročnejšie.

Uniformné rozhranie — hlavným prvkom, ktorý rozlišuje architektonický štýl REST od iných štýlov založených na sieti, je jeho dôraz na jednotné rozhranie medzi komponentami. Zjednodušuje a oddeľuje architektúru, čo umožňuje každej časti sa nezávisle vyvíjať.

REST uplatňuje štyri základné princípy jednotného rozhrania. Prvým je založenie na zdrojoch. Každý zdroj má vlastnú URI, ktorá identifikuje zdroj. Zdroj samotné sú konceptuálne oddelené od reprezentácie, ktorú dostane klient, tzn. že server nevráti klientovi samotnú databázu, ale radšej nejaké štruktúrovanú reprezentáciu ako HTML, XML, JSON.

Druhým je manipulácia so zdrojmi cez ich reprezentáciu. Keď klient obdrží odpoveď zo servera, napríklad v podobe JSON dát, ktoré zahŕňajú aj metadata, je schopný na základe týchto informácií zmeniť, poprípade zmazať dáta na serveri, ak má nato oprávnenia.

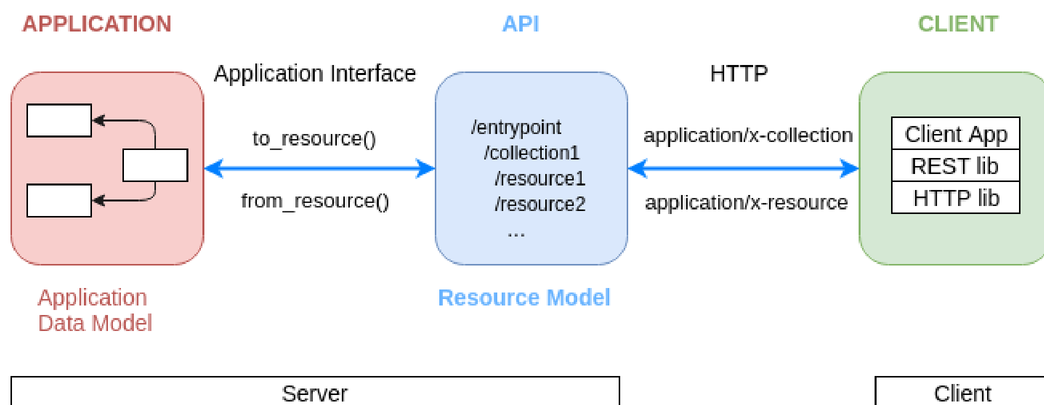
Tretí princíp sú samo-popisovacie správy, tzn. že správy obsahujú informácie o tom, ako sa má správa spracovať. Napríklad, ktorý parser sa má zavolať podľa hlavičky Internet media type[9].

Posledným je tzv. HATEOAS[10]. Znamená to, že môžeme používať hypertextové odkazy v správach. Napríklad, server vracia údaje o mojom bankovom účte, ak je môj stav kladný, vráti mi adresy, na ktorých som schopný vybrať peniaze, vložiť, alebo vykonať transakciu. Na druhej strane, ak je môj účet záporný, vráti mi adresu, na ktorej som schopný peniaze vložiť. Adresy v tomto zmysle, sú hypertextové odkazy.

Vrstvový systém — umožňuje, že architektúra je postavená na hierarchických vrstvách podľa obmedzenia správania sa komponent, za predpokladu, každá vrstva môže „vidieť“ iba vrstvy vo svojej bezprostrednej blízkosti. Ak sa napríklad medzi klienta a server vložia ďalšie servery, klient o tom nebude vedieť. Takéto servery (vrstvy) môžu byť vložené za účelom zvýšenie bezpečnosti, alebo pridanie cache pamätí medzi jednotlivými vrstvami a podobne. Nevýhodou je, že s každou pridanou vrstvou sa zvyšuje doba spracovania požiadavku.

Kód na vyžiadanie — `ang. code on demand`. je posledným voliteľným obmedzením pre REST. Ide o schopnosť servera, kedy vie dočasne rozšíriť alebo upraviť funkcionalitu klienta, prenesením logiky na klientsku stranu. Príkladom môžu byť kompilované komponenty ako sú Java applety alebo klientské scripty ako napríklad JavaScript. Toto obmedzenie je voliteľné z dôvodu znižovania viditeľnosti.

Pri porušení akéhokoľvek iného obmedzenia nemožno považovať takúto architektúru za REST [12].



Obr. 3.2: Princíp ako funguje REST (Zdroj: https://www.hongkiat.com)

[9]http://www.htmlquick.com/reference/mime-types.html
[10]Hypertext As The Engine Of Application State

HTTP metódy

Architektúra REST využíva na komunikácia HTTP protokol. Tento protokol okrem iného definuje aj množinu metód, ktorých cieľom je bližšie špecifikovať akcie nad určitým zdrojom.

Primárne používanými HTTP metódami pre REST sú POST, GET, PUT, PATCH a DELETE. Tie korešpondujú s operáciami vytvorenia, aktualizácie, a zmazania, tiež známe ako `CRUD`[11] operácie. Existujú aj ďalšie metódy, ktoré sa až tak často nepoužívajú. Za zmienku stojí OPTIONS a HEAD.

Ďalej si bližšie popíšeme najčastejšie používané metódy:

POST — najčastejšie sa využíva na vytvorenie nového zdroja, najmä na vytvorenie podzdrojov, tzn. podedených z niektorého rodiča, nadradeného zdroja (napríklad POST na adresu `http://www.example.com/customers`, vytvorí nového zákazníka).

Požiadavky teda smerujú na rodičovský zdroj, ktorý ho vytvorí a asociuje ho so sebou, priradí mu jedinečný identifikátor (URI, na ktorej je novo-vytvorený zdroj dostupný) a podobne. Pri úspešnom vytvorí, vracia HTTP status kód `201`, spolu s odkazom na novovytvorený zdroj v hlavičke `Location`.

Táto metóda nie je bezpečná a ani nepatrí do skupiny tzv. idempotentných metód. Bezpečná nie je z toho dôvodu, pretože modifikuje stav servera. Idempotentná nie je z toho dôvodu, že ak opakovane sa vykoná ta istá požiadavka, nevráti sa ten istý výsledok ako z predchádzajúcej.

GET — slúži na získanie, respektíve čítanie reprezentácie daného zdroja (napríklad GET na adresu `http://www.example.com/customers/12345`, získa dáta o zákaznikovi s identifikátorom 12345). Ak všetko prebehlo na serveri v poriadku, GET vracia v odpovedi dáta v štruktúrovanej podobe ako XML alebo JSON so status kódom 200. V prípade ak daný zdroj neexistuje, alebo server požiadavku nerozumie vracia návratový kód `404 - Not Found`, prípadne `400 - Bad Request`.

Podľa HTTP špecifikácie, je metóda GET (spolu s HEAD), určená iba na čítanie dát, nie na zmenu. Ak sú takto používané, považujú sa za bezpečné. Znamená to, že môžu byť volané bez rizika zmien dát alebo inej kolízie – rovnaký efekt je pri žiadnom alebo viacnásobnom volaní. Z toho vyplýva že táto metóda patrí medzi idempotentné, pretože viacero rovnakých požiadaviek vracia rovnaký výsledok ako jeden.

GET by nikdy nemal meniť zdroje na serveri.

PUT — slúži na úpravu už existujúcich zdrojov. Pomocou tejto metódy je však možné aj zdroj vytvoriť za predpokladu, že výsledná URI na ktorej je vytvorený zdroj dosiahnuteľný je v réžii klienta (napríklad PUT na adresu `http://www.example.com/customers/12345`, vytvorí nového zákazníka s identifikátorom 12345). V prípade vytvorenia nového zdroja, server vracia status kód `201` avšak v hlavičke sa už URI nenachádza, kvôli tomu že adresa je klientovi známa.

Tak isto ako POST nepatrí do skupiny bezpečných metód, kvôli tomu, že modifikuje dáta na serveri. Patrí však do skupiny idempotentných, pretože ak vytvárame alebo upravujeme zdroje pomocou PUT-u a následne zavoláme PUT znova, zdroj je stále na serveri a stále sa nachádza v rovnakom stave ako pri prvom volaní.

Mnohým príde metóda PUT zbytočne mätúca. V dôsledku toho by sa mala používať striedmo, ak vôbec.

[11] Create, read, update and delete

PATCH — slúži na úpravu zdrojov. Požiadavky, ktoré sú zaslané touto metódou potrebujú iba zmeny, ktoré nastali v danom zdroj, nie zdroj celý (napríklad PATCH na adresu `http://www.example.com/customers/12345`, upraví dáta zákazníka s identifikátorom 12345).

Podobá sa na PUT, ale narozdiel od neho telo obsahuje súbor pokynov, ktoré popisujú, ako by mal zdroj byť zdroj upravený, aby vytvoril novú verziu. Metóda nie je bezpečna a ani idempotentná.

Aby sa predišlo kolíziám v zmysle, že viacero používateľov upravuje naraz ten istý zdroj, posiela sa v požiadavku v hlavičke `If-Match` tzv. `ETag`. Ten hovorí, akú verziu zdroja chce klient na serveri upravovať, tzn. že ETag na serveri a u klienta musia byť rovnaké. Ak sa ETag-y líšia, server vracia odpoveď so status kódom `412 - Precondition Failed`.

DELETE — tak ako názov napovedá, slúži táto metóda na mazanie zdrojov (napríklad DELETE na adresu `http://www.example.com/customers/12345`, zmaže zákazníka s identifikátorom 12345). Pri úspešnom mazaní, server vracia správu so status kódom `200`, v tele správy môže poprípade vrátiť aj reprezentáciu zmazaného zdroja.

Metóda nepatrí do skupiny bezpečných metód, ale patrí do skupiny idempotentných. Pokus o vymazanie už vymazaného zdroja vráti chybu zo strany servera s príslušným návratovým kódom.

3.4.2 REST serveru Livy

Server Livy poskytuje webové rozhranie pre vzdialenú prácu so Sparkom, ako bolo spomenuté v časti 3.3. V tejto časti sú popísané zdroje, ktoré budú použité a čo konkrétne znamenajú.

POST - /sessions — vytvorí na serveri novú reláciu, v ktorej bude očakávať statements, posielané do tejto relácie. Pred vytvorením sa špecifikuje o aký typ relácie pôjde, to znamená v ktorom jazyku bude jednotlivé statements akceptovať. Do tela správy je možné vložiť ďalšie atribúty, ako napríklad cestu k .jar/.py súborov, ktoré sa majú v relácii použiť, koľko exekútorov sa má v danej relácií spustiť, koľko pamäte si majú alokovať atď.. Pre naše účely bude zatiaľ postačujúci atribút `kind`. Jeho hodnota je textový reťazec, ktorý predstavuje názov programovacieho jazyka, ktorý daná relácia akceptuje. Hodnoty môžu byť:

- `spark` — akceptuje kód v jazyku Scala.
- `pyspark` — akceptuje kód v jazyku Python (<3).
- `pyspark3` — akceptuje kód v jazyku Python 3.
- `sparkr` — akceptuje kód v Spark R.
- `sql` — akceptuje SQL.

GET - /sessions — získa všetky aktívne interaktívne relácie. V parametroch požiadavku je index, ktorý udáva začiatok načítania relácií a množstvo relácií, ktoré chcem získať.

V odpovedi sa nachádzajú zoznam objektov všetkých načítaných relácií, počet koľko ich bolo načítaných a index, od ktorého sa načítavalo.

GET - /sessions/{sessionId} — vráti informácie o danej relácií. V podstate vracia objekt Session popísaný **??**

GET - /sessions/{sessionId}/state — vráti, v akom stave sa nachádza relácia. Stavy v ktorých sa môže relácia nachádzať sú následovné:

- `not_started` — relácia sa nezačala.
- `starting` — relácia práve štartuje.
- `idle` — relácia čaká na vstup (statement alebo batch).
- `busy` — relácia práve vykonáva úlohu.
- `shutting_down` — relácia sa vypína.
- `error` — relácia skončila s chybou.
- `dead` — relácia ukončená.
- `success` — relácia sa úspešne zastavila.

DELETE - /sessions/{sessionId} — zastaví danú reláciu.

POST - /sessions/{sessionId}/statements — spustí statement v danej relácií. V tele požiadavku, v atribúte `code`, je kód, ktorý sa má v danej relácií vykonať.

POST - /sessions/{sessionId}/statements/{statementId}/cancel — zrušenie tohoto statementu.

GET - /sessions/{sessionId}/statements — vráti všetky statementy v danej relácií.

GET - /sessions/{sessionId}/statements/{statementId} — vráti informácie o danom statemente. V odpovedi sú dáta ako kód, ktorý daný statement vykonáva, v akom je stave, prípadne výstup. Statement sa môže nachádzať v týchto stavoch. /sessions/sessionId/statements/

- `waiting` — čaká vo fronte, vykonávanie zatiaľ nezačalo.
- `running` — práve sa vykonáva.
- `available` — vykonávanie sa skončilo, sú dostupné dáta.
- `cancelling` — statement sa práve ruší.
- `cancelled` — statement zrušený.

Livy poskytuje aj ďalšie URI, na ktoré sa dá dotazovať [12]. Pre účely tejto aplikácie si vystačíme z vyššie uvedenými.

[12]http://livy.incubator.apache.org/docs/latest/rest-api.html

Kapitola 4

Súčasné riešenia

V predchádzajúcich kapitolách boli predstavené technológie, ktoré sa v súčasnosti používajú na analýzu *Big data*. Či už sa jedná o samotné spracovanie dát Sparkom, alebo uľahčenie práce s týmto prostredím pomocou serveru Livy. Ďalej budú predstavené riešenia, ktoré tieto dva technológie spájajú a vytvárajú tak nástroje pre efektívnejšiu prácu s dátami. Bude sa jednať o tzv. „notebooky“, ktoré predstavujú jednoduché vývojové prostredia pre prácu s rôznymi technológiami na jednom mieste.

4.1 Projekt Apache Zeppelin

Apache Zeppelin [15] je open-source projekt. Jedná sa o webový notebook, ktorý umožňuje interaktívnu analýzu dát. Interaktívnym spôsobom poskytuje dátovým inžinierom možnosť byť produktívnejší vo vývoji, organizovaní, vykonávaní a zdieľaní kódu. Umožňuje tiež vizualizáciu výsledkov bez potreby ďalších nástrojov.

Notebooky neslúžia vývojárom len na spúšťanie kódu ale aj na prácu s celým workflow počas analýzy. Zeppelin poskytuje niekoľko notebookov, ktoré sú schopné pracovať so Sparkom.

Hlavnú úlohu v tomto projekte preberajú interpreti. Tých ponúka Zeppelin niekoľko. Ide o programovacie jazyku, ktoré môžu byť pripojené do Zeppelinu ako pluginy a tie potom môže vývojár využívať na ďalšiu prácu s výsledkami. Najnovšia verzia 0.7.3 ponúka dvadsaťtri interpretov. Patrí medzi ne napríklad Spark, JDBC, Python, Livy, HDFS atď. . V jednom notebooku je možné používať niekoľko interpretov naraz. Je už na programátorovi akú kombináciu si vyberie.

Programátor si pri vytvorení nového notebooku vyberie prednastavený interpret, ktorý potom pri písaní kódu špecifikuje na začiatku ako %<názov-interpretu>. Po vytvorí je možné ďalší interpret pridať a ďalej v notebooku používať. Ukážka kódu, ktorý je napísaný SQL interpretom, vráti z tabuľky bank, záznamy age, ktoré majú hodnotu menšiu ako 30, zoradené vzostupne 4.1.

```
%sql
select age, count(1) value
from bank
where age < 30
group by age
order by age
```

Výpis 4.1: Ukážka kódu z Zeppelin notebooku

Do jedného notebooku je možné pripojiť viac interpretov. Jednotlivé interpreti majú vlastné nastavenia, ktoré sa exportujú ako premenné prostredia. V Zeppeline sú interpreti zlučované do skupín. Jednú skupinu tak tvorí napríklad Spark, PySpark, Spark SQL a dependency loader. Technicky, Zeppelin interpreti z jednej skupiny bežia v rovnakom JVM (*Java Virtual Machine*).

Ďalej je možné si vytvoriť vlastný interpret, ktorý je potom možné používať v Zeppeline, alebo pripojiť sa na vzdialený server, na ktorom beží nejaký interpret.

