Univerzita Komenského, Bratislava

Fakulta Matematiky, Fyziky a Informatiky

Návrh a implementácia klientských pohľadov nad NoSQL databázami

Diplomová práca

2015 **Bc. Igor Liška**

Univerzita Komenského, Bratislava

Fakulta Matematiky, Fyziky a Informatiky

Návrh a implementácia klientských pohľadov nad NoSQL databázami

Diplomová práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra Informatiky
Školiteľ: RNDr. Tomáš Kulich, PhD.

Bratislava, 2015 **Bc. Igor Liška**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Igor Liška
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Návrh a implementácia klientských pohľadov nad NoSQL databázou
Design and implementation of client-side views of NoSQL database

Cieľ: Navrhnúť architektúru mechanizmu, pomocou ktorého vzdialený klient reflektuje pohľad nad NoSQL dátami servra. Zamerať sa na aspekty synchronizácie týchto pohľadov medzi serverom a veľkým počtom klientov s dôrazom na optimalizáciu dátového toku. Spraviť vzorovú implementáciu tohto mechanizmu.

Vedúci: RNDr. Tomáš Kulich, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 23.10.2013

Dátum schválenia: 27.11.2013

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....
študent vedúci práce

Ďakujem vedúcemu svojej diplomovej práce, RNDr. Tomášovi Kulichovi, PhD., za jeho podporu a cenné rady, ktoré nemalou mierou prispeli k dotiahnutiu tejto práce do konca.

Abstrakt

Autor:	Bc. Igor Liška
Názov bakalárskej práce:	Návrh a implementácia klientských pohľadov nad NoSQL databázami
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta Matematiky, Fyziky a Informatiky
Katedra:	Katedra Informatiky
Vedúci bakalárskej práce:	RNDr. Tomáš Kulich, PhD.

Bratislava, 2015

Tvorba webových aplikácií, ktoré kladú dôraz na plynulosť práce užívateľa a vyvolávajú dojem klasických desktopových aplikácií, je v súčasnosti náročná. Väčšina webových aplikácií trpí príliš vysokou latenciou užívateľského rozhrania, nakoľko akcie užívateľa musia čakať na odpoveď zo servera. Naskytá sa tu priestor pre systém, ktorý zastreší synchronizáciu dát medzi klientami a serverom a umožní klientom pracovať s dátami, ktoré sú dostupné lokálne pre rýchlu a plynulú prácu užívateľov a jednoduchý vývoj aplikácie. Riešenie, ktoré sme navrhli a implementovali v tejto práci prináša synchronizačný protokol, ktorý je integrovaný do open-source knižnice sync, ktorá je tiež produktom tejto práce. Myšlienka riešenia je založená na udržiavaní histórie zmien dát a efektívnym počítaním rozdielu medzi dátami klienta a servera. Finálne riešenie tejto práce je funkčná knižnica, ktorá slúži ako základný stavebný kameň pri tvorbe interaktívnych webových aplikácií.

Kľúčové slová: single page aplikácie, synchronizácia, NoSQL, MongoDB, publish, subscribe

Abstract

Author: Bc. Igor Liška
Title: Design and implementation of client-side views of NoSQL database
University: Comenius University in Bratislava
Faculty: Faculty of Mathematics, Physics and Informatics
Department: Department of Computer Science
Advisor: RNDr. Tomáš Kulich, PhD.
Bratislava, 2015

The development of web applications that are optimized for fluid user experience and that resemble standard desktop applications is challenging nowadays. Most web applications suffer from a poor latency of the user interface as all user actions have to wait for the roundtrip to the server. The issue can be seen as an opportunity to create a system which takes care of the synchronization of the data between the client and the server. The clients can access the data locally for improved user experience. The solution we designed and implemented as a part of this master thesis introduces a synchronization protocol, which is integrated into the open-source library called sync, which is also product of this thesis. The idea behind the solution is based on maintaining a history of changes of the data and efficient computation of difference between the client-side and the server-side view of the data. The sync library can be used as a base for development of interactive web applications.

Keywords: single page aplications, synchronization, MongoDB, NoSQL, publish, subscribe

Obsah

Úvod **1**

1 Definícia pojmov a prehľad technológií **2**

1.1 Definícia pojmov 2

1.1.1 NoSQL databáza 2

1.1.2 MongoDB 3

1.1.3 Klientský pohľad 4

1.1.4 Publish - subscribe model 5

1.2 Cieľ práce 6

2 Single page aplikácie **8**

2.1 Definícia single page aplikácie 8

2.2 Synchronizácia dát v single page aplikáciách 10

2.2.1 Latencia užívateľského rozhrania 10

2.2.2 Aktuálnosť dát 11

3 Existujúce riešenia **12**

3.1 Meteor 12

3.1.1 Reaktívne programovanie 14

3.1.2 Synchronizácia dát 16

3.2 Datomic 22

3.2.1 Architektúra 23

3.2.2 Reprezentácia dát 25

3.2.3 Dotazovací jazyk 26

3.2.4 Synchronizácia dát 29
3.3 Firebase 32

4 Synchronizačný protokol 34
4.1 Správanie protokolu 34
4.1.1 Inicializácia protokolu 35
4.1.2 Beh protokolu 36
4.2 História kolekcie 36
4.3 Synchronizácia filtrovaného pohľadu 38
4.4 Synchronizácia úplného pohľadu 41
4.5 Efektívnosť synchronizačných algoritmov 45

5 Implementácia 47
5.1 Knižnica sync 47
5.2 Dart 47
5.3 Štruktúra knižnice sync 48
5.4 Server 49
5.5 Klient 50

Záver 51

Úvod

Cieľom tejto diplomovej práce je navrhnúť a implementovať synchronizačný mechanizmus, ktorý umožní reflektovať pohľady nad dátami servra na veľký počet klientov s dôrazom na optimalizáciu dátového toku medzi klientom a serverom. Riešenie uvedené v tejto práci pracuje s dátami uloženými v NoSQL databáze. Referenčná implementácia, ktorá je výstupom tejto práce sa opiera o NoSQL databázu MongoDB.

V úvodnej kapitole tejto práce si prejdeme niekoľkými definíciami pojmov, aby sme uviedli čitateľa do terminológie. Následne sa pozrieme na oblasť single page aplikácií a zdôvodníme, prečo má zmysel zaoberať sa optimalizovanou synchronizáciou dát v tejto oblasti. Pozrieme sa aj na alternatívne prístupy k tomuto problému v podaní frameworku Meteor a databázových riešení Datomic a Firebase. Po analýze uvedených alternatív ukážeme výhody a nevýhody ich prístupu k danej problematike a porovnáme ich s naším riešením.

Po analýze alternatívnych riešení predstavíme náš synchronizačný protokol na teoretickej a algoritmickej rovine. Ukážeme, akým spôsobom nám udržiavanie histórie dát pomôže pri efektívnom výpočte rozdielu medzi serverovým pohľadom a klientským pohľadom na dáta. Predvedieme algoritmy, ktoré nám umožnia rozdiel medzi pohľadmi vypočítať a analyticky zhodnotíme ich vlastnosti. Zdôvodníme, prečo ich považujeme za efektívne. Prácu uzavrieme pohľadom na referenčnú implementáciu, ktorá je tiež výstupom tejto práce v podaní open-source knižnice sync.

Kapitola 1

Definícia pojmov a prehľad technológií

V prvej kapitole priblížime prostredie, v ktorom budeme pracovať a spomenieme tiež technológie tretích strán, ktoré nám umožnia docieliť požadovaný výsledok rýchlejšie a efektívnejšie.

1.1 Definícia pojmov

V prvom rade je dôležité uviesť a definovať pojmy, s ktorými sa budeme stretávať v každej kapitole tejto práce. Spomenieme termíny ako NoSQL databáza, dátová kolekcia, dokument a klientský pohľad.

1.1.1 NoSQL databáza

Ako je uvedené v [Chr], v súčasnosti sú relačné databázové systémy dominantnou technológiou na trhu webových a biznisových aplikácií. Takéto databázové systémy sa opierajú o pravidlá relačnej algebry a dotazovanie na dáta umožňujú prostredníctvom jazyka SQL. Podľa [Chr], termín NoSQL bol prvýkrát použitý v roku 1998 pre relačnú databázu, ktorá nevyužívala jazyk SQL. Termín opäť ožil v roku 2009, kedy bol spomínaný na konferenciách zástancov nerelačných databáz. NoSQL databázové systémy nevylučujú

používanie jazyka SQL, avšak prinášajú nové myslenie a riešenia do databázových systémov a preto pod skratkou NoSQL rozumieme skôr Not-only-SQL (nie len SQL), ako No-SQL (nie SQL). V súčasnom svete existujú desiatky rôznych NoSQL databázových systémov, medzi najpoužívanejšie patrí napríklad MongoDB, Redis a Memcached, ale nemenej známe sú aj ElasticSearch, alebo Apache Cassandra. Naším reprezentantom NoSQL databáz bude práve MongoDB, ktoré poslúži ako stavebný pilier pre riešenie navrhnuté v tejto práci.

1.1.2 MongoDB

MongoDB je open-source databázové riešenie, spadajúce do kategórie noSQL, absentujúce od relačného prístupu k databázam (ako napríklad databázy postavené nad jazykom SQL). Ako môžeme ďalej nájsť v [Mon], výhodou MongoDB je jednoduchosť návrhu a horizontálneho škálovania.

Kolekcia, dokument

Základnou entitou v MongoDB je dokument, ktorý si môžeme predstaviť ako slovníkovú dátovú štruktúru na bázi kľúč - hodnota, pričom hodnota môže byť další slovník, pole, alebo niekoľko základných typov ako číselná hodnota, poprípade reťazec znakov. Dokumenty sú zoskupené v kolekciách. Je dôležité poznamenať, že kolekcie nemajú predpísanú schému, respektíve dokumenty v jednej kolekcii môžu mať rozdielne štruktúry. Niekoľko kolekcií dokopy tvorí databázu. Dokument je analógiou k riadku v SQL databáze, kolekcia zase analogicky predstavuje tabuľku v SQL databáze.

Dotazovací jazyk v MongoDB podporuje vyhľadávanie podľa hodnoty kľúča, rozsahu hodnôt, alebo pomocou regulárnych výrazov a vlastných JavaScript funkcií. Indexácia je plne podporovaná (podobne ako v klasických relačných databázach), sú podporované tiež zložené indexy. Agregácia fun-

guje pomocou techniky map-reduce.

1.1.3 Klientský pohľad

Ako názov práce napovedá, cieľom práce je synchronizovať klientské pohľady (clientside views) so serverom (databázou). V predošlej podsekcii sme uviedli kolekciu ako množinu dokumentov. Neformálne, pohľad na kolekciu je taká podmnožina dokumentov kolekcie, ktorá je:

- filtrovaná nejakou množinou atribútov
- usporiadaná podľa nejakej množiny atribútov
- limitovaná na počet dokumentov

Filtrovanie, usporiadanie a limitovanie nazývame parametrami pohľadu a pohľad, ktorý je definovaný všetkými tromi parametrami, nazývame úplným pohľadom. Pre lepšie ilustrovanie rozdielu medzi kolekciou a pohľadom uvedieme dva príklady. Na ukážke 1.1 môžeme vidieť kolekciu *osoby*, s atribútmi *id*, *name* a *age*.

```
[
  #1,  name: Adam,    age: 22
  #2,  name: Daniel,  age: 35
  #3,  name: Martin,  age: 45
]
```

Obr. 1.1: Kolekcia *osoby*

Na ukážke 1.2 je uvedený pohľad na kolekciu z ukážky 1.1, avšak tento pohľad je filtrovaný podľa atribútu *age*, ktorý musí byť vyšší ako hodnota 24, je usporiadaný podľa atribútu *age* zostupne a počet dokumentov v pohľade je

limitovaný na dva dokumenty. V ľudskej reči je to pohľad na dvoch najstarších ľudí z kolekcie *osoby*, ktorí su starší ako 24 rokov, usporiadaní podľa veku zostupne.

```
[
  #3,  name: Martin,  age: 45
  #2,  name: Daniel,  age: 35
]
```

Obr. 1.2: Kolekcia *osoby*

Nakoniec ešte uvedieme, že ak sa takýto pohľad na kolekciu fyzicky nachádza na strane klienta, hovoríme o klientskom pohľade.