Vstupy a výstupy

Ako bolo opísané vyššie, Zeppelin ponúka niekoľko interpretov [5]. S tým súvisí aj vstupný program. Programátor môže vytvoriť SQL tabuľku v Scale s dátami, ktoré načítal z Amazon S3. Tabuľku ďalej spracuje pomocou Spark SQL. Všetko závisí od problému, ktorý programátor rieši a ktorý z interpretov sa mu na vyriešenie úlohy viac hodí.

Vstupné a výstupné dáta sú v notebookoch zobrazené v samostatných paragrafoch. Zeppelin umožňuje v jednom riadku zobrazovať viac paragrafov, narozdiel od Jupytera 4.2.

Zaujímavo sú v projekte Zeppelin vyriešené dynamické vstupy. Jedná sa napríklad o dotazy do databázy, ktoré sa menia. Napríklad v tabuľke Banka by chcel programátor často meniť parameter meno a priezvisko užívateľa, ktorý uskutočnil transakciu. Namiesto toho aby pri každom dotaze musel ručne vpisovať do kódu nové meno a priezvisko, stačí mu vytvoriť formulár s dvoma poliami, kde by hľadané dáta vpisoval a z kódu by sa odkazoval na hodnotu vo formulári. Toto chovanie umožňuje integrácia AngularJS do notebookov [1].

Notebooky, ktoré si programátor vytvoril, môžu byť potom uložené no Git-e, Amazon S3, Microsoft Azure alebo ZeppelinHub.

Ďalšie rozšírenia pre notebooky, čo sa týka grafického editoru zatiaľ nie sú známe.

Najzaujímavejšia na projekte Zeppelin je práca s výstupnými dátami. Ako bolo spomenuté v časti 4.1, Zeppelin umožňuje interaktívnu analýzu dát. Jedná sa o zobrazovanie výstupných dát pomocou rôznych grafov, tabuliek. Na obrázku 4.1 vidieť interpretáciu SQL dotazu na záznamy ľudí z banky, ktorých vek je nižší ako tridsať rokov.

Zhrnutie

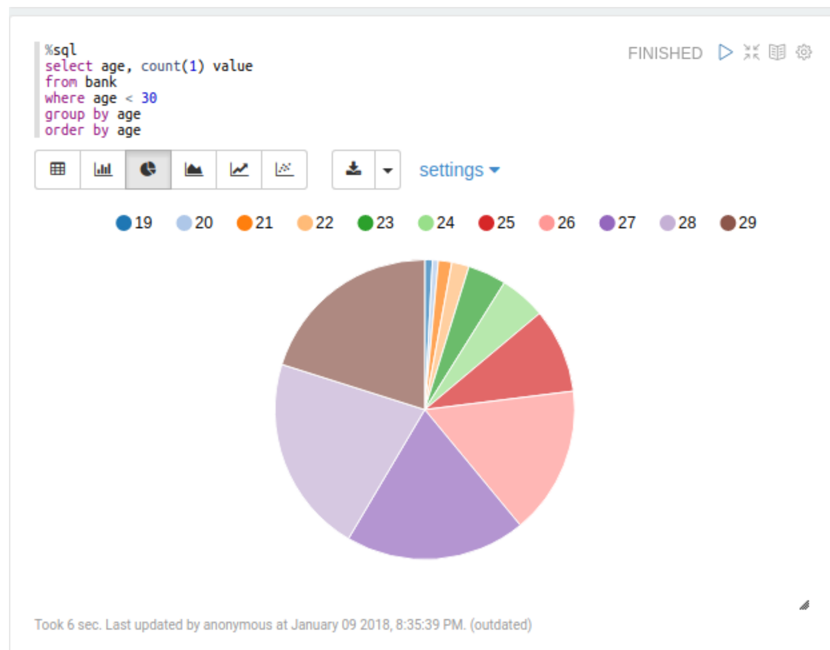
Apache Zeppelin je šikovný webový nástroj, ktorý pomôže pri vizualizácií a objavovaní veľkých datasetov. Jedná sa o riešenie, ktoré môže značne urýchliť prácu pri spracovaní Big data, za zmienku určite stojí aj zdieľanie notebookov, čo urýchli prácu medzi vývojármi [8].

4.2 Projekt Jupyter

Projekt Jupyter [3] je podobne ako projekt Zeppelin, vytvorený ako webové rozhranie, cez ktoré je možné vytvárať a spúšťať rôzne úlohy. Jedná sa o známy notebook, ktorý využívali firmy ako Google alebo NASA. Je to open-source projekt, vytvorený z projektu IPython, v roku 2012, na podporu interaktívneho dátového inžinierstva a vedeckých výpočtov cez všetky programovacie jazyky. Oproti Zeppelinu má asi 10x väčšiu komunitu (podľa počtu prispievateľov na GitHube).

Jupyter poskytuje vyše osemdesiat kernelov (jedná sa v podstate o interpret v Zeppeline). Ak potrebuje programátor v notebooku využívať napríklad PySpark, potrebuje nastaviť premenné prostredia, aby Jupyter vedel o Sparku a Pythone.

[1] https://zeppelin.apache.org/docs/0.7.3/manual/dynamicform.html



Obr. 4.1: Intepretácia dát pomocou grafu (Zdroj: https://zeppelin.apache.org)

Programátor môže písať a spúšťať svoj kód v nezávislých častiach, ktoré môžu zdieľať rovnaký namespace. Tento spôsob výrazne urýchľuje testovanie a tvári sa viac modulárne.

Vstupy a výstupy

Jupyter ponúka pre programátora vstupný notebook, kde je možné písať kód, spúšťať a získavať výsledky. Notebook je rozdelený do paragrafov, ktoré obsahujú kód a výstup. Svojím spôsobom to funguje podobne ako v Zeppeline s tým rozdielom, že programátor si musí zobrazovanie výstupov definovať sám. Zeppelin ponúkal vstavané nástroje, napríklad pre vykresľovanie grafov. Na obrázku 4.2 je ukážka kódu na spočítanie slov v súbore pomocou Sparku.

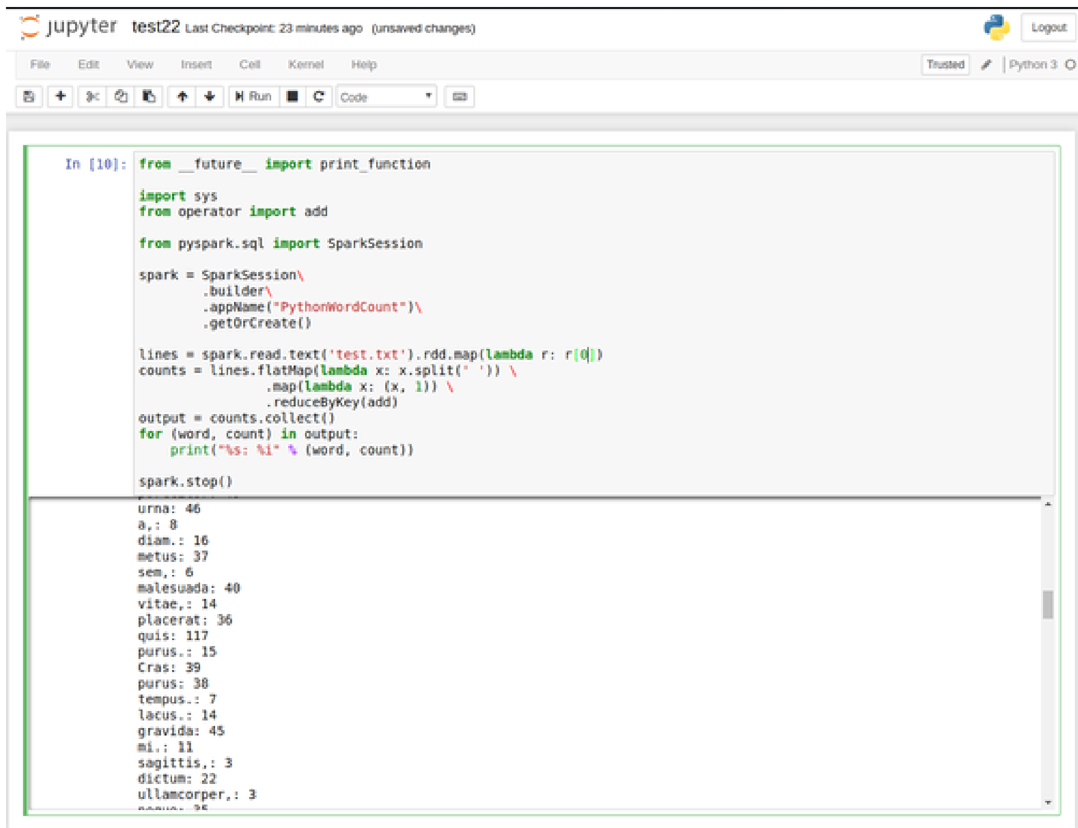
Editor kódu v notekooch Jupytera poskytuje automatické dopĺňanie kľučových slov pre Python, SQL a podobne. Je možné doplniť ďalšie rozšírenia, ktoré rozšíria funkcionalitu notebookov a uľahčia programátorovi prácu [2].

Zhrnutie

Na prvý pohľad sa môže zdať, že sa Jupyter od Zeppelinu vôbec nelíši. Hlavný rozdiel je v podpore kernelov (interpretov), kde to Jupyter ponúka omnoho viac ako Zeppelin. Ďalšou nevýhodou pre Jupyter môže byť chýbajúca kombinácia viacerých interpretov v jednom notebooku.

V základnom nastavení Jupyter nepodporuje žiadnu autentizáciu. Ak je potrebné je možné nainštalovať Jupyterhub, ktorý sa o ňu bude starať. Problém je však v náročnosti, pretože Jupyterhub spustí každému autentizovanému používateľovi vlastný Jupyter server, čo môže pri veľkom počte používateľov spôsobiť priveľkú záťaž a tá sa môže odraziť na vý-

[2] https://github.com/ipython-contrib/jupyter_contrib_nbextensions



Obr. 4.2: Ukážka vstupu a výstupu v rozhraní Jupyter (Zdroj: http://jupyter.org/)

kone. Narozdiel od Zeppelinu, ktorý si autentizáciu rieši sám a nepotrebuje k tomu ďalšiu službu.

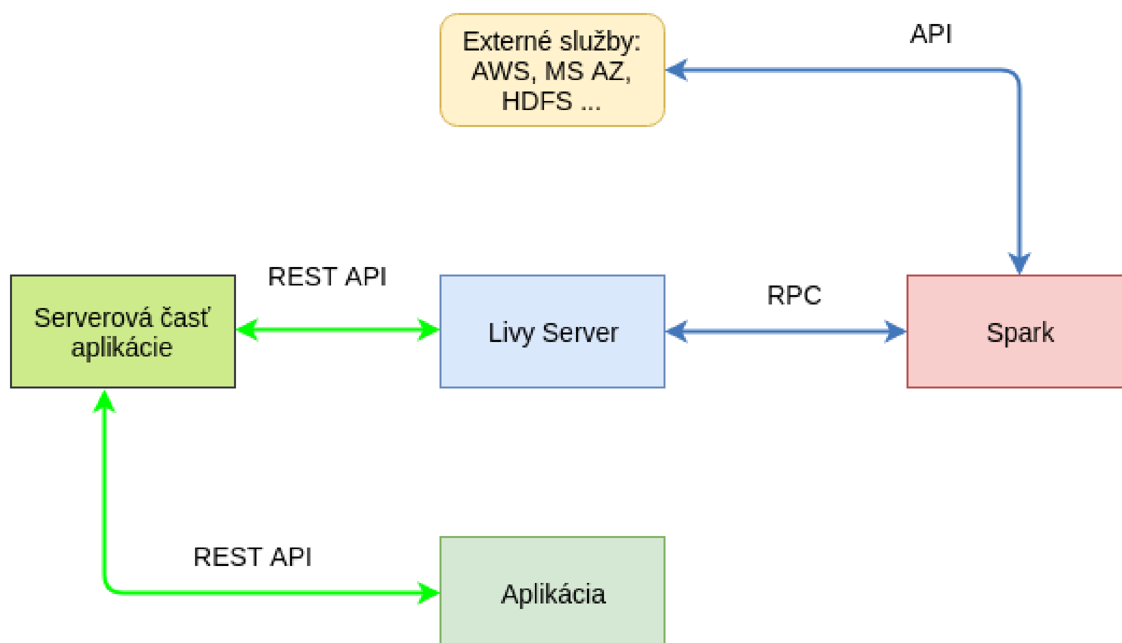
Kapitola 5

Návrh aplikácie

Cieľom tejto práce je navrhnúť webovú aplikáciu pre interaktívne zadávanie a spúšťanie Spark úloh. V predchádzajúcej časti boli predstavené riešenia 4.2, 4.1, ktoré sa v súčasnosti používajú. Jedná sa však o komplexné aplikácie, ktoré ponúkajú nielen spúšťanie Spark úloh, ale aj ďalšie programovacie techniky nielen na prácu s Big data. V tejto kapitole bude cieľom predstaviť aplikáciu, ktorá poskytne používateľovi interaktívne vytvárať Spark aplikácie, ktoré bude možné spúšťať, sledovať ich priebeh, získavať a ukladať výsledky.

Dôraz sa kladie na grafické vytváranie Spark úlohy. Postupnosť volania Spark funkcií by mala byť reprezentovaná graficky, formou orientovaného grafu. Užívateľ si potom z preddefinovaných grafových komponent, ktoré reprezentujú Spark funkcie, bude môcť poskladať úlohu, nastaviť parametre vybraným funkciám a následne úlohu spustiť.

5.1 Návrh architektúry



Obr. 5.1: Architektúra aplikácie

5.2 Externé služby

Vstupom sú pre Spark úlohy vo väčšine prípadov dáta, ktoré sú uložené v súboroch na disku. Pre potreby uloženia takýchto dát budeme v tejto práci využívať systém Hadoop, konkrétne distribuovaný súborový systém HDFS (Hadoop Distributed File System) [21]. Jedná sa o samo-zotavovací distribuovaný súborový systém, ktorý je spoľahlivý, škálovateľný, odolný voči chybám. Dokáže uložiť dáta v rôznych formátoch (text, obrázky, videá ...) bez ohľadu na architektúru.

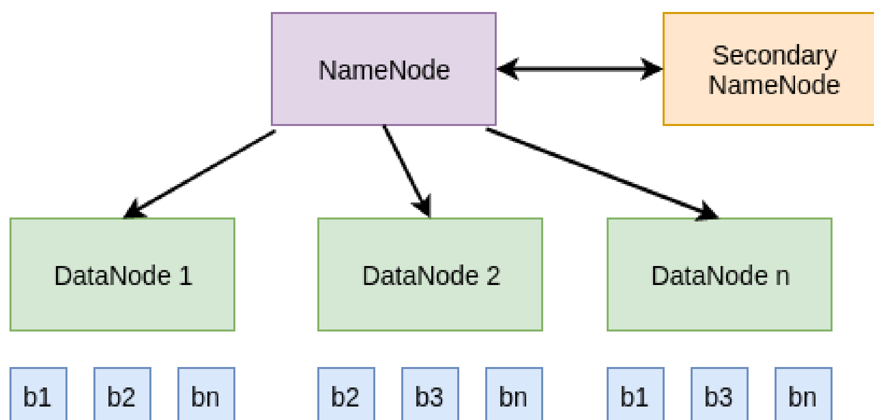
Hlavnou výhodou HDFS je tolerancia voči chybám. Poskytovaním rýchleho prenosu dát medzi uzlami a umožnením Hadoopu naďalej poskytovať služby aj v prípade zlyhania uzlov minimalizuje rizika zlyhania výpočtu.

Využíva architektúru master/slaves 5.2. Na master uzlu je spustený NameNode démon a sekundárny NameNode. Na ostatných (slave) uzloch bežia DataNode démony.

NameNode ukladá a manažuje metadata o súborovom systéme v súbore fsimage. Metadata sa cachujú do hlavnej pamäte, pre rýchlejší prístup. Kontroluje DataNode démony pri vykonávaní I/O. Ďalej kontroluje a riadi ako sú súbory rozdelené do blokov, identifikuje ktorý zo slave uzlov môže mať daný blok dát uložený spolu s celkovým stavom a životnosťou súborového systému.

DataNode sú primárnym úložným miestom pre bloky dát a sprostredkúvajú služby pre zápis a čítanie dát (kontrolované od NameNode). Bloky dát uložené na DataNode-och sú replikované podľa konfigurácie, čo zaručuje spoľahlivosť a vysokú dostupnosť.