1.1.4 Publish - subscribe model

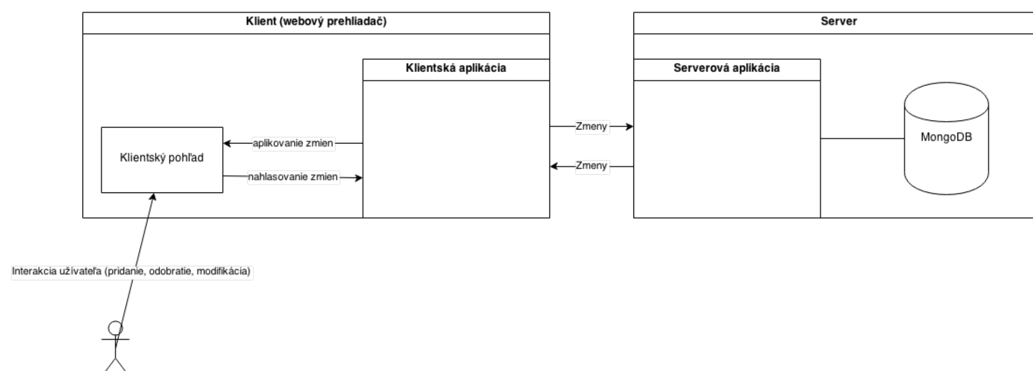
V tejto sekcii sa budeme venovať modelu publish - subscribe, ktorý upresňuje spôsob, akým prebieha komunikácia medzi klientom a serverom. Tento model prináša dve nové role: publisher (vydavateľ) a subscriber (odoberateľ). V čisto teoretickom modeli správy prirodzene prúdia od vydavateľov k odoberateľom, avšak správy nemajú presne určeného adresáta (odoberateľa), ale sú klasifikované do tried a vydavatelia nemajú žiadne vedomosti o tom, či existujú nejakí odoberatelia pre nejakú triedu. Na druhej strane, odoberatelia prejavia záujem odoberať správy klasifikované nejakou triedou a nemajú žiadnu informáciou o tom, čí existuje nejaký vydavateľ, ktorý poskytuje správy pre danú triedu. V našom prípade prevezme úlohu vydavateľa serverová aplikácia, ktorá poskytne odoberateľom - klientom rôzne pohľady na kolekcie, ktoré má k dispozícii v databázovom úložisku. Jednotlivé pohľady si môžeme predstaviť ako vyššie uvedené triedy správ. V súčasnej verzii riešenia je model publish - subscribe len simulovaný, nakoľko súčasné riešenie je postavené na HTTP protokole, ktorý umožňuje iniciovanie spojenia len zo

strany klienta, dôsledkom čoho vydavateľ (server) nemôže svojvoľne posielať správy odoberateľom, ale je nútený čakať, kým si každý klient periodicky vypýta ďalšie správy. Plánom do budúcnosti je implementácia komunikácie prostredníctvom websocketov, ktoré umožňujú obojsmernú komunikáciu a umožnia využívať jednu z hlavných výhod publish - subscribe modelu a to priamočiaru horizontálnu škálovateľnosť počtu vydavateľov a teda možný nárast výkonu.

1.2 Cieľ práce

Cieľom tejto práce je implementovať synchronizačný protokol, preto je v prvom rade dôležité uviesť strany, medzi ktorými bude synchronizácia prebiehať. Ako sme už spomenuli v predošlých sekciách, naše prostredie je postavené na klient - server komunikačnom modeli. Server pozostáva z aplikačnej časti a databázového úložiska. Serverová aplikácia je zodpovedná za komunikáciu s klientami, synchronizáciu klientov a má úplný prístup k databázovému úložisku. Na strane klienta beží aplikácia priamo vo webovom prehliadači. Užívateľ prostredníctvom grafického rozhrania pracuje s klientským pohľadom (pridáva nové dokumenty, odoberá a modifikuje existujúce dokumenty). Klientská aplikácia zodpovedá za propagovanie zmien nad klientským pohľadom na server a za komunikovanie a synchronizovanie zmien zo servera. Celú schému môzeme detailne vidieť v ukážke 1.3.

Cieľom práce je vytvoriť riešenie, ktoré bude schopné obslúžiť veľký počet klientov súčasne. Pre zabezpečenie tohto parametru je kladený dôraz jednak na efektívnosť synchronizačného algoritmu, ako aj na horizontálnu škálovateľnosť serverovej aplikačnej časti. Pod horizontálnou škálovateľnosťou rozumieme paralelne bežiace serverové aplikácie, ktoré sú schopné pracovať s jedným databázovým úložiskom. Databázové úložisko MongoDB, ktoré v tejto práci využijeme, je produktom tretej strany, preto jeho škálovateľnosť v



Obr. 1.3: Schéma komunikácie

tejto práci riešime len z užívateľského hľadiska. Dôraz kladieme aj na veľkosť dátového toku na jednotlivých klientov. Cieľom je navrhnúť synchronizáciu takým spôsobom, aby server na klienta posielal len najnutnejšie informácie, z ktorých je klient schopný aktualizovať svoje dáta.

Kapitola 2

Single page aplikácie

Predtým, ako sa ponoríme do jadra tejto práce si najskôr uvedieme dôvody jej vzniku. V tejto kapitole vysvetlíme koncept single page aplikácií a vysvetlíme, ako single page aplikácie súvisia s touto prácou.

2.1 Definícia single page aplikácie

Single page aplikácia, ako nájdeme v [Sin], je webová stránka, ktorej cieľom je zabezpečiť plynulú prácu užívateľa a vyvolať v užívateľovi dojem, že pracuje s klasickou desktopovou aplikáciou. Všetky zdroje single page aplikácie (HTML kód spolu s ostatnými zdrojmi ako obrázky, CSS štýly a skripty) sa stiahnu pri úvodnom načítaní webovej stránky alebo sa dynamicky sťahujú dodatočne podľa potreby ako reakcia na činnosť užívateľa. Hlavnou vlastnosťou single page aplikácie je, podľa [Sin], že sa stránka nikdy fyzicky v prehliadači neobnoví (pod obnovením myslíme opätovné načítanie v prehliadači) a ani sa nikdy neodovzdá riadenie inej webovej stránke prostredníctvom odkazu alebo iným spôsobom. Na základe tejto vlastnosti vznikol názov single page aplikácia, nakoľko sa stretávame s jednou webovou stránkou, ktorá je jediným styčným bodom s užívateľom a zároveň poskytnutým spôsobom práce vyvoláva dojem aplikácie a nie webovej stránky.