Sekundárny NameNode neslúži ako záloha primárneho NameNode. Jeho úlohou je periodické sledovanie súborového systému, sledovanie zmien a ich aplikovanie do súboru fsimage, čo pomáha k updatovaniu a rýchlejšiemu štartu primárnemu NameNodu pri ďalšom spustení.



Obr. 5.2: Architektúra HDFS (Zdroj: [21])

Výhodou používania HDFS v Hadoop systéme je aj vlastné používateľské rozhranie, ktoré Hadoop poskytuje po nainštalovaní na lokálny počítač cez webové rozhranie. To budeme používať pre správu súborov pre naše Spark aplikácie. Na lokálnom počítači je potom cesta k súboru na adrese „hdfs://localhost:54310/“. (54310 je prednastavený port pre HDFS).

Kapitola 6

Návrh webovej aplikácie

V tejto časti bude predstavený návrh pre webovú časť tejto práce. Cieľom je vytvoriť aplikáciu spustiteľnú v prehliadači, ktorá bude schopná vytvárať a spúšťať Spark úlohy, informovať užívateľa o behu danej úlohy, o ukončení, o chybách a podobne.

Umožní mu vytvárať vlastné komponenty z už pred definovaných Spark komponent, ukladanie a mazanie takýchto komponent. Užívateľ by mal mať možnosť svoje vytvorené komponenty zdieľať ostatným používateľom, aby aj ľudia bez znalosti programovania so Sparkom, vedeli tento nástroj použiť.

Užívateľ by mal byť schopný si svoju aktuálnu prácu uložiť, prípadne neskôr sa k nej vrátiť a ďalej pracovať. Perzistenciu dát bude mať na starosti serverová časť aplikácie.

Aplikácia je teda určená pre ľudí, ktorí vedia vytvárať aplikácie v prostredí Apache Spark, ale tým, že umožní vytvárať funkčné celky (komponenty) a zdieľať ich medzi ostatnými používateľmi, sa zväčšuje základňa potenciálnych používateľov tejto aplikácie.

Jednotlivé požiadavky sú zhrnuté v diagramu prípadov použitia 6.1, ktorý bol vytvorený v jazyku UML[1] pre grafické znázornenie požiadaviek.

6.1 Diagram prípadov použití

Diagram prípadov použitia na obrázku 6.1 znázorňuje možnosti interakcie používateľa s webovou aplikáciou. Pri návrhu sa počíta s tým, že užívateľ pri práci s aplikáciou je po celú dobu prihlásený. O autentifikáciu a autorizáciu používateľa sa bude starať serverová časť aplikácie.

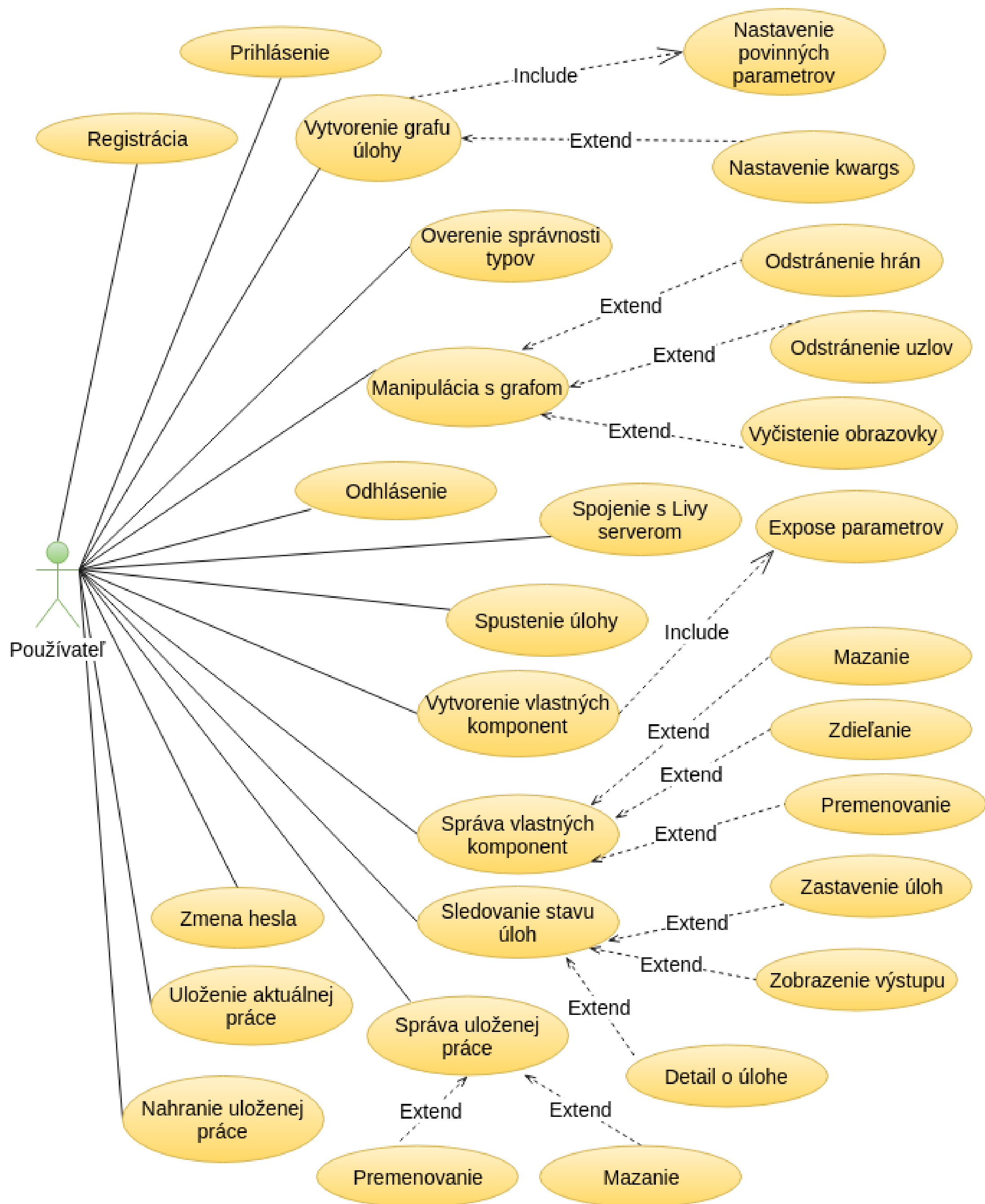
6.2 Návrh užívateľského rozhrania

Užívateľské rozhranie by malo pokrývať, respektíve malo by umožňovať používateľovi vykonávať funkcie, ktoré sú popísané v diagramu prípadu použití 6.1. Jednotlivé časti užívateľského rozhrania popisujú obrázky, ktoré bližšie opisuje sprievodný text.

6.2.1 Prihlasovacia obrazovka

Pri prvom spustení aplikácie sa vyžaduje od používateľa prihlásenie. Ak sa užívateľ niekedy pred tým neprihlásil, bude presmerovaný na stránku `/login`, kde sa mu zobrazí prihlasovacie okno 6.2, kde sa od neho vyžadujú prihlasovacie údaje, email a platné heslo. Email musí

[1] http://www.uml.org - Unified Modeling Language

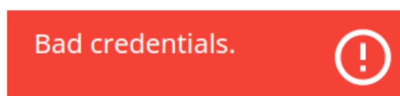


Obr. 6.1: Diagram prípadov použitia

byť v správnom formáte a obe polia musia byť vyplnené. Ak užívateľ účet vytvorený nemá, má možnosť sa zaregistrovať sa, čím sa mu vytvorí účet nový, po kliknutí na odkaz `Create an account`, čím bude presmerovaný na stránku `/register` 6.2.2.

Ak sa užívateľ pokúsi prihlásiť s nesprávnymi prihlasovacími údajmi, bude o tom informovaný v tzv. `popup` okne s chybovým hlásením.

(a) Prihlasovacie okno.



(b) Chybová hláška v popup okne pri nesprávnom emaily alebo heslu.

Obr. 6.2: Pokus o prihlásenie.

6.2.2 Registrácia

Táto stránka sa užívateľovi zobrazí pod adresou `/register`. Dostať sa na ňu môže zo stránky `/login`, tak ako to bolo opísané v časti 6.2.

Vo formulári sa vyžaduje od užívateľa zadanie emailovej adresy v správnom formáte. V ďalších dvoch poliach sa vyžaduje heslo, ktoré musí obsahovať najmenej 6 znakov. Ak sa používateľ bude snažiť zaregistrovať už pod existujúcou emailovou adresou, alebo nezadané zhodné heslá v oboch poliach bude o tom informovaný chybovým hlásením.

Po úspešnom vytvorení profilu bude informovaný hlásením o úspešnosti operácie a následne presmerovaný na stránku prihlásenia.

Obr. 6.3: Formulár pre registráciu používateľa.

6.2.3 Hlavná obrazovka

Po úspešnom prihlásení je užívateľ presmerovaný na stránku `/home`. Ak bol predtým autorizovaný, je presmerovaný rovno na túto stránku. Pre lepšiu prehľadnosť budú jednotlivé časti opísané po častiach, aby boli jasne viditeľné, doplnené o sprievodný text. Kompletný návrh obrazovky je súčasťou prílohy A.

Menu

V hornej časti obrazovky sa nachádza navigácia medzi jednotlivými stránkami, obrázok 6.4. Odkaz na `Worksheet`, smeruje na obrazovku pod adresou `/home`, čiže na tú na ktorej sa aktuálne nachádzame. Odkaz na `Status` presmeruje používateľa na obrazovku 6.2.4.

Odkaz v pravej časti menu, zobrazuje email práve prihláseného používateľa. Po kliknutí na tento odkaz bude užívateľ presmerovaný na stránku `/profile`, kde si môže editovať svoj profil 6.2.5.

Obr. 6.4: Navigačné menu.

Užívateľské menu

Hneď pod hlavným menu je umiestnené užívateľské menu, obrázok 6.5. To slúži pre nahrávanie alebo ukladanie aktuálnej práce, pridávanie ďalších spojení s Livy serverom alebo zmenu aktuálnej relácie, do ktorej chceme posielať náš kód. Tlačidlo `Save as` ponúka uloženie aktuálnej práce do nového súboru. Po kliknutí na tlačidlo sa zobrazí modálne okno, kde sa žiada od používateľa aby zadal názov súboru.

Ďalšie tlačidlo `Save` uloží aktuálnu prácu do súboru, ktorý je nastavený v pole `File name`. Tlačidlo `Load` nahraje obsah súboru nastavený v poli File name do pracovnej plochy. Operácie ako nahranie alebo uloženie si vyžadujú užívateľovu pozornosť, preto každé vyvolanie bude doprevádzané modálnym oknom pre potvrdenie operácie.

Tlačidlo `Connect to Livy` spôsobí, že sa vytvorí spojenie s Livy serverom. Vytvorené spojenie (relácia) sa nastaví ako aktuálna. Všetky dostupné relácie sú dostupné v poli `Current session`.

Obr. 6.5: Používateľské menu.

Výber komponent

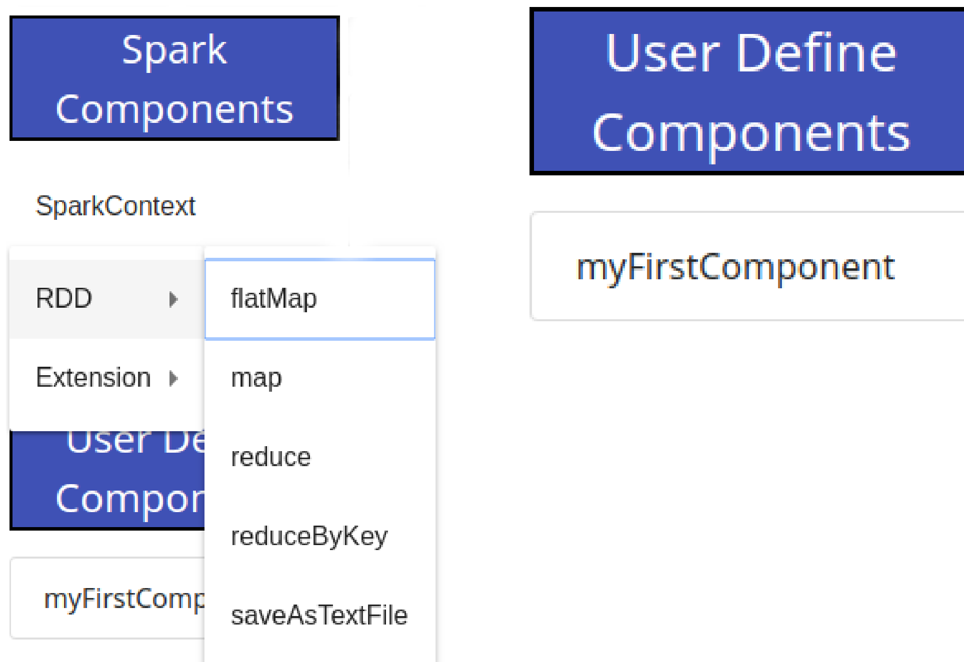
V pravej časti obrazovky sa nachádza zoznam preddefinovaných Spark komponent, obrázok 6.6. Spark komponenty sú vlastne funkcie, metódy, ktoré tvoria rozhranie Sparku. Tie patria do rôznych modulových tried a na základe toho, do akej triedy metóda (komponenta) patrí, tak potom sa aj pod danou triedou v zozname, ktorá sa zobrazí po kliknutí, zobrazuje.

Napríklad, trieda `RDD` definuje metódy `flatMap(), map()` atď., tie sa teda budú zobrazovať pod položkou RDD, viď 6.6. Ak by existovala trieda, ktorá dedí, napríklad z RDD,

tak po kliknutí na RDD by sa táto trieda zobrazila ako ďalší rozbaľovací zoznam. Takéto vnáranie je možné do dvoch úrovní.

Pod Spark komponentami sa nachádza zoznam užívateľských komponent, tj. komponent, ktoré si sám vytvoril. Táto časť sa zobrazuje iba prihlásenému používateľovi.

Medzi nezaradenú komponentu patrí komponenta typu `own_component`, ktorá umožňuje používateľovi pridať vlastný kód. Tlačidlo pre jej pridanie sa nachádza nad zoznamom Spark komponent.



(a) Spark komponenty. (b) Užívateľom definované komponenty.

Obr. 6.6: Výber komponent.

Pracovná plocha

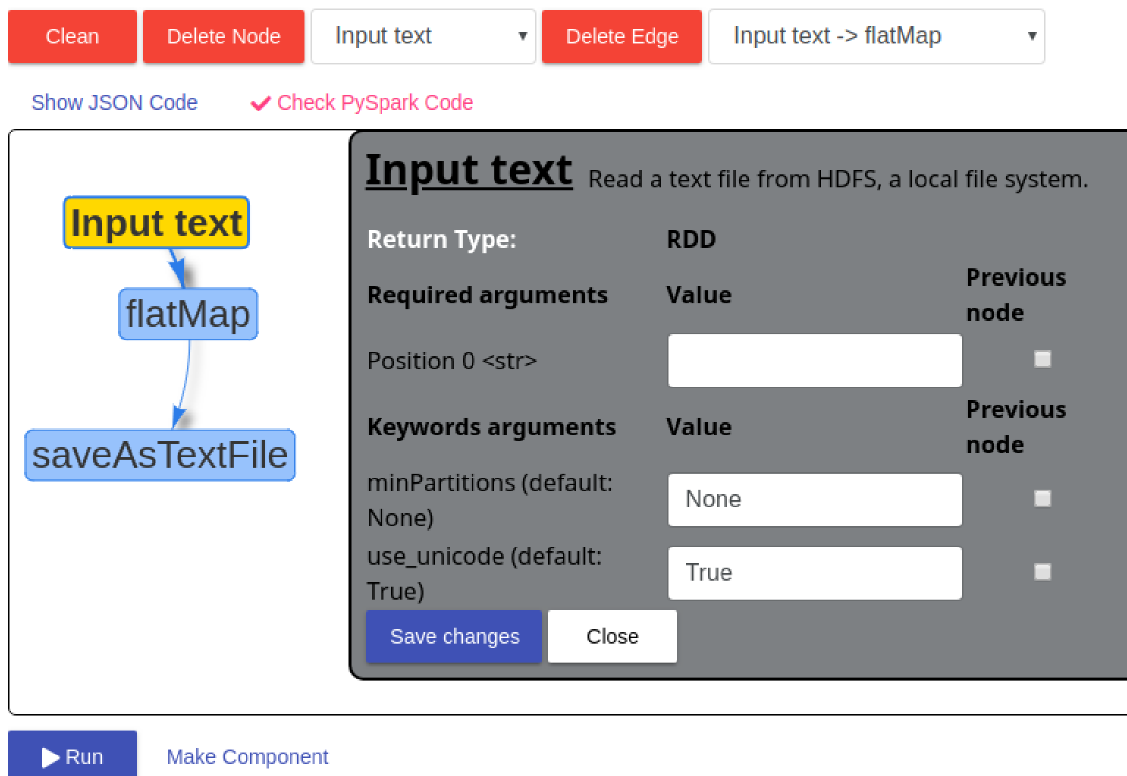
Zbytok stránky tvorí plocha, obrázok 6.7, na ktorú sa pridávajú komponenty a kde sa vytvára Spark aplikácia. Komponenta sa do grafu pridá kliknutím na niektorú z komponent zo zoznamu komponent, obrázok 6.6. Po pridaní na plochu je možné komponentu (uzol) voľne presúvať po ploche.