Skutočná sila single page aplikácií (ďalej SPA) spočíva v spojení sveta

webových aplikácii so svetom desktopových aplikácií. Single page aplikácie prinášajú výhody z obidvoch uvedených svetov:

- **multiplatformovosť** - SPA fungujú na každej platforme, pre ktorú existuje webový prehliadač
- **dostupnosť** - SPA synchronizuje dáta so serverom, takže máme rovnaké dáta dostupné na všetkých použitých zariadeniach. V ideálnom prípade SPA dokážu bežať bez ohľadu na prítomnosť internetového pripojenia.
- **plynulosť užívateľského rozhrania** - riadenie patrí jedinej webovej stránke, dáta sa synchronizujú na pozadí.

Pre úplný a objektívny pohľad na SPA uvedieme aj niektoré známe nevýhody, ktoré so sebou SPA prinášajú:

- **delenie kódu medzi klienta a server** - existujú prípady, kedy je nutné duplicitne implementovať totožnú logiku na obidvoch stranách. Typickým príkladom sú validátory užívateľských vstupov, ktoré je z hľadiska odozvy užívateľského rozhrania vhodné implementovať na strane klienta, avšak nesmú chýbať ani na strane servera. V súčasnosti je tento problém riešiteľný vhodným výberom jazyka, v ktorom je SPA implementovaná. JavaScript alebo Dart sú vhodné na implementáciu obidvoch strán - klienta, aj servera.
- **história prehliadača** - nakoľko SPA, podľa definície, pozostávajú z jedinej webovej stránky, tak priamo rozbíjajú mechanizmus histórie prehliadača a jej navigáciu pomocou tlačidiel *Späť* a *Ďalej*. Tento problém je riešiteľný ukladaním stavu SPA do URL adresy a rekonštrukciou stavu na základe URL adresy. Detailnejší pohľad na túto problematiku

môžeme nájsť v [Sin]. Aj napriek riešiteľnosti tohoto problému považujeme jeho dopad za negatívny, nakoľko zodpovednosť prehliadača za históriu sa presúva do réžie samotnej SPA.

- **Rýchlosť úvodného načítania stránky** - úvodné načítanie SPA, ako sme uviedli v definícii, sťahuje všetky, respektíve väčšinu zdrojov (HTML kód, framework na ktorom ja SPA postavená, CSS štýly, obrázky, šablóny). Dôsledkom tohoto dizajnu existuje možnosť, že úvodné načítanie SPA môže trvať dlhšie ako pri tradičných webových stránkach, avšak tento problém považujeme za kompromis, ktorým následovne získame plynulejšiu prácu.

V ďalšej časti ukážeme, ako single page aplikácie súvisia s cieľom tejto práce.

2.2 Synchronizácia dát v single page aplikáciách

Ako sme uviedli v definícii single page aplikácie (ďalej SPA), jedným z cieľov SPA je orientácia na plynulosť práce, čo si technicky môžeme predstaviť ako minimálnu odozvu užívateľského rozhrania.

2.2.1 Latencia užívateľského rozhrania

Na odozve užívateľského rozhrania sa najviac podpisuje dynamické načítavanie obsahu, nakoľko najskôr musí klient požiadať server o nové dáta, poprípade vykreslený komponent a následne server musí spracovať požiadavku, vyžiadať dáta z databázy, naformátovať odpoveď a odoslať ju naspäť klientovi, pričom celá akcia môže trvať v ráde niekoľkých stoviek milisekúnd v lepšom prípade, niekoľko sekúnd v horšom.

Pri dokonalej webovej aplikácii by sa mala odozva užívateľského rozhrania blížiť k nule, respektíve by mala byť dostatočne malá na to, aby ju ľudské vnímanie nebolo schopné spozorovať. Pre zaujímavosť, [Car91] hovorí, že zhruba 100 milisekúnd je limit pre dosiahnutie pocitu okamžitej reakcie systému.

2.2.2 Aktuálnosť dát

Okrem minimálnej latencie užívateľského rozhrania chceme v SPA zabezpečiť aj aktuálnosť dát. V praxi si to môžeme predstaviť napríklad na aplikácii Google Docs, ktorá predstavuje online alternatívu ku štandardným kancelárskym textovým procesorom, ako napríklad Microsoft Word. Pod aktuálnosťou dát v aplikácii Google Docs označujeme zobrazovanie zmien v dokumente od všetkých kolaborantov v reálnom čase. Analogicky si môžeme predstaviť aktuálnosť dát pre ľubovoľnú SPA, v závislosti od jej funkcionality. Formálnejšiu definíciu aktuálnosti dát sformulujeme v nasledujúcej kapitole.

Latenciu užívateľského rozhrania a aktuálnosť dát sú ovplyvnené najmä spôsobom, akým prebieha synchronizácia dát medzi klientskou časťou SPA a serverom. Cieľom tejto práce je navrhnúť vhodný mechanizmus, respektíve synchronizačný protokol, ktorý umožní udržiavať aktuálne dáta na klientoch s dôrazom na objem dátového toku a zároveň nebude prekážkou pre odozvu užívateľského rozhrania.

Predtým, ako predstavíme návrh tohoto synchronizačného protokolu, uvedieme v nasledujúcej kapitole niekoľko existujúcich riešení a techník, ktoré majú rôzne výhody a nevýhody, poprípade kladú obmedzujúce podmienky pri ich použití.

Kapitola 3

Existujúce riešenia

Druhá kapitola tejto práce detailne rozoberie existujúce prístupy k problematike synchronizácie dát. Najskôr sa pozrieme na framework Meteor a následne predstavíme databázu Datomic. Trojicu nakoniec uzavrieme databázovou službou Firebase.

3.1 Meteor

Projekt Meteor je open-source framework pre vývoj webových aplikácií, ktorý vznikol koncom roku 2011 pod názvom Skybreak. Vývojovým jazykom tohoto frameworku je JavaScript, a to nielen na strane klienta, ale aj na strane servera. Serverová časť sa opiera o exekučné prostredie Node.js, ktoré pomocou virtuálneho stroja V8 od Googlu dokáže exekuovať programy napísané v JavaScripte na rôznych platformách ako Windows, OS X, operačné systémy z rodiny Linux a FreeBSD, ako nájdeme v [Nod]. Meteor je produktom skupiny Meteor Development Group. V podobe startupu bol inkubovaný v Y Combinatore a dostal investíciu vo výške 11,2 milióna dolárov. Cieľom projektu v budúcnosti, ako uvádza [Met], je vytvoriť komerčné hostingové riešenie pre aplikácie vytvorené prostredníctvom frameworku Meteor.