Vytvorenie hrany medzi jednotlivými uzlami sa prevedie označením uzlu (zvýraznenie žltou farbou) a následným kliknutím na ďalší uzol, tým sa vytvorí hrana. Takýmto spôsobom sa vytvorí Spark úloha reprezentovaná grafom.

Tlačidlo `Clean` slúži pre kompletné vyčistenie pracovnej plochy. Odstránia sa všetky uzly a hrany. Pred mazaním bude užívateľ upozornený v modálnom okne, že či operáciu skutočne praje uskutočniť.

Pre odstránenie uzlov alebo komponent, slúžia tlačidla `Delete Node` a `Delete Edge`. Uzol alebo hranu, ktorú chceme odstrániť je možné vybrať zo zoznamu všetkých uzlov/hrán, poprípade kliknutím na uzol alebo hranu, ktorú chcem odstrániť.

Tlačidlá `Show JSON Code` a `Check PySpark Code` slúžia na debugovacie účely. Prvý menovaný zobrazuje dátovú štruktúru jednotlivých komponent, ktoré sú v grafe, druhý



Obr. 6.7: Plocha pre vytvorenie Spark úlohy s nastavením Spark komponenty.

slúži na overenie správnosti prepojenia jednotlivých uzlov, kontroluje či sú vyplnené povinné argumenty komponent a podobne. Ak všetko prebehlo bez chýb, zobrazí sa vygenerovaný kód 6.8.

Show JSON Code   ✔ Hide PySpark Code

```
1 out_node_1 = sc.textFile("hdfs://localhost:54310/test.txt")
2 out_node_2 = out_node_1.map(lambda x: len(x))
3 out_node_3 = out_node_2.reduce(lambda a,b: a+b)
4 out_node_4 = print(out_node_3)
5
```

Obr. 6.8: Skontrolovaný a vygenerovaný python kód.

Po dvojkliku na uzol grafu (komponentu) sa zobrazí okno, kde sa zobrazí detail vybranej komponenty. Vedľa názvu sa nachádza stručný popis. Pod ním je hodnota `Return Type` ako z názvu vyplýva ide o typ, ktorý vracia daná funkcia.

Ďalej sa nachádzajú polia pre zadanie argumentov funkcie. Ako prvé sa vypĺňajú povinné argumenty od pozície nula až po N-1 (0. pozícia = najviac vľavo, N-1. pozícia = najviac vpravo). Za pozíciou je uvedený typ, aký funkcia v tomto argumente očakáva.

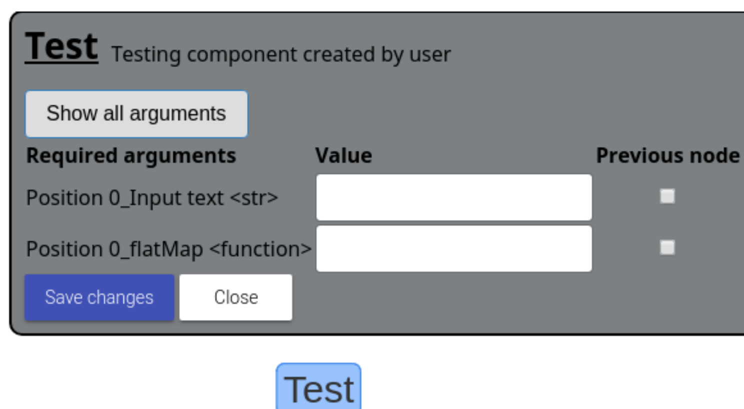
Pod povinnými argumentami sa ďalej nastavujú tzv. „keywords argumenty“, ktorých defaultné hodnoty sú uvedené v zátvorkách (pri zmene, aby bolo jasné aké boli) a v input boxe je nastavená hodnota.

Niektoré funkcie môžu potrebovať ako hodnotu argumentu výstup z prechádzajúcej operácie. Takýto vstup zaistíme zaškrtnutím checkboxu „Previous node“ pri danom argumente.

Nastavenie parametrov Spark komponenty a užívateľskej komponenty sa mierne líšia. Užívateľská komponenta je tvorená z viacerých Spark komponent. Ak by boli viditeľné všetky parametre, tak by bolo nastavenie pri väčšom množstve Spark komponent neprehľadné. Preto sa pri vytvorení vlastnej komponenty špecifikuje, ktoré argumenty sa budú modifikovať, respektíve budú vystavené (expose). Tie sa potom primárne zobrazia pri dvoj kliku na komponentu, obrázok 6.9.

Ak by náhodou užívateľ potreboval zmeniť parameter, ktorý nie je vystavený, použije tlačidlo „Show all arguments“, kde sa mu zobrazia všetky parametre danej komponenty.

Textový popis pri jednotlivých argumentoch hovorí, ktorý parameter danej komponente sa nastavuje, napríklad `Position_0Input text` na obrázku 6.9, nastavuje prvý argument funkcii `Input text`. Argumenty sú zobrazené v poradí, v akom sú pospájané jednotlivé Spark komponenty vnútri užívateľskej komponenty.



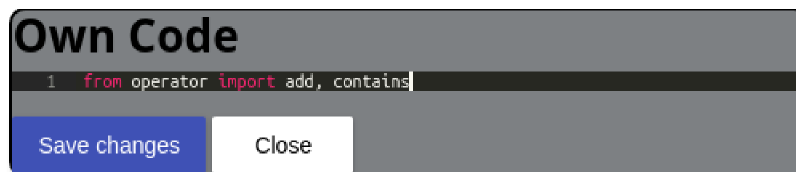
Obr. 6.9: Nastavenie užívateľskej komponenty.

Ako bolo spomenuté v časti 6.2.3, okrem Spark a užívateľských komponent má užívateľ možnosť pridať aj komponentu s vlastným kódom, obrázok 6.10. Ta nemá špeciálne pomenovanie, pre jednoduchosť sa na ploche zobrazuje ako `Own Code`, poprípade môže končiť s číslom, ak by sa ich na ploche nachádzalo viac. Môže sa hodiť na importovanie iných modulov, pridávanie vlastných funkcií, úprava premenných a podobne. Pri nastavovaní komponenty si užívateľ do textového poľa pridá svoj vlastný kód, ktorý sa potom vygeneruje do výsledného kódu. V grafe sa s týmto uzlom pracuje ako s ostatnými typmi komponent.

Vytvorenie užívateľskej komponenty

Užívateľské komponenty sa vytvárajú z pred definovaných Spark komponent. Užívateľskú komponentu teda tvorí jedna alebo niekoľko Spark komponent. Vytvoriť takúto komponentu bude možné pri označení jednej alebo niekoľkých za sebou nadväzujúcich Spark komponent. Graficky bude tento sled reprezentovaný ako spojitý graf.

Po výbere tohto sledu, užívateľ klikne na tlačidlo `Make component` dole na obrázku 6.7, kde sa mu zobrazí modálne okno, obrázok 6.11. V tomto okno sa ako prvé od používateľa vyžaduje názov jeho komponenty a krátky popis. Oba polia sú povinné. Názov komponenty



Obr. 6.10: Nastavenie komponenty s vlastným kódom.

sa vyžaduje jedinečný, ak sa užívateľ pokúsi zadať názov, ktorý už v systéme existuje, bude o tom informovaný chybovým hlásením.

Ďalej sú v paneloch vypísané jednotlivé Spark komponenty. Po kliknutí na danú komponentu sa zobrazia jej povinné a „keywords“ argumenty s možnosťou `Expose`. Pri zaškrnutí Expose niektorého z argumentov, to bude znamenať, že vo výslednej komponente budú pri nastavovaní hodnôt jednotlivých argumentov viditeľné práve tieto argumenty, obrázok 6.9.

Pred tým ako sa komponenta vytvorí, skontroluje sa, že či argumenty, ktoré sú povinné a neboli zaškrtnuté ako Expose, sú vyplnené.

Settings for component
Name of new component: (must be unique!)
Short description:
Input text
Arguments Expose
Position 0
minPartitions
use_unicode
flatMap
reduce
Close
Create component

Obr. 6.11: Modálne okno pri vytvorení užívateľskej komponenty.

6.2.4 Sledovanie stavu

Obrazovka sledovania stavu je dostupná na adrese `status`. Na tejto obrazovke, obrázok 6.12, sa pod hlavným menu zobrazuje stav jednotlivých relácií, ktoré má aplikácia vytvorené so serverom Livy. Všetky Spark úlohy, ktoré používateľ spustí sú vykonávané v niektorej z vybraných relácií. Ak je úloha vykonávaná v niektorej z nich, vytvorí sa v relácií ako

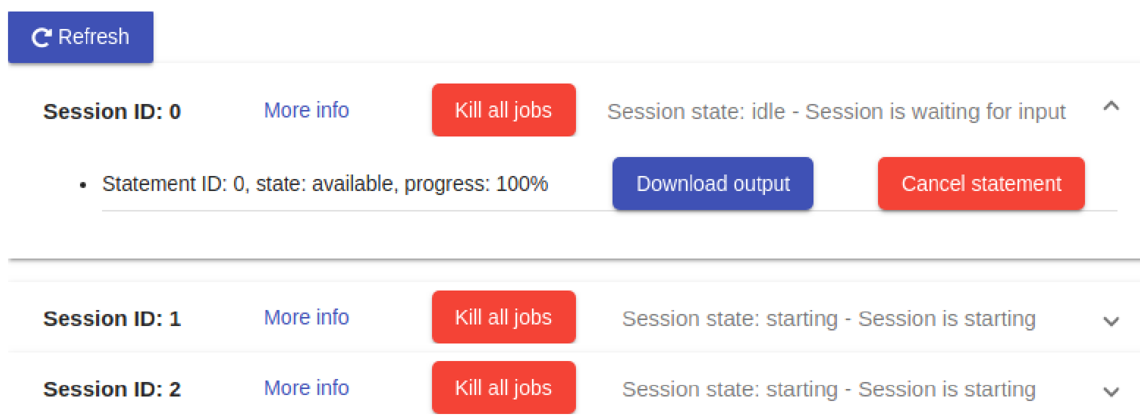
statement. Jeho stav môže používateľ sledovať po kliknutí reláciu, ktorá daný statement vykonáva.

Tlačidlo More info presmeruje používateľa na adresu, na ktorej Spark poskytuje vlastné webové rozhranie pre monitorovanie a detailnejšie sledovanie stavu aplikácie. Po kliknutí na tlačidlo sa užívateľovi otvorí nová záložka, na ktorej sa zobrazí webové užívateľské rozhranie Sparku.

Okrem sledovania stavu je možné zrušiť vykonávanie úlohy tlačidlo Cancel statement, poprípade je možné celé zmazanie danej relácie tlačidlo Kill all jobs, čím sa nielen zmaže daná relácia ale zastavia sa všetky vykonávané úlohy v danej relácií.

Ak úloha nevyžaduje ukladanie výstupných dát na HDFS, užívateľ má možnosť stiahnuť výpis štandardného výstupu vykonanej úlohy, môže sa jednať napríklad o výstup z funkcie print, či už celej úlohy alebo jej častí. Ak sa náhodou vyskytla vo výslednom kóde syntaktická alebo iná chyba, užívateľ to taktiež uvidí v tomto výpise. Tlačidlo sa zobrazí až keď bude úloha v stave available. So stavom sa priebežne zobrazuje aj koľko % je z úlohy už vykonané, čo reprezentuje údaj progress.

Aby užívateľ získal aktuálny stav jeho spojení s Livy serverom, v hornej časti obrazovky sa nachádza obnovovacie tlačilo Refresh, ktoré obnoví stav všetkých relácií a úloh.



Obr. 6.12: Obrazovka sledovania stavu úloh a spojení.

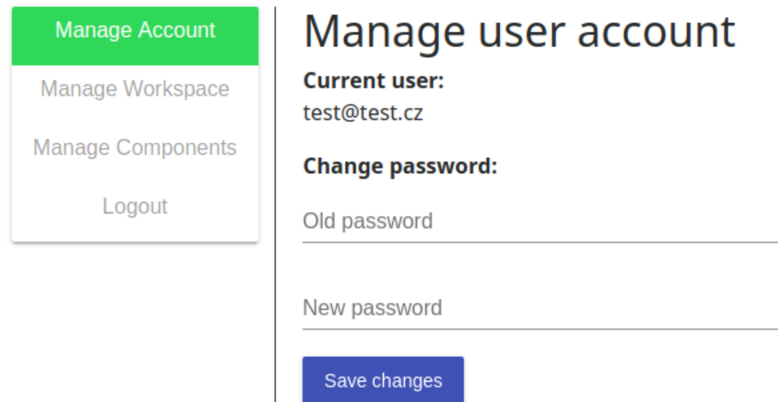
6.2.5 Správa profilu

Táto stránka je dostupná na adrese /profile. Užívateľ sa na ňu vie dostať kliknutím na odkaz v pravej časti hlavného menu, kde sa zobrazuje jeho emailová adresa.

V ľavej časti obrazovky sa nachádza menu pre túto časť stránky. Pri načítaní stránky sa ako prvé zobrazí časť Manage Account, rovnako sa aj zvýrazní zobrazovaná časť v pravom menu, obrázok 6.13. V tejto má užívateľ možnosť zmeniť si svoje prihlasovacie heslo do aplikácie.

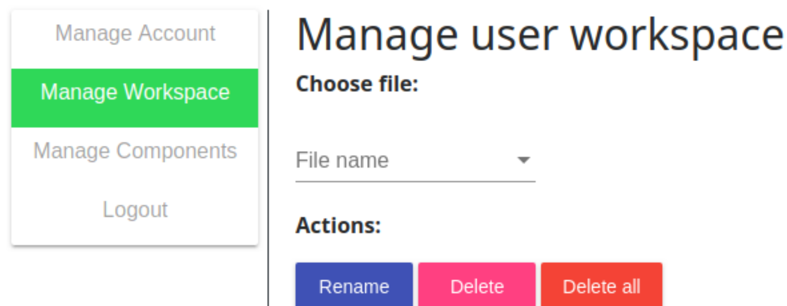
V druhej časti Manage Workspace, obrázok 6.14, má užívateľ možnosť upravovať si svoje uložené súbory. Pri kliknutí na list sa mu zobrazí zoznam jeho uložených súborov. Po výbere niektoré z nich má na výber z nasledujúcich akcií:

- Rename — po kliknutí sa v modálnom okne zobrazí input pre zadanie nového názvu zvoleného súboru, po potvrdení sa názov súboru zmení.
- Delete — zmaže súbor, ktorý užívateľ vybral.



Obr. 6.13: Stránka profilu - zmena hesla.

- `Delete all` — zmaže všetky uložené súbory daného používateľa.



Obr. 6.14: Stránka profilu – správa používateľových súborov.

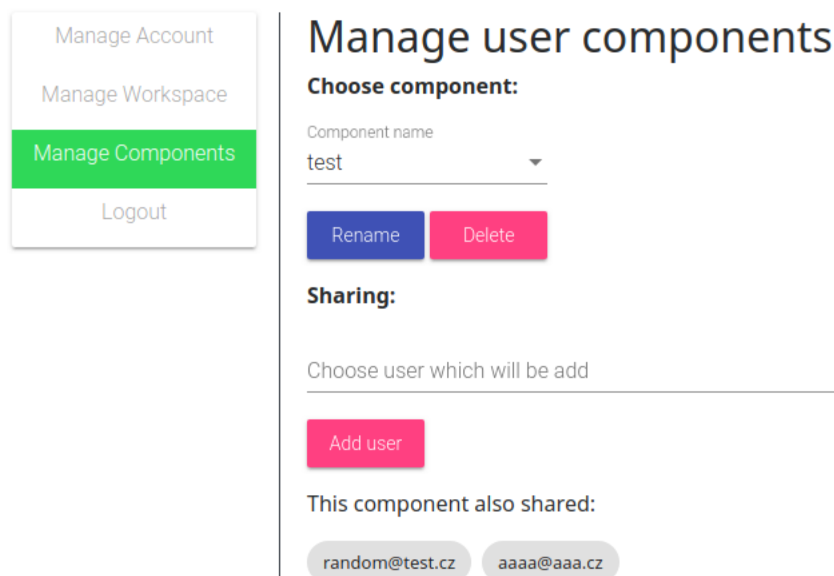
V tretej časti `Manage Components`, má používateľ niekoľko možností pre správu komponent, ktoré si sám vytvoril. Ako prvé si môže vybrať komponentu, s ktorou chce upravovať. Ďalšie vykonané akcie sa viažu na práve vybranú komponentu. Pomocou tlačidla `Rename`, podobne ako v časti Manage Workspace, má možnosť zmeniť názov komponenty. Tlačidlo `Delete`, umožňuje zmazanie danej komponenty.

V časti `Sharing`, má užívateľ možnosť vybrať si zo zoznamu používateľov, ktorí budú schopní komponentu používať, obrázok 6.15. Z prvého zoznamu si užívateľ vyberie užívateľa, podľa emailovej adresy, ktorému by chcel zdieľať svoju komponentu. Po tom, čo užívateľa vyberie, potvrdí pridanie tlačidlom `Add user`. Následne sa užívateľova emailová adresa pridá do zoznamu `This component also shared:`, ktorý zobrazuje všetkých používateľov, ktorý zdieľajú túto komponentu.

Poslednú časť ľavého menu na tejto stránke tvorí tlačidlo `Logout`. Ako z názvu vyplýva, po kliknutí bude užívateľ z aplikácie odhlásený a presmerovaný na stránku prihlásenia 6.2.

6.2.6 Dátové štruktúry

Aby bolo možné z vytvoreného grafu, ktorý predstavuje Spark úlohu generovať spustiteľný kód, je potrebné definovať dátovú štruktúru, ktorá bude niesť informácie o danej komponente. Navrhnutá dátová štruktúra by mala niesť informácie o tom pod akým názvom sa



Obr. 6.15: Stránka profilu – správa používateľom vytvorených komponent.

daná komponenta bude zobrazovať na pracovnej ploche, aký kód nesie, čiže čo je to za funkcia, aké ma parametre, či sú povinné alebo voliteľné a podobne.

Tak ako bolo popísané vyššie, užívateľ má k dispozícií preddefinované Spark komponenty a ďalej mu aplikácie umožňuje vytvárať aj vlastné. Na základe týchto požiadaviek bolo vytvorená dátová štruktúra, ktorej atribúty popisujú danú komponentu. Pre oba typy komponent platia tieto spoločné atribúty:

- `label` — textový identifikátor komponenty. Pod týmto názvom sa komponenta bude zobrazovať na pracovnej ploche.
- `type` — rozlišuje, či je komponenta Spark alebo vytvorená užívateľom. Nadobúda hodnoty spark_component alebo user_component.
- `description` — dlhší slovný popis danej komponenty.

Ďalej pre Spark komponentu sú definované ďalšie atribúty a to:

- `class` — do akej triedy daná metóda patrí. Na základe tohto atribútu sa komponenta priradí do do zoznamu Spark komponent pod danú triedu.
- `returnType` — dátový typ, ktorý daná komponenta vracia.
- `applicableOn` — na aký dátový typ je možné komponentu (funkciu) aplikovať.
- `applyOn` — tento atribút hovorí, že ak sa pred daným uzlom v grafe nenachádza žiaden ďalší, tak sa pokúsi aplikovať funkciu z daného uzlu na túto hodnotu. Jedná sa väčšinou o premennú SparkContex (sc).
- `code` — jedná sa o zložený atribút, ktorý obsahuje ďalšie pod atribúty na niekoľkých úrovniach. Na prvej úrovni je typ, názov programovacieho jazyka, ktorý daný atribút nesie. Môže to byť napríklad typ `pyspark3, scala` a podobne. V tejto práci budeme používať typ pyspark3.

Hodnoty tohoto atribútu sú ďalšie atribúty na nižšej úrovni. Konkrétne sa jedná o atribút `rawCode`, ktorého hodnota je kód, komponenty v danom programovacom jazyku. Táto hodnota sa bude používať pri generovaní výsledného kódu. Ďalší atribút je `requeired_arguments`, ktorý obsahuje list hodnôt povinných argumentov danej komponenty. Každá položka listu obsahuje aký dátový typ sa pre daný argument očakáva (`type`) a na ktorej pozícií zľava sa argument nachádza (`position`). Posledným atribútom na tejto úrovní je atribút `kwargs`, ktorý popisuje tzv. „keyword arguments" danej komponenty. Jedná sa tiež o list, kde každá položka listu obsahuje atribút `name`, názov argumentu a ďalší atribút `default_value`, ktoré hodnota je defaultná hodnota pre daný argument. Ukážka takejto dátovej štruktúry sa nachádza v prílohe B.1.

Užívateľom vytvorené komponenty potom budú komponenty, vytvorené z jednej alebo viacerých Spark komponent. Táto komponenta bude teda obsahovať základné atribúty a k nim pribudne jeden atribút:

- `node` — jedná sa o list Spark komponentu, ktoré tvoria danú užívateľskú komponentu.

Aby bolo možné na pracovnej ploche jednoznačne identifikovať komponentu, je nutné pridať ďalší atribút. Kedže Spark úlohy budú tvorené spojitým grafom, bude potrebné pridať atribút, ktorý bude hovoriť s ktorým uzlom je daný uzol spojený. Na tieto účely budú vytvorené atribúty:

- `nodeId` — jednoznačný identifikátor uzlu v grafe.
- `connectToNode` — zoznam `nodeId`, kam všade je daný uzol pripojený.

Návrh popísaný v tejto časti zabezpečí, že výsledná Spark úloha bude dátovo reprezentovaná ako list, ktorý obsahuje jednotlivé komponenty, či už užívateľom definované alebo Spark, ktoré navzájom na seba ukazujú cez atribút `connectToNode`.

Príklad tejto dátovej štruktúry, užívateľom vytvorenej komponenty sa nachádza v prílohe B.2

Kapitola 7

Návrh serverovej časti aplikácie

Táto kapitola je venovaná návrhu serverovej časti. Jej hlavnou úlohou bude pre-posielanie požiadaviek z aplikácie na server Livy z dôvodu absencie nastavenia CORS[1] na strane Livy serveru.

Server bude využívaný pre autorizáciu používateľov, ukladanie používateľských úloh, uloženie Spark komponent a aj užívateľom vytvorených komponent. Ako budú dáta reprezentované v databáze, bude v tejto kapitole znázornené diagramom, ktorý popisuje dátový model aplikácie.

Následne bude navrhnuté REST rozhranie, pomocou ktorého bude webová aplikácia komunikovať so serverom.

7.1 Pre-posielanie požiadaviek

Jednou z funkcií, ktorú bude server vykonávať je pre-posielanie požiadaviek z webovej aplikácie na server Livy. Bude sa jednať o všetky požiadavky, ktorých `PATH` časť URL adresy začína na `/seesions*`.

Dôvod, prečo využijeme server na pre-posielanie požiadaviek je, ako bolo spomenuté vyššie, absencia nastavenie CORS na strane serveru Livy. Ide o to, že ak webová aplikácia beží na doméne `domain-a.com`, v našom prípade je to (localhost:4200) a potrebuje komunikovať zo serverom, ktorí beží na doméne `domain-b.com`, v našom prípade (localhost:8998), musí mať nato oprávnenie, ktoré kontroluje webový prehliadač na základe HTTP hlavičiek [13].

Aplikácia získa oprávnenie z hlavičiek odpovedí HTTP správy `OPTIONS`. Tieto hlavičky bude pridávať práve tento server.

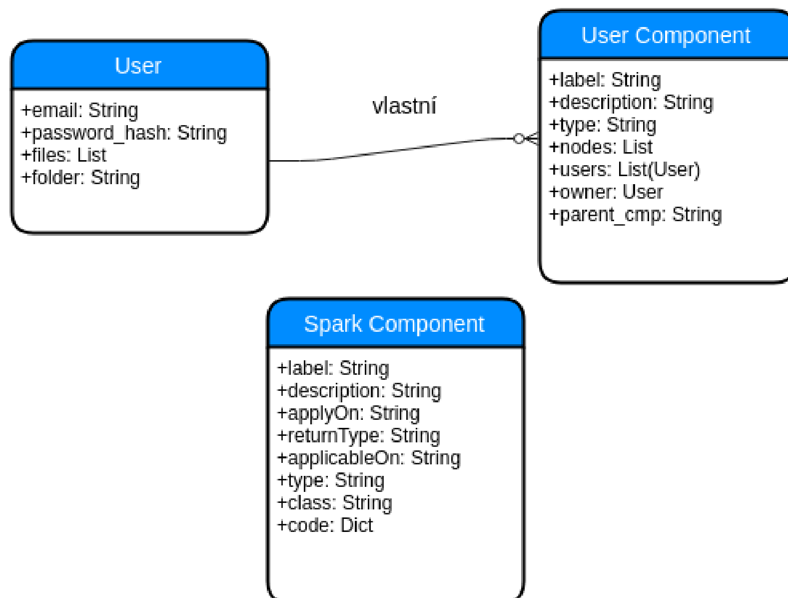
Toto obmedzenie aplikujú webové prehliadače kvôli bezpečnosti.

7.2 Dátový model

Uloženie dát na serveri predstavuje dátový model 7.1. Vychádza z požiadaviek na dátovú časť aplikácie opísanej 6.2.6 a taktiež diagramom prípadu použitia, opísanom v časti 6.1. Pridáva model používateľa (User), ktorý sa bude voči serveru autentizovať svojím emailom a heslom. Kvôli bezpečnosti sa heslo nebude ukladať ako čistý text, ale bude z neho vygenerovaný hash.

[1] Cross-Origin Resource Sharing - https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Dátová reprezentácia užívateľskej komponenty na serveri si vyžaduje pridať ďalšie atribúty, aby bolo možné nastavovať zdieľanie týchto komponent. Každá užívateľom vytvorená komponenta má zoznam používateľov, ktorí ju používajú, atribút `users`. Ďalej atribút `owner`, ktorý identifikuje komu komponenta patrí a identifikátor tej komponenty, z ktorej bola táto komponenta vytvorená, reprezentované atribútom `parent_cmp`.



Obr. 7.1: Dátový model aplikácie.

7.3 REST API serveru

Aby mohla webová aplikácia pohodlne komunikovať so serverom, poskytne jej okrem preposielania požiadaviek na server Livy aj ďalšie rozhrania, pomocou ktorého bude schopná autentizovať, autorizovať používateľa, čítať a ukladať dáta z aplikácie na server.

Server definuje tieto REST volania:

- `POST /login` — telo požiadavku (request), kedže sa jedná o POST metódu, očakáva email a heslo používateľa (položky vo formáte JSON `email, password`). Pri odbavovaní požiadavky server skontroluje správnosť používateľských údajov, ak sa zhodujú s tými, ktoré sú uložene v databáze, server vráti vygenerovaný `token`, ktorého platnosť vyprší za jeden rok a k tomu aj databázový identifikátor používateľa.

  Ak sa prijaté údaje nezhodujú s tými databázovými, server vráti príslušný návratový kód.

- `GET /spark_components` — vracia všetky Spark komponenty, ktoré sú uložené v databáze.

- `GET /user/<id>` — vráti záznam používateľa, kde `<id>` je databázový identifikátor používateľa.

- `POST /user/<id>/files` — uloží súbor, tzn. úlohu, ktorú vytvoril užívateľ s identifikátorom `<id>` v aplikácií, do adresára, ktorý je asociovaný s daným používateľom.

V tele požiadavku sa vyžadujú atribúty filename, pod akým názvom sa súbor uloží a atribút files, kde sa nachádza JSON reprezentácia dát, ktoré klient na server posiela a ktoré budú v súbore uložené. V tele odpovede sa nachádza URL adresa, na ktorej je práve uložený súbor dostupný a taktiež názov súboru.

- GET /user/<id>/files/<filename> — vráti úlohu v JSON formáte, ktorú užívateľ vytvoril. <id> je identifikátor používateľa a <filename> je súboru, v ktorom je úloha uložená na disku.
- PATCH /user/<id>/changePassword — zmení heslo používateľovi s identifikátorom <id>. V dátach požiadavku sa nachádza ako staré, tak aj nové heslo.
- PATCH /user/<id>/renameWorkspace — zmení názov uloženého súboru používateľa s identifikátorom <id>. V dátach požiadavku sa nachádza názov dokumentu, ktorý chce užívateľ zmeniť a jeho nový názov.
- DELETE /user/<id>/workspace/<name> — zmaže súbor s názvom <name> užívateľovi s identifikátorom <id>.
- DELETE /user/<id>/workspace — zmaže všetky súbory používateľa s identifikátorom <id>.
- GET /user_components/<id> — vráti všetky užívateľské komponenty, ktoré užívateľ vytvoril alebo mu ich od iného používateľa zdieľa. Časť adresy <id> je databázový identifikátor používateľa.
- POST /user_components/<id> — uloží do databáze užívateľom vytvorenú komponentu. <id> v adrese je identifikátor používateľa, ktorý komponentu práve vytvoril.
- PATCH /user_components/<id>/rename/<new_name> — zmení názov komponenty s identifikátorom <id> na názov <new_name>.
- DELETE /user_components/<id> — odstráni komponentu s identifikátorom <id>.
- POST /user_components/<id>/add_user — vytvorí kópiu komponenty užívateľskej komponenty, ktorá ma identifikátor <id>. V dátach požiadavku sa nachádza emailová adresa, na základe ktorej sa vyhľadá užívateľ.
- GET /user_components/who_shared/<id> — vráti emailové adresy všetkých používateľov, ktorí zdieľajú užívateľskú komponentu s identifikátorom <id>.
- PATCH /user_components/<id> — zmení položky v atribúte nodes užívateľskej komponente s identifikátorom <id>.

Kapitola 8

Implementácia

Táto kapitola je venovaná implementácií tejto práce. V prvej časti autor opisuje implementáciu webovej časti práce, konkrétne aké technológie boli pri tvorbe využité, ako sa spracúvajú dáta od používateľa, generovanie kódu z vytvorenej Spark úlohy, vytvorenie užívateľskej komponenty a podobne. Tiež sú poznamenané obmedzenia, s ktorými sa v návrhu nepočítalo a boli objavené až v čase implementácie.

V druhej časti je popísané, ako bol implementovaný webový server, použité technológie a podobne.

8.1 Implementácia webovej časti aplikácie

Na implementáciu súčasných webových aplikácií je vhodné siahnuť po niektorom zo súčasných webových frameworkov, ktorý môže značne uľahčiť a urýchliť vývoj. Pre túto prácu bol zvolený JavaSriptový framework Angular[1] vo verzii 5.0.1. Vývoj tohto open-source frameworku ide rýchlo dopredu, autori stále na ňom pracujú, vychádzajú nové aktualizácie, v týchto dňoch[2] vychádza verzia 6.

Angular aplikácia sa skladá z modulov, ktoré obsahujú množiny komponent [6]. Aplikácia môže byť zložená z niekoľkých modulov, no musí existovať minimálne jeden. Modul tvorí spoločný menný priestor pre komponenty, ktoré sú v ňom obsiahnuté. Okrem spomínaných komponent, môže obsahovať aj ďalšie služby, ktoré využívajú komponenty.

Komponenty tvoria základné kamene aplikácie. Skladajú sa z dvoch častí, prvou je časť s `typescript` kódom, kde sú definované metódy, premenné atď., jednoducho sa jedná o kontrolér a druhú časť tvorí HTML kód, ktorý môže zobrazovať obsah premenných z kontroléra, zbieranie dát v podobe formulárov od používateľa a podobne. Súčasťou je aj súbor s obsahom CSS pre nastavenie štýlov.

Framework poskytuje možnosť vytváranie vlastných direktív, ktoré je ďalej možné používať v HTML súboroch.