Pozrime sa na projekt Meteor z technického hľadiska. Do sveta webových aplikácií prináša progresívne a inovatívne myslenie, nakoľko sa vymyká zo

zabehnutých koľají drvivej väčšiny tradičných frameworkov pre vývoj webových aplikácií ako napríklad Ruby on Rails (jazyk Ruby), Zend a Symphony (jazyk PHP), alebo Django a Flask (jazyk Python). Uvedené frameworky sa síce líšia v drobných detailoch, avšak všetky sa silne opierajú o návrhový vzor MVC (Model - View - Controller), teda rozdelenie webovej aplikácie na tri základné komponenty, ktoré sú navzájom prepojené a každý zodpovedá za jednu oblasť webovej aplikácie.

MVC návrhový vzor

- **Model** zodpovedá za správanie aplikácie na základe jej domény, implementuje biznisovú logiku, spravuje dáta aplikácie a poskytuje informácie pre *view*.
- **View** má na starosti prezentáciu dát užívateľovi, ktoré poskytuje *model*. V praxi za *view* pokladáme napríklad šablóny, ktoré po doplnení dát z modelu vyprodukujú HTML kód, ktorý je prezentovaný užívateľovi.
- **Controller** uzatvára trojicu a zohráva úlohu mostu medzi *modelom* a *view*. *Controller* riadi *model* na základe vstupov od užívateľa a výstup *modelu* posiela do *view*.

Návrhový vzor MVC je dodnes neoddeliteľnou súčasťou vývoja webových aplikácií a usudzujeme, že v najbližších rokoch si svoje dominantné postavenie udrží. Meteor upustil od klasického MVC prístupu k stavbe webových aplikácií a namiesto toho umožňuje tvoriť webové aplikácie na základe myšlienok z reaktívneho programovania. Meteor zároveň mení prístup k práci s databázou a prenáša ju na stranu klienta, čo v prípade webových aplikácií postavených na uvedených MVC frameworkoch neprichádza do úvahy, nakoľko databáza spadá pod *model*, ktorý zase spadá do domény servera.

3.1.1 Reaktívne programovanie

Reaktívne programovanie je programovacia paradigma zameraná na tok dát a následnú propagáciu zmeny. Pre jednoduché porovnanie s klasickým imperatívnym programovaním si uveďme nasledujúci príklad. Majme výraz $a = b + c$.

- *imperatívne programovanie* - pri vyhodnotení výrazu by premenná a okamžite nadobudla výsledok $b + c$ a ak by sa v budúcnosti hodnoty premenných b, alebo c zmenili, nemalo by to žiadny vplyv na hodnotu premennej a, respektíve by sme premennú a museli opätovne aktualizovať.

- *reaktívne programovanie* - hodnota premennej a by sa automaticky aktualizovala vždy, keď by sa zmenila hodnota premennej b, alebo c. Za aktualizovanie hodnoty nie je zodpovedný programátor, ale prostredie, v ktorom je daný program spustený, respektíve automatická propagácia zmeny je priamo podporovaná programovacím jazykom.

Ako uvádza [Ing], reaktívne programovanie adresuje problémy, ktoré prináša zaužívaný návrhový vzor observer. Observer funguje na jednoduchom princípe, v ktorom objekt pri zmene svojho stavu pošle notifikácie všetkým objektom, ktoré sa registrovali ako pozorovatelia. Pod poslaním notifikácie si môžeme v praxi predstaviť volanie nejakej metódy pozorovateľa. Ako [Ing] ďalej spomína, návrhový vzor observer porušuje nasledujúce princípy softvérového inžinierstva:

- **Zapúzdrenie** - pre simulovanie činnosti nejakého stavového stroja je zvyčajne potrebných niekoľko observerov. [Ing] ilustruje tento problém na príklade kreslenie čiary pomocou myši - potrebujeme sledovať stav tlačidla a pohyb myši. Všetci pozorovatelia zvyčajne zdieľajú nejakú

spoločnú premennú, ktorá je však prístupná aj vrámci širšieho kontextu a môže byť zmenená iným, nezávislým kódom.

- **Modulárnosť** - skupina observerov, ktorá spolupracuje na jednej funkcii (napríklad kreslenie čiary z predošlého príkladu) tvorí voľné zoskupenie objektov, ktoré boli vytvorené na rôznych miestach v rôznych časoch. Preto je komplikované odstrániť funkciu kreslenia čiary ako celok.

- **Správa zdrojov** - životný cyklus observera je nutné spravovať explicitne. Ak chceme kresliť čiaru len v prípade, že je stlačené tlačidlo myši, musíme explicitne pridať a odobrať observera na pohyb myši.

- **Separácia úloh** - príklad na kreslenie čiary, ktorý uvádza [Ing], naraz detekuje pohyb myši a zároveň aj kreslí čiaru. V tomto prípade má jeden kód dve rôzne úlohy. V praxi sa ukazuje užitočnejšie separovať rôzne úlohy rôznym komponentom, príkladom je aj návrhový vzor MVC, ktorý sme už spomenuli.

V [Ing] ďalej nájdeme, ako sa reaktívne programovanie vysporiada s týmito problémami, avšak tieto detaily presahujú rámec tejto práce.

Meteor využíva paradigmu reaktívneho programovania, aby automaticky reflektoval zmeny v databáze do užívateľského rozhrania bez nutnosti písania riadiaceho kódu, ktorý by musel manuálne kontrolovať, či sa zmenili dáta v databáze a následne by manuálne aktualizoval relevantné komponenty užívateľského rozhrania.

Reaktívne programovanie v Meteore je zastrešené podprojektom Tracker. Pomocou knižnice Tracker vieme implementovať reaktívne zdroje dát a prepojiť ich s reaktívnymi konzumentmi (napríklad užívateľské rozhranie) bez toho, aby sme museli implementovať mechanizmy na sledovanie

zmien v dátach a aktualizáciu konzumentov. Na ukážke 3.1 môžeme vidieť náhľad kódu, ktorý zabezpečuje, že vždy, keď sa zmení hodnota výrazu `Session.get("currentPostId")`, zaregistruje sa nový odber komentárov vzhľadom na hodnotu `currentPostId`, ktorá predstavuje identifikátor aktuálne zvoleného článku. `Session` je reaktívny zdroj dát, takže vždy, keď `Session.get("currentPostId")` zmení hodnotu, Tracker zavolá anonymnú funkciu, ktorá je definovaná v ukážke a zabezpečí sa odoberanie komentárov k aktuálnemu článku.

```
Tracker.autorun(function () {
  Meteor.subscribe("comments", Session.get("currentPostId"));
});
```

Obr. 3.1: Ukážka reaktívneho kódu vo frameworku Meteor

Tracker prináša reaktívne programovanie do jazyka JavaScript, ktorý je vo svojej podstate imperatívny. Nad knižnicou Tracker je postavený šablónový systém Meteoru, nazvaný Blaze. Prostredníctvom systému Blaze vieme v Meteore vyvíjať HTML šablóny s užívateľským rozhraním, ktoré sa automaticky aktualizuje, kedykoľvek sa zmenia dáta, ktoré zobrazuje. Pozrime sa ďalej na spôsob, akým Meteor pracuje s dátami.

Meteor je postavený nad NoSQL databázou MongoDB, ktorú si najprv predstavíme, nakoľko je významná aj pre výsledky tejto práce, ako ukážeme v štvrtej kapitole.

3.1.2 Synchronizácia dát

Spôsob, akým Meteor pristupuje k práci s databázou je dôvod, prečo Meteor považujeme za unikátny a inovatívny framework a zároveň prečo je významný pre túto diplomovú prácu. V klasických webových aplikáciách klient dokáže

pracovať s databázou jedine prostredníctvom serverovej časti aplikácie, ktorá je zodpovedná za prístup k databáze, formátovanie a spracovanie dát. Meteor prináša databázu priamo na klienta, v podobe implementácie MongoDB, nazvanej *Minimongo*, ktorá pracuje priamo vo webovom prehliadači v operačnej pamäti. Minimongo sa oproti štandardnému MongoDB líši tým, že nie je persistentné a nepodporuje indexy. Ani jeden z týchto rozdielov však nepredstavuje neprekonateľný problém, nakoľko dáta na klientovi sú relevantné len po dobu, počas ktorej užívateľ pracuje s aplikáciou a objem dát by mal byť rozumne malý (rozhodne nie úplná kópia serverovej databázy).

Minimongo na strane klienta je synchronizované s MongoDB, ktoré je súčasťou servera. Klient komunikuje výhradne s lokálnou databázou. Vždy, keď do nej zapíše nejakú zmenu (pridanie nového dokumentu, zmena alebo zmazanie existujúceho), Meteor túto zmenu okamžite propaguje na server. V prípade, že zmena úspešne prejde validačným mechanizmom na strane servera, je zapísaná do serverovej databázy a propagovaná na ostatných klientov. Je dôležité uviesť, že serverová databáza je nadradená klientským databázam v zmysle validity dát. Môžeme si všimnúť, že klient, ktorý takúto zmenu spôsobil, môže z hľadiska užívateľského rozhrania reagovať okamžite, nakoľko pracuje s lokálnou databázou a pred zápisom prešiel rovnakými validačnými mechanizmami ako zápis na serveri. Pre dôveryhodných klientov je potvrdenie o zápise zo servera len formalitou. V prípade, že server z nejakej príčiny zmenu odmietne, klient je zodpovedný za vrátenie lokálnej databázy do pôvodného stavu.

Meteor funguje na modeli publish - subscribe, ktorý sme detailne rozobrali v prvej kapitole tejto práce. Minimongo na klientovi neodzrkadluje celú databázu na serveri, ale len konkrétne vybrané dotazy, ktoré server dopredu definuje a klient si zvolí odoberanie ich výsledkov. Na ukážke 3.2 môžeme vidieť spôsob, akým serverová časť aplikácie dokáže sprístupniť konkrétny dotaz klientom, v tomto príklade server sprístupňuje všetky články, ktoré sú

buď verejné, alebo platí, že autorom článku je prihlásený užívateľ. Na strane klienta stačí len požiadať o odoberanie výsledkov z dotazu `public_posts` a Meteor sa postará o automatickú synchronizáciu s lokálnym minimongom, kde v kolekcii `public_posts` nájdeme vždy aktuálne výsledky.

```
Posts = new Mongo.Collection("posts");

Meteor.publish("public_posts", function () {
  return Posts.find({$or: [
                             {"public": true},
                             {author: this.loggedUserId}
                           ]});
});
```

Obr. 3.2: Ukážka kódu pre poskytnutie kolekcie klientom

Poll-and-diff algoritmus

Ako uvádza [Dav], Meteor až do verzie 0.7.0 synchronizoval dáta medzi klientom a serverom metódou poll-and-diff. Na ukážke 3.3 vidíme, ako tento algoritmus funguje pri synchronizácii výsledkov jedného dotazu vo forme pseudokódu.

Poll-and-diff algoritmus repetitívne spúšťa preddefinovaný dotaz a pamätá si výsledky z predošlej iterácie. Na základe predošlých výsledkov dokáže vypočítať rozdiel výsledkov, výstup tejto operácie si môžeme predstaviť ako postupnosť operácií `add`, `remove` a `change` s príslušným identifikátorom dokumentu a dátami. Postupnosť operácií je následne odoslaná všetkým klientom, ktorí požiadali o odoberanie výsledkov daného dotazu. Algoritmus opakuje svoju činnosť každých desať sekúnd.

```
previous_result = [];

while length(subscribers) > 0 {
  result = Mongo.query(...);
  operations = diff(previous_result, result);
  previous_result = result;
  send(subscribers, operations);
  sleep(10);
}
```

Obr. 3.3: Poll-and-diff algoritmus v Meteor verzii 0.7.0 a nižšie

Pozrime sa na jednu iteráciu poll-and-diff algoritmu z hľadiska časovej a pamäťovej zložitosti. Z hľadiska časovej zložitosti považujeme tento algoritmus za efektívny, nakoľko jeho zložitosť závisí najmä od samotného dotazu na MongoDB, vzhľadom na jeho účel, použitie indexov a iné faktory. Výpočet rozdielu medzi zapamätanými výsledkami a novými výsledkami ja v prípade Meteoru lineárny od počtu výsledkov. Pamäťová zložitosť je lineárna od počtu výsledkov, nakoľko si ich pamätáme medzi iteráciami a rovnaká pamäť postačuje aj pre výpočet rozdielu.