Keďže sa jedná o webový framework poskytuje vlastný routovací modul, ktorú sa stará o to, aby sa na URL adresách sa zobrazovali správne komponenty. Ďalej poskytuje mnoho ďalších možností, ako sú napríklad tzv. `guard`, ktorý môže slúžiť na kontrolu toho, že má daný užívateľ práva vidieť túto stránku.

[1] https://angular.io/

[2] 8.5.2018

HTML elementy ako sú tlačidla, zoznam, vstupy formulárov a podobne bol použitý Angular Material Design[3]. O rozloženie elementov na stránke sa stará modul Angular Flex-Layout[4] doplnený o ďalšie vlastné CSS štýly.

8.1.1 Grafová reprezentácia úlohy

Samotný Angular pre potreby tejto aplikácie nestačí. Vytvorenie vlastného riešenia pre zobrazenie grafovej časti aplikácie by bolo veľmi časovo náročné, preto bola zvolená cesta už existujúcich riešení. Pre tento účel bola vybraná knižnica `vis.js`[5], konkrétne jeden z jej modulov, `Network`.

Samotná knižnica sa stará o zobrazenie, manipuláciu, pridávanie, odoberanie uzlov a podobne.

Aby bolo možné pohodlne využívať metódy, ktoré poskytuje knižnica `vis.js`, bola implementovaná trieda `GraphService`, ktorá sa stará o komunikáciu s touto knižnicou. Zabezpečuje to, aby bol každý uzol v grafe jednoznačne identifikovateľný, komunikuje s ďalšími komponentami aplikácie, tým že dáva vedieť o novo pridaných uzloch a hranách, prípadne ich o odstránení.

8.1.2 Vytváranie dátovej štruktúry

Zatiaľ čo o zobrazenie uzlov a hrán sa stará trieda `GraphService`, pre vytvorenie dátovej štruktúry, ktorá nesie informácie pre vytvorenie Spark aplikácie má na starosti trieda `CodeCreator`.

Podľa toho, ako bolo v časti 6.2.6 navrhnuté dátové štruktúry aplikácie, tak podľa toho sa v tejto triede udržiava celá Spark úloha, reprezentovaná vo formáte JSON. Konkrétne sa jedná o list objektov, ktoré nesú informácie o komponente, ktorú reprezentujú. Každá položka listu je jeden uzol grafu.

Dôraz sa kladie na udržanie konzistencie medzi tým, čo sa zobrazuje v grafe a tým, čo je uložené v tejto dátovej štruktúre. Preto všetky zmeny, ktoré sa udejú na zobrazovacej úrovni sa musia premietnuť aj do tej dátovej. Jedná sa hlavne o mazanie, pridávanie uzlov/hrán a v neposlednom rade aj vytváranie užívateľských komponent. Pri týchto operáciach sa menia položky v atribúte `connectToNode`, prípadne sa pridávajú ale odstránia celé záznamy.

Správnosť údajov, ktoré uchováva táto trieda je nutná pre správne generovanie kódu popísané v časti 8.1.4. Užívateľ si môže aktuálny stav kódu reprezentovaného touto štruktúrou pozrieť kliknutím na tlačidlo `Show JSON Code`, popísaný 6.2.3.

8.1.3 Vytvorenie užívateľských komponent

Pre vytváranie užívateľských komponent bola vytvorená trieda `ComponentCreator`. Stará sa o celý proces vytvorenia užívateľskej komponenty.

Pre túto prácu sa obmedzíme na to, že komponenty ktoré užívateľ smie vytvárať sú vždy spojité časti grafu. Je to z dôvodu, že užívateľské rozhranie, konkrétne reprezentácia komponent ako uzlov grafu, nevraví nič o tom, čo je „vo vnútri" komponenty, koľko má vstupov/výstupov a podobne.

Preto pred každým vytvorením sa kontroluje, že či je komponentu možné vytvoriť, tým že sa kontrolujú spojenie vybraných uzlov. Na začiatku sa zistí, ktoré uzly boli užívateľom

[3]https://material.angular.io/
[4]https://github.com/angular/flexlayout
[5]http://visjs.org

vybrané. Ak užívateľ vybral uzly, ktoré netrovia spojitý graf, bude o tom informovaný chybovým hlásením. Ak kontrola prebehla správne, zobrazí sa mu modálne okno popísané v časti 6.2.3.

Po vyplnení názvu komponenty, krátkeho popisu, výberu argumentov, ktoré chce užívateľ potom explicitne zobrazovať pri nastavení hodnôt pre argumenty, kliknutím na tlačidlo `Create component`, prebehne proces vytvorenia, kde sa vytvorí nová užívateľská komponenta, ktorá obsahuje v atribúte `nodes`, všetky vybrané Spark komponenty.

Ďalej sa pri vytváraní kontroluje, že či argument, je povinný a nebol označený ako `expose`, má nastavenú nejakú hodnotu. Je to z dôvodu, že užívateľ explicitne neuvidí, akú hodnotu má daný argument nastavenú.

Po úspešnom vytvorení sa v liste, ktorý nesie všetky uzly prítomné v grafe, odstránia položky uzlov, z ktorých bude vytvorená nová komponenta, tie sa stanú súčasťou novo vytvorenej užívateľskej komponenty.

Na ploche grafu sa tak isto odstránia vybrané uzly so svojimi hranami a budú nahradené jedným uzlom, ktorý reprezentuje užívateľskú komponentu, poprípade aj s doplnením o hrany, ktoré z pôvodných komponent mali prvý a posledný uzol.

Dáta novo vytvorenej komponenty sa odošlú na server a užívateľovi sa zobrazia v menu pre výber komponent pod hlavičkou `User Define Components`, kde pridaná položka má názov, ktorý užívateľ zadal v modálnom okne pri vytvorení.

8.1.4 Generovanie kódu

Generovanie kódu patrí medzi najdôležitejšie funkcie webovej časti aplikácie. Úlohou je vygenerovať z vytvorenej dátovej štruktúry kód, ktorý odoslaný na server Livy a ten ho ďalej predá Sparku na spracovanie. V tejto časti sú implementované aj kontroly typov. Kontroluje sa najmä `správnosť spojenia` jednotlivých komponent, poprípade či sú vyplnené povinné argumenty a podobne.

Pred tým ako užívateľ spustí (odošle) vygenerovaný kód na server Livy, môže si overiť správnosť jeho úlohy (na tej úrovni, ktorú podporuje webová aplikácia), tlačidlom `Check PySparkCode`. Tým sa spustí generovanie kódu, bez toho aby sa niečo posielalo na vykonávanie. Užívateľ sa tak môže pozrieť na výsledný kód.

Samotné generovanie má na starosti trieda `CodeGenerator`. Najprv sa nájde uzol, do ktorého nevedia žiadna hrana, štartovací uzol. Generovanie začína tak, že sa najprv zistí o aký typ komponenty sa jedná, spark, user, alebo komponenta s vlastným kódom a na základe toho sa volajú funkcie, ktoré sa postarajú o spracovanie.

Pri spracovaní komponenty sa prihliada nato, že či sa jedná o Spark komponentu alebo užívateľskú komponentu. Aby sa predišlo kolízií názvov premenných, výstup Spark komponenty sa vytvorí ako out_node_<id>, kde `<id>` je identifikátor komponenty, narozdiel výstup komponenty, ktorá je vnútri užívateľskej komponenty sa vytvorí ako out_cmp_node_<id>, kde `<id>` je identifikátor v rámci užívateľskej komponenty.

Bola implementovaná aj kontrola názvov výstupov. Ak sa výstup s daným názvom už vo výslednom reťazci nachádza, bude na koniec názvu premennej prikonkatenované náhodne vygenerované číslo. Výstup je potom v tvare out_node_<id>_<int> alebo out_cmp_node_<int>, kde `<int>` je náhodne vygenerované číslo.

Po vytvorení správneho názvu výstupnej premennej sa zisťuje, že či pre aktuálne spracovávaný uzol existuje nejaký vstup. Ak nie, tak funkcia ktorú uzol nesie sa aplikuje na hodnotu, ktorá má uzol v atribúte `applyOn`.

Naopak, ak pre uzol vstup existuje, overí sa aký typ premennej je možné aplikovať na funkciu, ktorú nesie uzol, čo predstavuje hodnota v atribúte `aplicableOn` s typom, ktorý vracia prechádzajúci uzol. Ak typy sedia, vytvorí sa kód, ktorý vyzerá ako <previousOutput>.<actualFunction>, kde hodnota `<previousOutput>` je názov výstupnej premennej prechádzajúceho uzla a hodnota `<actualFunction>` je funkcia, ktorú nesie aktuálny uzol. Následne sa spracujú argumenty daného uzla, overí sa či sú zadané všetky povinné a podobne.

Niektoré komponenty nemusia aplikovať svoju funkciu na výstup prechádzajúcej komponenty, ale spracujú výstup ako argument svojej funkcie. Pre tieto prípady je vytvorený checkbox `Previous node`, popísaný 6.2.3, po ktorého zaškrnutí si užívateľ zvolí výstup, z ktorého uzla chce použiť ako tento vybraný argument. Výstup z takejto komponenty môže potom vyzerať ako <previousOutput1>.<actualFunction>(<previousOutput2>), kde `<previousOutput*>` sú vstupy do aktuálneho uzla, jeden z nich je spracovaný ako argument.

Podobne sa spracúvajú aj užívateľské komponenty, s tým rozdielom že sa rešpektujú obmedzenie popísané v časti 8.1.3.

Špeciálny typ uzla `own_component`, popísaný 6.2.3, ktorý nesie vlastne vytvorený kód, sa jednoducho konkatenuje na výstup v poradí, v akom bol umiestnený v grafe a pokračuje sa so spracovaním ďalej.

8.1.5 Pripojenie na Livy server

Aplikácia vytvára relácie s Livy serverom pomocou tlačidla `Connect to Livy`. Najskôr sa zistí, že či neexistuje nejaká relácia, do ktorej by sa dalo pripojiť. To znamená, že je v stave `idle` alebo `busy`. Je to z toho dôvodu, že vytvorenej novej relácie na lokálnom počítači je náročne na zdroje, s pridaním ďalšej sa značne spomaľuje odozva celého systému.

Ak však neexistuje žiadna relácia, ktorá vyhovuje podmienkam, vytvorí sa nová s parametrami, ktoré sú vypísané 8.1.5. Parametre hovoria o tom, ako Spark spustí tzv. „spark-submit“, popísaný tu 3.3.

- `driverMemory: '2g'` — množstvo pamäte, ktoré použije master uzol.
- `driverCores: 1` — počet jadier, ktoré použije master node.
- `executorMemory: '1g'` — množstvo pamäte, ktoré použijú tzv. worker uzol.
- `executorCores: 1` — počet jadier, ktoré využije worker uzol.
- `numExecutors: 2` — počet worker uzol

8.2 Implementácia serverovej časti aplikácie

Aby bolo možné odbavovať požiadavky prichádzajúce z aplikácie je potrebné mať k tomuto účelu implementovaný webový server. Aby sme sa vyhli zdĺhavej implementácii vlastného webové servera, ktorý by spĺňal naše požiadavky, využijeme jeden z webových frameworkov, ktoré by sa mohol na tieto účely hodiť. [28]

Webový framework tvorí množinu komponent, ktorá je navrhnutá tak, aby urýchlila a uľahčila vývojový proces. Obsahuje základné kamene, bez ktorých sa ťažko zaobídeme. Umožňuje sa zameriavať na ciele projektu, namiesto vytváranie vecí, ktoré už niekto pred nami vyriešil a sú pre náš projekt nepodstatné.

Framework, ktorý budeme v práci používať by mal spĺňať isté kritéria. Mal by byť schopný prijímať a vracať požiadavky, poprípade ich preposielať ďalej. Malo by sa jednať o dáta, ktoré potrebuje aplikácia, poprípade dáta o užívateľoch. Ďalej by mal byť schopný pripojiť sa do databáze, odkiaľ si načíta dáta o užívateľoch, poprípade načítať alebo zapísať súbor na disk. Z toho vyplýva, že pre túto aplikáciu bohate postačí jednoduchý webový framework, ktorý tieto požiadavky bude spĺňať. V súčasnosti existuje množstvo frameworkov, z ktorých si vieme vyberať. Nájsť ten správny môže byť neľahká úloha, vzhľadom k tomu, že ich je v súčasnosti veľa. Pomôcť zredukovať množstvo nám môže pomôcť výber jazyka, v ktorom je framework implementovaný.

Ak sa pre účely tejto práce obmedzíme na programovací jazyk Python, môžeme využiť rôzne frameworky implementované v tomto jazyku ako sú napríklad Django, Pyramid, Flask, Tornado, Bottle atď.. Podľa [7] sú tri najpoužívanejšie Django, Pyramid a Flask. Pre účely tejto práce využijeme framework Flask 8.2.1.

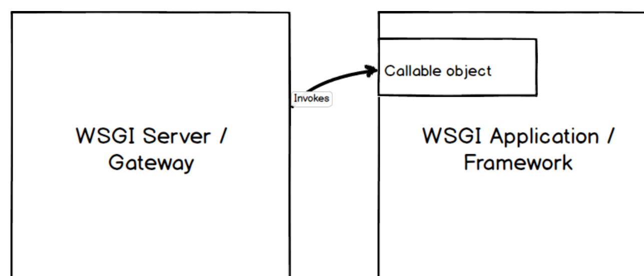
8.2.1 Python Flask

Python Flask [25] je webový micro framework. Micro znamená, že framework si udržiava jednoduché jadro ale je jednoducho rozširovateľný, napríklad Flask nemá preddefinovanú databázu, ktorú má používať, aký templatovací engine sa má používať atď. Všetko je na používateľovi, je možné používať všetko alebo nič.

Defaultne, Flask nezahŕňa žiadnu databázovú vrstvu, validáciu formulárov alebo čokoľvek iné na čo už existujú knižnice, ktoré sa o to vedia postarať. Narozdiel od toho, Flask podporuje rozšírenia, ktoré dodajú takú funkcionalitu aplikácií, ako keby bola implementovaná v samotnom jadre. Mnoho rozšírení ponúka integráciu databázy, validáciu formulárov, niekoľko autentifikačných technológií atď..

Flask je WSGI aplikácia [19]. WSGI je Web Server Gateway Interface, čo je rozhranie, ktoré umožňuje spúšťať Python aplikácie na strane servera. V tradičných webových serveroch neexistoval spôsob ako spustiť Python aplikáciu. V 90. rokoch minulého storočia, bol vytvorený Apache module nazývaný mod_python [6], ktorý umožňoval spúšťanie ľubovoľného Python kódu.

Keď vývoj mod_python bol zastavený a objavili sa zraniteľnosti, Python komunita sa rozhodla, že je potrebné vytvoriť konzistentný spôsob pre spúšťanie Python aplikácií. Tak bolo naimplementované WSGI ako štandardné rozhranie, ktoré moduly a kontainery môže využívať. WSGI server jednoducho invokuje callable objekt WSGI aplikácie, obrázok 8.1, tak ako to definuje štandard PEP 3333 [7].



Obr. 8.1: Ako funguje WSGI aplikácia (Zdroj: https://www.fullstackpython.com/wsgi-servers.html)

[6] https://grisha.org/blog/2013/10/25/mod-python-the-long-story/
[7] https://www.python.org/dev/peps/pep-3333/

Aby Flask vedel správne odbaviť požiadavok prichádzajúci z webovej aplikácie, musíme zabezpečiť, aby pre konkrétnu URL bol definovaný tzv. endpoint. V tejto aplikácii využijeme tzv. Blueprint, ktorý poskytuje priamo Flask, pre obsluhu požiadaviek, ktoré sú ďalej preposielané na server Livy.

Ďalšie endpointy, ktoré sú popísané v časti 7.3, implementujeme pomocou rozšírenia Flask-Classy[8]. Toto rozšírenie nám umožní pohodlné vytváranie REST volaní pre webovú aplikáciu. Na ukážke 8.1 je endpoint, implementovaný pomocou Flask-Classy, ktorý je dostupný na adrese `/user/<id>/files`, pre metódu POST, kde `<id>` je identifikátor používateľa a zároveň sa povinný argument pre funkciu `store_file()`.

```
from flask_classy import FlaskView, route

class UserView(FlaskView):
    route_base = '/user'
    trailing_slash = False

    @route('/<id>/files', methods=['POST'])
    @private_source
    def store_file(self, id):
    ...
```

Výpis 8.1: Príklad endpointu implementovaného pomocou Flask-Classy.

8.3 Databáza

Aby bolo možné pohodlne pracovať s dátami uloženými na serveri, bolo potrebné implementovať dátové úložisko. V súčasnosti existuje množstvo dátových serverov (databáz), ktoré sa odlišujú tým, akým spôsobom je možné dáta ukladať, aké dáta uchovávajú a vytvárajú.

V tomto kontexte máme na výber z dvoch typov databáz: relačné a objektové [23]. Zatiaľ čo relačné databáze organizujú dáta do matematických relácií, objektové naopak majú skôr dokumentový charakter a môžu obsahovať dáta ľubovoľného formátu, ktorý sa môže dynamicky meniť.

V tejto aplikácií je množstvo štruktúrovaných dát. Či už sa jedná o Spark komponenty alebo užívateľom definované komponenty, oba obsahujú zložitejšie štruktúry, ktoré by bolo popisovať pomocou relačnej databázy komplikované. Preto padla voľba na databázu objektovú a to konkrétne na MongoDB[9].

Pre pohodlnú prácu s databázou bol použitý mongoengine[10], ktorý poskytuje rozhranie pre prácu s Mongom. Ako už bolo spomínané vyššie, objektové databázy sú typické tým, že uloženie dát má dokumentový charakter. Ako môže vyzerať dokumentu, definovaný pomocou mongoengine `Documentu`, ktorý uloží Spark komponentu, je na ukážke 8.2.

Hodnota v atribúte meta, hovorí do akej kolekcie v databáze sa budú tie dokumenty ukladať. Je možné definovať rozdielne názov polí reprezentovaného v databáze a v Python reprezentácií, hodnota `db_field`.

Na základe dátového modelu 7.1, boli pomocou tohoto rozhrania implementované triedy modelov, ktoré reprezentujú ako budú uložené dáta v jednotlivých dokumentoch v databáze.

[8]https://pythonhosted.org/Flask-Classy/
[9]https://www.mongodb.com/
[10]http://mongoengine.org/

```python
from mongoengine import Document

class Component(Document):
    """
    Spark component
    """
    meta = {
        'collection': 'Component'
    }
    label = StringField(required=True)
    applyOn = StringField()
    returnType = StringField()
    applicableOn = StringField()
    comp_type = StringField(default="spark_component", db_field='type')
    spark_class = StringField(db_field='class')
    code = DictField()
    description = StringField()
```

Výpis 8.2: Príklad modelu v mongoengine.

8.4 Autentizácia, autorizácia

V súčasnosti sa pri webových aplikáciach kladie dôraz na bezpečnosť. Z tejto oblasti nás v tejto práci hlavne zaujíma autentizácia – overenie identity používateľa, teda o koho sa jedná a autorizácia – overenie, že daný užívateľ má povolenie čítať/manipulovať s daným zdrojom.

Pre tieto účely bola využitá metóda JWT[11], ktorá je popísaná v štandarde [17].

V krátkosti to funguje tak, že pri prihlásení, sa vygeneruje z JSON objektu (objekt obsahuje väčšinou dáta o používateľovi prípadne, čas generovania, čas ukončenia platnosti a podobne) token, (hash), pomocou niektorého šifrovacieho algoritmu (v tomto prípade RSA[12]).

Takto vygenerovaný token sa pošle v odpovedi používateľovi späť do aplikácie a tá si ho uloží (najčastie do `localStorage`, ak sa jedná o webovú aplikáciu). Následne sa s každým požiadavkom, ktorý by mal byť autentizovaný posiela v hlavičke `Authorization` ako hodnota `Bearer <token>`. Na základe toho server užívateľa autentizuje, poprípade ho autorizuje.

V tejto práci bol použitý PyJWT[13], samotná implementácia je popísaná na obrázku 8.2.

8.5 Zdieľanie komponent

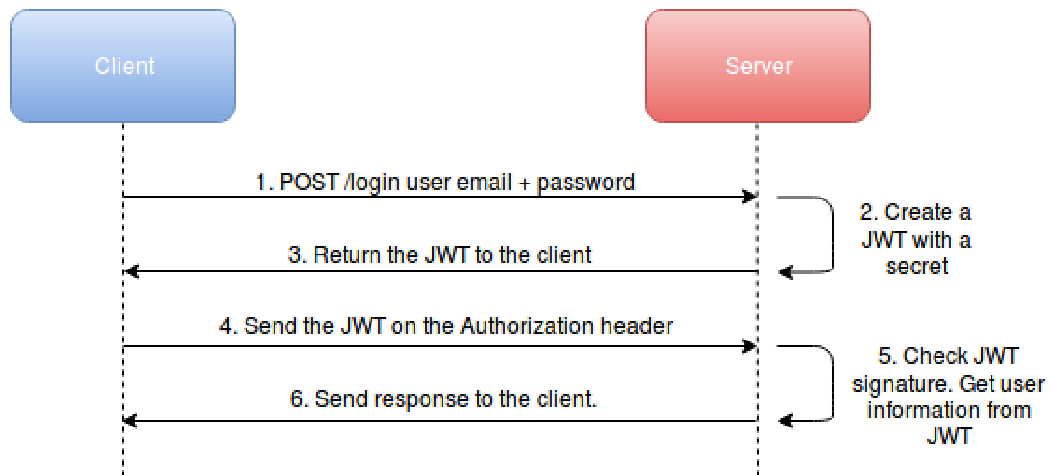
Zdieľanie komponent medzi užívateľmi implementuje serverová časť aplikácie. Aby sa zachovalo to, že každý používateľ si bude upravovať svoju kópiu komponenty, tak každému, komu bude dané povolenie s komponentou pracovať, tzn. komponenta mu bude zdieľaná, sa vytvorí nová kópia pôvodnej komponenty.

Nová kópia komponenty nastaví atribút `owner`, ktorý bude obsahovať referenciu užívateľa, ktorému bolo pridané „zdieľanie“, čiže sa stal majiteľom komponenty. Ďalej sa podľa

[11] JSON Web Token

[12] Rivest–Shamir–Adleman – názov dostal podľa priezvisk autorov, ktorí algoritmus prvý krát popísali v roku 1978.

[13] https://pyjwt.readthedocs.io/en/latest/



Obr. 8.2: Grafický popis implementácie JWT. (Zdroj: https://jwt.io/)

atribútu `parent_cmp`, ktorí obsahuje komponenta, z ktorej sa nová vytvára, vyberú všetky komponenty ktoré majú hodnotu tohoto atribútu rovnakú a do poľa `users` sa pridá referencia na používateľa, ktorý práve získal zdieľanie komponenty. Na základe týchto hodnôt potom vieme jednoducho zistiť, kto ďalší zdieľa túto komponentu.

Novej komponente sa taktiež nastavia aktuálne hodnoty `parent_cmp` a `users`.

Kapitola 9

Demonštrácia výsledkov

V tejto časti bude na typovej úlohe predstavený nástroj, ktorý bol implementovaný v tejto práci. Ako príklad bola zvolená Spark úloha `Word Count`, ktorá je dostupná na stránke Sparku ako príklad pre prácu s RDD[1].

Textovou formou bude vysvetlené ako sa užívateľ dostane k požadovanému výsledku. Grafická ukážka (video) je súčasťou vloženého DVD v tejto diplomovej práci. Ukážka je predvedená v prehliadači `Google Chrome Version 65.0.3325.181 (Official Build) (64-bit)`, na operačnom systéme `Linux Mint 18.3 Cinnamon 64-bit`.

9.1 Wordcount

Aplikácia Wordcount – čiže počítanie počtu slov vo vstupnom súbore. Kód aplikácie, ktorý budeme vytvárať vyzerá následovne:

```
text_file = sc.textFile("hdfs://localhost:54310/words.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://localhost:54310/wordCount.txt")
```

Výpis 9.1: Aplikácia wordcount. (Zdroj: https://spark.apache.org/examples.html)

9.1.1 Vytvorenie a spustenie úlohy

Pri načítaní aplikácie sa pre neprihláseného používateľa zobrazí prihlasovacie okno. Po zadaní správnych prihlasovacích údajov, je užívateľ presmerovaný na stránku `/home`.

Ak sme prihlásený, môžeme vytvárať Spark úlohu. Z kódu v časti je zrejmé, že úloha pracuje so súborom uloženým na lokálnom HDFS. Pre nahranie súboru na lokálny HDFS využijeme webové rozhranie, ktoré poskytuje Hadoop a ktoré je dostupné na adrese `localhost:50070`.

Podľa kódu z ukážky 9.1, ako prvé využijeme SparkContext pre načítanie súboru z HDFS. Zo zoznamu `Spark Component`, si pod položkou SparkContext, vyberieme metódu `Input text`. Metóda sa zobrazí v grafe ako nový uzol s rovnakým názvom, aký je v zozname komponent. Postupne pridáme všetky ďalšie komponenty, ktoré sú v tejto úlohe potrebné.

Následne vytvoríme hrany, ktoré reprezentujú smer vykonávania úlohy. Komponentu `Input text` spojíme s komponentou `flatMap`, tak že najprv kliknutím označíme Input text,

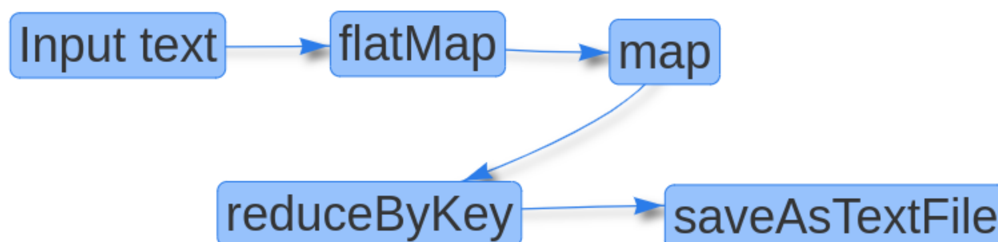
[1] https://spark.apache.org/examples.html

potom klikneme na flatMap, čím sa nám vytvorí hrana. Podobným spôsobom pridáme ďalšie hrany.

Pri dvojkliku na uzol sa zobrazí okno s nastavením povinných a tzv. keywords argumentov metódam, ktoré nesú dané komponenty. Po otvorení nastavení komponenty Input text, do vstupného poľa pre prvý povinný argument zadáme textový reťazec vo formáte 'hdfs://localhost:54310/words.txt'. Nastavovanie ďalších argumentov nie je v túto chvíľu potrebné.

Takto postupne nastavíme všetkým komponentám povinné argumenty. Po nastavení tlačidlom `Check PySpark Code`, skontrolujeme správnosť vytvorenej úlohy. Ak sme niekde zabudli vyplniť povinný argument, daná komponenta zmení farbu na červenú.

Vytvorenú Spark aplikáciu reprezentuje graf na obrázku 9.1.



Obr. 9.1: Spark úloha vytvorená zo Spark komponent

Ak máme úlohu vytvorenú, môžeme ju spustiť. Kliknutím na tlačidlo `Run` aplikáciu spustíme. Ak nemáme vytvorené spojenie (reláciu) so serverom Livy, budeme nato upozornení. Po vytvorení relácie, tlačidlo `Connect to Livy`, a prechodu reláciu do stavu `idle`, čo znamená že relácia je pripravená prijímať vstupy (waiting for inputs), spustíme úlohu tlačidlom Run.

Po odoslaní sa v spodnej časti obrazovky zobrazí `Progress bar`, ktorý indikuje, v akom stave sa nachádza spracovanie aktuálnej úlohy. Ak by užívateľ znova zaslal úlohu, progress bar bude zobrazovať vždy poslednú spustenú úlohu.

Keď je úloha ukončená, užívateľ si môže pozrieť výsledok cez webové rozhranie na HDFS. V tomto prípade pôjde o súbor `wordCount.txt`. Ďalej na stránke `/status`, pod reláciou, v ktorej spustil úlohu, si vie užívateľ stiahnuť výstup z danej úlohy. Jedná sa však o výstup zo `STDOUT`, v tomto prípade sme na tento výstup nič nezapisovali, tak ten bude prázdny[2].

Ďalej po kliknutí na tlačidlo `More info`, je užívateľ presmerovaný na webové rozhranie Sparku, kde si vie detailnejšie pozrieť jednotlivé časti spracovania. Výsledok úlohy je teda dostupný na HDFS, v súbore `wordCount.txt`.

9.1.2 Uloženie práce

Ak by si užívateľ chcel prácu odložiť a vrátiť sa k nej neskôr, má k dispozícii tlačidlo `Save as`, ktoré uloží aktuálnu prácu na server. Po vložení názvu práce, sa zobrazí hlásenie o úspešnom uložení a v zoznam `File name` sa zobrazí práve vytvorený súbor. Súbory sa ukladajú s príponou `.json`.

[2]Výstup je vo formáte JSON, STDOUT je v časti data - text/plain

9.1.3 Vytvorenie vlastnej komponenty

Po odhlásení súčasného používateľa, vytvoríme účet ďalšiemu používateľovi (random). Po prihlásení nového používateľa, vidíme že zoznam jeho súborov je prázdny.

Znova prihlásime predchádzajúceho používateľa (test). Rozhodli sme sa, že z úlohy, ktoré sme vytvorili v časti 9.1.1, vytvoríme vlastnú komponentu, ktorú budeme zdieľať s ďalším používateľom.

Zo zoznamu súborov vyberieme uloženú úlohu, tlačidlom `Load` ju nahráme na pracovnú plochu. Teraz podržaním tlačidla CTRL, označíme všetky uzly grafu. Potom klikneme na tlačidlo `Make component`, zobrazí sa nám modálne okno. Ako prvé vyplníme jedinečný názov komponenty, krátky popis. Užívateľ tejto novej komponente bude explicitne nastavovať argumenty pre vstupný a výstupný súbor, ostatné budú pri nastavovaní komponenty skryté. To dosiahneme tak, že po kliknutí na pole s názvom Spark komponenty, zaškrtneme jej checkbox v stĺpci `Expose`. To urobíme pre komponentu `Input text` a `SaveAsTextFile`.

Po kliknutí na tlačidlo `Create component` sa z pracovnej plochy odstránia všetky uzly, ktoré tvoria novú komponentu a nahradia sa novou, v tomto prípade budú všetky uzly nahradené jediným s názvom novej komponenty. Zároveň sa objaví nová položka v zozname komponent pod `User Define Components`.

Vytvorenú komponentu je možné spúšťať ako pôvodnú úlohu, ktorá bola vytvorená zo Spark komponent.

9.1.4 Zmazanie úlohy

Teraz keď máme vytvorenú vlastnú komponentu, môžeme pôvodne uloženú úlohu zmazať zo súborov. Po prejdení na stránku `/profile`, v časti `Manage Workspace`, vyberieme názov, pod ktorým bola uložená prechádzajúca úloha a tlačidlo `Delete` ju odstránime.

9.1.5 Zdieľanie komponenty

Teraz keď máme vytvorenú vlastnú komponentu, môžeme ju zdieľať ostatným používateľom. Zdieľanie nastavíme na stránke `/profile`, v časti `Manage Components`. Zo zoznamu komponent vyberieme tú, ktorú chcem zdieľať s ostatnými. Potom z ďalšom poli zadáme emailovú adresu používateľa, ktorému chceme zdieľať vybranú komponentu. Po kliknutí na tlačidlo `Add user`, je užívateľ schopný túto komponentu používať, taktiež sa zobrazí jeho emailová adresa v časti `This component also shared:`.

Po odhlásení užívateľa a prihlásení užívateľa (random), uvidí v časti `User Define Components`, komponentu ktorá mu bola zdieľaná a môžu ju ďalej bez problémov používať.

Kapitola 10

Záver

Cieľom tejto diplomovej práce bolo vytvoriť webovú aplikáciu, ktorá by umožnila pomocou grafického editora vytváranie a interaktívne spúšťanie Spark úloh. Úlohy by mali byť vytvárané z vopred definovaných komponent, ale zároveň by malo byť možné vytváranie takýchto komponent.

K vytvoreniu tejto práce bolo potrebné naštudovať ako funguje prostredie Apache Spark pre distribuované výpočty, aké sú možnosti vzdialeného zadávania úloh pomocou rozhrania REST v projekte Apache Livy. Boli preskúmané existujúce riešenia pre znázorňovanie úloh – projekt Zeppelin, projekt Jupyter, aké sú v týchto projektoch možnosti vytvárania úloh, vstupy a výstupu dát, užívateľské rozhranie a podobne.