Zo spôsobu, akým poll-and-diff algoritmus funguje by sa mohlo zdať, že stačí, aby bol spustený pre každý dotaz osobitne a všetci odoberatelia (klienti) môžu zdieľať jeden beh algoritmu. Problém spôsobujú parametrizované dotazy, ktoré pre každého klienta vracajú rôzne výsledky v závisloti od parametrov klienta. Príklad parametrizovaného dotazu sme mohli vidieť v ukážke 3.2, ktorá prostredníctvom parametra `author` umožňuje každému klientovi odoberať len jeho články. V najhoršom možnom prípade môže aplikácia postavená na frameworku Meteor definovať len parametrizované dotazy, čím prakticky vyžaduje, aby existovala bežiaca inštancia poll-and-diff algo-

ritmu pre všetky dvojice (klient, dotaz), pričom pre každú dvojicu zbehne jedna iterácia každých desať sekúnd.

Návrh poll-and-diff algoritmu síce optimalizuje veľkosť toku dát na klientov, nakoľko im posiela iba zmeny v databáze oproti ich stavu, avšak kladie zbytočné nároky na serverovú časť aplikácie, nakoľko dookola kontroluje stav databázy a zisťuje, či sa niečo zmenilo. Problémom zostáva aj fakt, že v najhoršom možnom prípade kontroluje stav pre každého klienta a dotaz osobitne. Riešenie, ktoré predstavíme v ďalšej kapitole, adresuje najmä tieto problémy a rieši ich efektívne.

Oplog tailing

Nedostatky algoritmu poll-and-diff si uvedomili aj tvorcovia frameworku Meteor, keďže sa prejavili v praxi v produkčných aplikáciách. Z hľadiska tejto práce považujeme za dôležité doplniť aj spôsob, akým vývojári frameworku Meteor riešia synchronizáciu dát v súčasnosti. Ich nový prístup je postavený na podobnej metóde, akú využijeme v tejto práci a volá sa oplog tailing, ako nájdeme v [Dav].

Oplog je skrátený názov pre špeciálnu kolekciu v MongoDB, ktorá zaznamenáva zmeny v databáze v priebehu času. Oplog kolekcia je interná funkcia MongoDB, ktorú Mongo používa pri horizontálnom škálovaní a vytváraní dababázových replík (replica-set). Viac informácií o replikácii v MongoDB sa môžeme dozvedieť v [Mon]. Oplog je implementovaný ako štandardná kolekcia v MongoDB, jednotlivé dokumenty predstavujú operácie, ktoré sa v databáze uskutočnili. Pozrime sa na ukážku 3.4 jedného dokumentu z kolekcie oplog, aby sme zistili, aké informácie nám oplog poskytuje. Prítomný je atribút `ts`, teda časová pečiatka, kedy bola daná operácia vykonaná. Ďalší zaujímavý atribút je `h`, ktorý obsahuje unikátny identifikátor pre každú operáciu. Atribút `op` kategorizuje operácie podľa ich významu a nadobúda nasledujúce hodnoty:

- "**i**" - označuje operáciu vkladania nového dokumentu (insert)
- "**u**" - označuje úpravu existujúceho dokumentu (update)
- "**d**" - predstavuje vymazanie dokumentu (delete)

```
{
"ts" : Timestamp(1395663575, 1),
"h" : NumberLong("-5872498803080442915"),
"v" : 2,
"op" : "i",
"ns" : "mydb.posts",
"o" : {
    "_id" : ObjectId("533022d70d7e2c31d4490d22"),
    "author" : "Name of the author",
    "title" : "Post title"
    }
}
```

Obr. 3.4: Príklad dokumentu z oplog kolekcie v MongoDB

Atribút `ns` (namespace) podáva informáciu o kontexte, v ktorom bola operácia vykonaná. Každá operácia môže byť vykonaná vzhľadom na konkrétnu databázu, alebo kolekciu nejakej databázy. Posledný, avšak nemenej dôležitý atribút je `o`, ktorý hovorí o konkrétnej zmene dát, ktorú operácia spôsobila. Dokument môže obsahovať ešte ďalšie atribúty, ktoré však pre metódu oplog tailing nie sú zaujímavé.

Ako názov metódy oplog tailing napovedá, synchronizácia dát funguje na báze sledovania oplogu (tailing) takým spôsobom, že na začiatku vykonáme dotaz ako v prípade algoritmu poll-and-diff a následne pozeráme na pribúdajúce operácie v oplogu a zisťujeme, či sú relevantné pre výsledky daného

dotazu. Akým spôsobom Meteor posudzuje relevanciu operácie vzhľadom na konkrétny dotaz je otázne, nakoľko jedinou dokumentáciou je samotný zdrojový kód, ktorého analýza nie je predmetom tejto práce, avšak naše riešenie ukáže spôsob, akým vieme tento netriviálny problém riešiť. Je dôležité uvedomiť si, že na sledovanie oplogu je potrebná len jedna jediná inštancia algoritmu, ktorý sleduje stav databázy, oproti množstvu inštancií závislého od počtu dotazov a klientov pri algoritme poll-and-diff. Dotaz ktorý zisťuje posledné vykonané inštancie od nejakého pevného časového bodu (poprípade od konkrétnej, naposledy spracovanej operácie) je efektívnejší, ako dotaz, ktorý opakovane vykonáva poll-and-diff, aby zistil zmeny v databáze, keď berieme do úvahy, že poll-and-diff môže vykonávať ľubovoľne zložitý dotaz.

Riešenie, ktoré Meteor implementoval namiesto poll-and-diff algoritmu považujeme za efektívne a utvrdilo nás to v presvedčení o riešení, ktoré prinášame v tejto práci. Za nevýhodu oplog tailingu považujeme silné previazanie s databázou MongoDB, nakoľko oplog je jej interná funkcia. Naše riešenie je síce tiež implementované nad MongoDB, avšak rovnako dobre by fungovalo aj na iných databázových systémoch, ktoré majú ekvivalentnú silu dotazovacieho jazyka a nie je závislé na žiadnej internej funkcionalite MongoDB.

3.2 Datomic

V predošlej sekcii sme ukázali plnohodnotný framework pre tvorbu single page aplikácií a spôsob, akým rieši synchronizáciu dát medzi klientom a serverom. V tejto sekcii si predstavíme databázový systém Datomic, ktorý síce neposkytuje kompletnú sadu nástrojov pre tvorbu SPA, avšak prináša zaujímavý koncept, ktorý by sa teoreticky dal využiť aj v našom riešení.