Na základe tohoto prieskumu bolo navrhnuté grafické rozhranie pre interaktívne vytváranie Spark úlohy, reprezentovanej pomocou grafu. Doplnené boli časti užívateľského rozhrania aplikácie, aby užívateľovi spríjemnili a najmä urýchlili prácu.

K vytvoreniu takejto aplikácie bolo potrebné štúdium najnovších trendov a technológií vo vývoji webových aplikácií spolu s technológiami, ktoré sa používajú na strane servera v takýchto riešeniach.

Výsledkom je aplikácia, ktorá umožňuje vo webovom prehliadači vytvárať Spark úlohy z vopred definovaných komponent, tzv. Spark komponent, vytváranie nových komponent zo Spark komponent s obmedzením, ktoré bolo popísané v časti 8.1.3, a zdieľania vlastných komponent medzi ostatnými používateľmi, čo môže veľmi dobre slúžiť pre užívateľov, ktorí nie sú až tak technicky zdatní a nepoznajú prostredie Spark. Užívateľ môže svoju prácu ukladať na server a kedykoľvek sa k nej vrátiť. Výsledok práce je demonštrovaný v časti 9.

Do budúcnosti by bolo vhodné vytvoriť generátor, ktorý by automaticky z aktuálnej dokumentácie Sparku vygeneroval potrebnú dátovú štruktúru pre túto aplikáciu, aby si užívateľ každú Spark komponentu nemusel vytvárať ručne sám. Vhodné by bolo upraviť zobrazovanie komponent, aby hneď na prvý pohľad bolo zrejmé, koľko vstupov a výstupov komponenta má, čo by vyriešilo vyššie spomínané obmedzenia.

Literatúra

[1] How to use the Livy Spark REST Job Server API for submitting batch jar, Python and Streaming Jobs. [Online], navštíveno 7.1.2018.
URL http://gethue.com/how-to-use-the-livy-spark-rest-job-server-api-for-submitting-batch-jar-python-and-streaming-spark-jobs/

[2] Livy: A REST Web Service For Apache Spark. [Online], navštíveno 7.1.2018.
URL https://databricks.com/session/livy-a-rest-web-service-for-apache-spark

[3] Project Jupyter. [Online], navštíveno 8.1.2018.
URL http://jupyter.org/

[4] Spark GraphX. [Online; navštíveno 4.1.2018].
URL https://spark.apache.org/graphx/

[5] What is Apache Zeppelin ? [Online], navštíveno 8.1.2018.
URL http://zeppelin.apache.org/docs/0.7.3/index.html

[6] Booth, J. D.: Angular 2 Succinctly. [Online], E-book, navštíveno 8.5.2018.
URL https://www.syncfusion.com/ebooks/angular2_succinctly

[7] Brown, R.: Django vs Flask vs Pyramid: Choosing a Python Web Framework. [Online], navštíveno 21.4.2018.
URL https://www.airpair.com/python/posts/django-flask-pyramid

[8] Chitturi, P. P.: *Apache Spark for Data Science Cookbook*. 2016, ISBN 978-1-78588-010-0.

[9] Databricks: A Gentle Introduction to Apache Spark. E-book.

[10] Dutcher, J.: What Is Big Data? [Online], navštíveno 10.1.2018.
URL https://datascience.berkeley.edu/what-is-big-data/

[11] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce, UNIVERSITY OF CALIFORNIA, IRVINE, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[12] Fredrich, T.: A RESTful Tutorial. [Online], navštíveno 22.4.2018.
URL http://www.restapitutorial.com

[13] Girdhar, I.: Understanding Cross Origin Resource Sharing (CORS). [Online], navštíveno 8.5.2018.

URL https://www.securityninja.io/understanding-cross-origin-resource-sharing-cors/

[14] Holdren Karau, A. K.: *Learning Spark*. 2014, ISBN 1449358624.

[15] HORTONWORKS: APACHE ZEPPELIN. [Online], navštíveno 8.1.2018.
URL https://hortonworks.com/apache/zeppelin/

[16] Hui Wang, X. W.: A measurement study of device-to-device sharing in mobile social networks based on Spark. *CONCURRENCY AND COMPUTATION-PRACTICE AND EXPERIENCE*, 2017, ISSN 1532-0626.

[17] Jones, M.: JSON Web Token (JWT). [Online], navštíveno 15.5.2015.
URL https://tools.ietf.org/html/rfc7519

[18] Laskowski, J.: Mastering Apache Spark 2 (Spark 2.2+). [Online; navštíveno 4.1.2018].
URL https://jaceklaskowski.gitbooks.io/mastering-apache-spark

[19] Makai, M.: WSGI Servers. [Online], navštíveno 21.4.2018.
URL https://www.fullstackpython.com/wsgi-servers.html

[20] Matei Zaharia, M. C.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA: USENIX, 2012, ISBN 978-931971-92-8, s. 15–28.
URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[21] Mohd RehanGhazi, D. G.: Hadoop, MapReduce and HDFS: A Developers Perspective. *Procedia Computer Science*, ročník 48, 2015: s. 45–50, ISSN 1877-0509.

[22] Monnappa, A.: Data Science vs. Big Data vs. Data Analytics. [Online], navštíveno 10.1.2018.
URL https://www.simplilearn.com/data-science-vs-big-data-vs-data-analytics-article

[23] Palovská, H.: Databáze: relační nebo objektové? [Online], navštíveno 15.5.2015.
URL https://nb.vse.cz/~palovska/Db_obj_rel.pdf

[24] Paul Adamczyk, R. E. J.: REST and Web Services: In Theory and in Practice. [Online], navštíveno 20.4.2018.
URL https://www.researchgate.net/publication/265236489

[25] Ronacher, A.: Flask Documentation. [Online], navštíveno 21.4.2018.
URL https://media.readthedocs.org/pdf/flask/latest/flask.pdf

[26] Schultz, J.: How Much Data is Created on the Internet Each Day? [Online], navštíveno 10.1.2018.
URL https://blog.microfocus.com/how-much-data-is-created-on-the-internet-each-day/

[27] Shao, S.: LIVY: A REST INTERFACE FOR APACHE SPARK. [Online], navštíveno 7.1.2018.
URL
https://hortonworks.com/blog/livy-a-rest-interface-for-apache-spark/

[28] Shaporda, A.: What Is a Web Framework? [Online], navštíveno 20.4.2018.
URL https://djangostars.com/blog/what-is-a-web-framework/

Prílohy

Zoznam príloh

A Návrh hlavnej obrazovky **59**

B Dátové štruktúry webovej časti **60**

B.1 Spark komponenta 60

B.2 Užívateľom definovaná komponenta 61

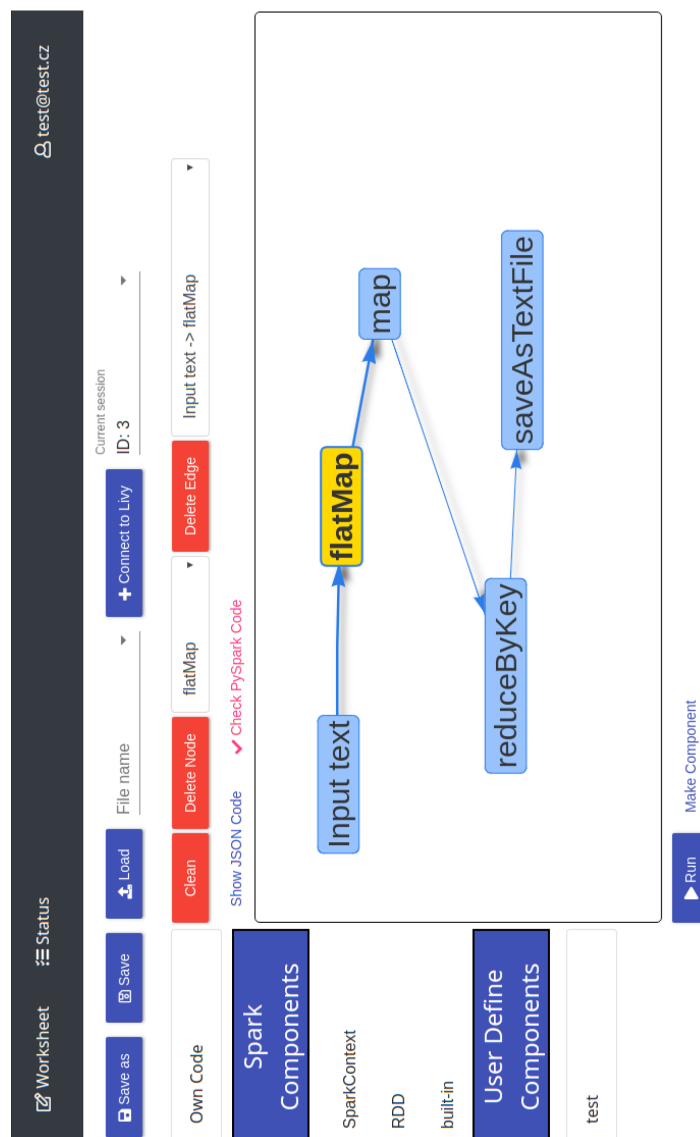
C Manuál **63**

D Obsah priloženého pamäťového média **66**

Príloha A

Návrh hlavnej obrazovky

Kompletná hlavná stránka aplikácie, dostupná na adrese `/home`.



Obr. A.1: Hlavná obrazovka aplikácie.

Príloha B

Dátové štruktúry webovej časti

B.1 Spark komponenta

Príklad Spark komponenty `Input text`:

```
{
   "label": "Input text",
    "applyOn": "sc",
    "returnType": "rdd",
    "applicableOn": "sc",
    "type": "spark_component",
    "class": "SparkContext",
    "code": {
        "pyspark3": {
            "rawCode": "textFile",
            "required_arguments": [
                {
                    "type": "str",
                    "position": 0
                }
            ],
            "kwargs": [
                {
                    "name": "minPartitions",
                    "default_value": "None"
                },
                {
                    "name": "use_unicode",
                    "default_value": "True"
                }
            ]
        }
    },
    "description": "Read a text file from HDFS, a local file system."
}
```

Výpis B.1: Výpis Spark komponenty Input text

B.2 Užívateľom definovaná komponenta

```
{
    "label": "myFirstComponent",
    "nodes": [
      {
        "applicableOn": "sc",
        "applyOn": "sc",
        "arguments": {
          "kwargs": [
            {
              "default_value": "False",
              "expose": true,
              "name": "preservesPartitioning",
              "value": "True"
            }
          ],
          "required_arguments": [
            {
              "expose": true,
              "position": 0,
              "type": "str",
              "value": ""
            }
          ]
        },
        "code": "textFile",
        "connectToNode": [
          3
        ],
        "label": "Input text",
        "nodeId": 1,
        "returnType": "rdd"
      },
      {
        "applicableOn": "rdd",
        "applyOn": "rdd",
        "arguments": {
          "kwargs": [
            {
              "default_value": "False",
              "expose": false,
              "name": "preservesPartitioning",
              "value": "False"
            }
          ],
          "required_arguments": [
            {
```

```
                "expose": true,
                "position": 0,
                "type": "function",
                "value": ""
              }
            ]
          },
          "code": "flatMap",
          "connectToNode": [],
          "label": "flatMap",
          "nodeId": 3,
          "returnType": "rdd",
          "type": "spark_component"
        }
    ],
    "description": "My first component.",
    "type": "user_component",
    "connectToNode": []
}
```

Výpis B.2: Výpis užívateľom definovanej komponenty

Príloha C

Manuál

Aplikácia bola testovaná na operačnom systéme Linux Mint 18.3. Tento manuál slúži ako návod pre spustenie aplikácie na operačných systémoch Linux.

Pre spustenie aplikácie a plného využitia, tzv. spúšťanie Spark úloh je potrebné mať nainštalované (v zátvorke je uvedená verzia, na ktorej bola aplikácia otestovaná a plne funkčná):

- Node.js (8.10.0) — https://nodejs.org/en/download/package-manager/#debian-and-ubuntu-based-linux-distributions
- Python (3.6.3) — https://tecadmin.net/install-python-3-6-ubuntu-linuxmint/
- MongoDB (3.6.1) — https://docs.mongodb.com/manual/installation/
- Apache Livy (0.4.0)— dostupné na priloženom DVD, alebo http://livy.incubator.apache.org/get-started/
- Apache Spark (2.2.1) — https://spark.apache.org/downloads.html, ďalšie prerekvizity:
  https://www.tutorialspoint.com/apache_spark/apache_spark_installation.htm
- Hadoop (2.6.0) — návod: http://www.bogotobogo.com/Hadoop/BigData
  _hadoop_Install_on_ubuntu_single_node_cluster.php,
  aby hadoop nevyžadoval heslo, do časti návodu, kde sa konfiguruje hdfs-site.xml doplnit:
  <property>
  <name>dfs.permissions</name>
  <value>false</value>
  </property>

Nastaviť v skripte livy-*/conf/livy-env.sh:

- HADOOP_CONF_DIR — ak ste šli podľa návodu vyššie, tak /usr/local/hadoop/etc/hadoop, ak nie tak podľa vlastnej inštalácie
- SPARK_HOME — ak ste šli podľa návodu vyššie, tak /usr/local/spark, ak nie podľa vlastnej inštalácie

Nastaviť v konfiguračnom súbore livy-*/conf/livy.conf:

- livy.spark.master = yarn
- livy.spark.deploy-mode = cluster

Ak bude použitá verzia Livy, ktorá je dostupná na priloženom DVD, táto konfigurácia už je v súboroch nastavená.

Serverová časť

Pre serverovú časť je vhodné nainštalovať `virtualenv`, ktorý vytvorí pre aplikáciu vlastné prostredie. Vhodné je použiť toto rozšírenie:
https://virtualenvwrapper.readthedocs.io/en/latest/.
Potom v adresári sources/master_dip_backend spustiť `pip install requirements.txt`.
Po inštalácií spustiť MongoDB (príkaz `sudo service mongod start`).
Potom cez `mongo-shell` vytvoriť databázu s názvom `ProjectDIP` bez mena a hesla na prednastavenom porte 27017.

Súčasťou priloženého disku v časti sources/utils/mongoAdmin, je utilita, ktorá ponúka administráciu databázy cez webové rozhranie. Dostupná tiež na:
https://github.com/mrvautin/adminMongo.

Po splnení vyššie uvedených krokov, sa flask aplikácia spustí takto:

- v adresári sources/master_dip_backend
- EXPORT_FLASK=run.py
- flask run

Po spustení aplikácie sa spustia inicializačné skripty, ktoré nahrajú do databázy Spark komponenty, vytvoria používateľa `test@test.cz, heslo: testtest` a pridajú mu do súborov ukážky, ktoré si potom môže vo webovej aplikácií vyskúšať. Jedná sa o ukážky:

- pi_estimation.json — odhad čísla PI,
zdroj: https://spark.apache.org/examples.html
- pagerank.json — algoritmus PageRank, zdroj: https://github.com/apache/spark/blob/master/examples/src/main/python/pagerank.py
- cartesian_example.json — príklad s funkciou cartesian,
zdroj: https://spark.apache.org/docs/latest/api/python/pyspark.html, časť sampleByKey
- wordCount.json — zistí počet slov vo vstupnom súbore,
zdroj: https://spark.apache.org/examples.html

Ak sa sa užívateľ rozhodne pridať novú Spark komponentu, vytvorí ju podľa vzoru B.1, výklad jednotlivých položiek je 6.2.6.

Vloží ju do súboru master_dip/init_data/spark_components.json, cez utilitu monogoAdmin zmaže celú kolekciu SparkComponent a znova spustí aplikáciu.

Webová časť

K spustenie webovej časti aplikácie je potrebné mať nainštalovaný už spomínaný Node.js a balík manažér npm (5.6.0) - https://www.npmjs.com/get-npm. Potom stačí v adresári sources/master_dip spustiť `npm install`. Angular aplikácia sa spustí príkazom `ng serve`. Prednastavená URI je http://localhost:4200.

Príloha D

Obsah priloženého pamäťového média

Zložky so zdrojovými kódmi aplikácie a ďalšie podporné súbory

- adresár sources/master_dip_backend — serverová časť aplikácie
- adresár sources/master_dip — webová časť aplikácie
- adresár sources/utils — webová administrácia MongoDB
- adresár sources/livy — Apache Livy

Zložka so súbormi tejto práce

- adresár tex

Zložka s demonstračnými materiálmi

- adresár demonstration — obsahuje video ukážku, ktorú textovo popisuje kapitola 9