Datomic je relatívne nový projekt, podľa [Dat], ktorý vznikol v roku 2012 a ukážková verzia (preview release) bola vydaná vo februári 2014 v dobe, keď naše riešenie už bolo naimplementované nad databázou MongoDB. Datomic

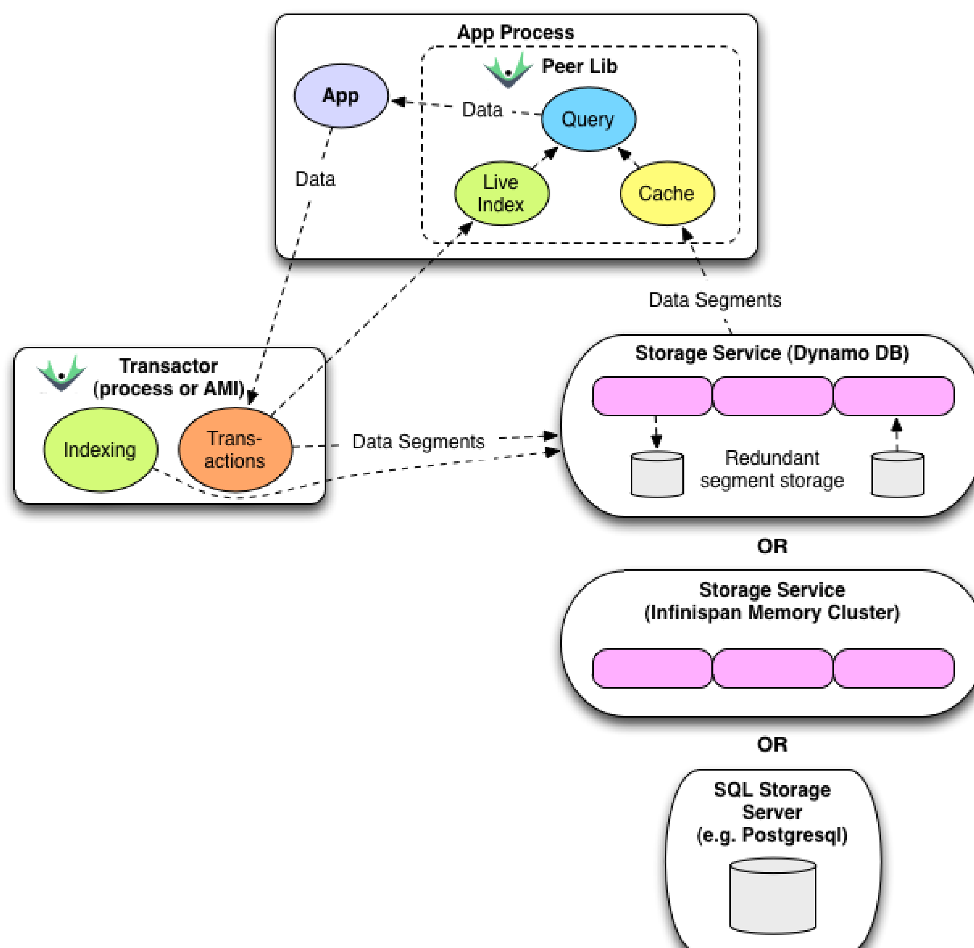
je komerčný proprietárny softvér, aj keď v limitovanej verzii ponúka aj voľne dostupnú verziu.

3.2.1 Architektúra

Ako nájdeme v [Dat], Datomic uchováva súbor faktov, ktoré sú nemenné, čím sa významne odlišuje od iných databázových systémov. Do databázy je možné pridávať nové fakty, ktoré môžu nahradiť existujúce v zmysle aktuálnej platnosti. Stav Datomic databázy môžeme definovať ako množinu faktov, ktoré sú v platnosti v nejakom konkrétnom zadanom čase. Ak čas nie je zadaný, dostávame aktuálny stav, v ktorom máme k dispozícii všetky aktuálne platné fakty.

Prístup Datomicu k správe dát inovuje klasický "update-in-place" prístup iných databázových systémov (napríklad MySQL, PostgreSQL, MongoDB, atď), ktoré podporujú len udržiavanie aktuálnych dát a pri zmene dát dochádza k strate pôvodných dát. Datomic akumuluje fakty v zmysle udalostí, ktoré sa stali v čase.

Predstavme si spôsob práce pri štandardných databázových systémoch. Pre prácu s databázou v prvom rade potrebujeme spojenie, cez ktoré s databázou komunikujeme. Následne cez toto spojenie posielame na databázu dotazy a dostávame naspäť výsledky dotazov. Datomic sa odklonil od tohoto štandardného postupu a prináša svoju Peer Library, ktorá sa integruje do aplikácie a aplikácia sa tak stane súčasťou databázového systému Datomic. Na ukážke 3.5 môžeme vidieť kompletnú architektúru Datomic databázy. Hlavným úložiskom dát je na diagrame zobrazená ako *Storage Service*. Úložisko môže byť čokoľvek od klasického súborového systému, cez rôzne databázové systémy (napríklad PostgreSQL, alebo iné databázy postavené na SQL) až po sofistikované distribuované a škálovateľné databázové riešenia ako DynamoDB, alebo Cassandra.



Obr. 3.5: Architektúra databázového systému Datomic

V úložisku sú uložené všetky fakty, ktoré boli do databázy pridané. Fakty sú z úložiská čítané peer procesmi. Peer je ľubovoľný proces, ktorý pracuje s Datomic databázou prostredníctvom Datomic Peer Library a na ukážke 3.5 je zobrazený ako *App process*. Nakoľko fakty v databázovom úložisku sú nemenné, peer procesy využívajú vlastnú cache pamäť a sami sa stávajú replikou dát. V prípade naplnenia cache pamäti sa používa technika LRU

(least recently used) pre uvoľnenie nepotrebných faktov z cache pamäti. Výhodou tejto cache pamäte je, že dotazy sú exekuované v peer procesoch nad lokálnou databázou a fakty z úložiska sú čítané len v prípade, že sa v lokálnej cache pamäti nenachádzajú. Cieľom tejto architektúry a nemennosti faktov je preniesť väčšinu read-operácií priamo do aplikácie pre okamžitý prístup k dátam bez nutnosti zaťažovať úložisko faktov. Tento prístup je efektívnejší ako napríklad v databázach postavených na jazyku SQL, kde čítacie transakcie súťažia so zapisovacími transakciami a do hry vstupuje uzamykanie dát. Túto silu navyše Datomic získava práve kvôli nemennosti faktov.

Predstavíme posledný komponent architektúry Datomic databázy z ukážky 3.5, ktorý sme ešte nespomenuli - správcu transakcií, pomenovaný *Transactor*. Peer procesy zapisujú nové fakty do úložiska požiadaním správcu transakcií (samostatný proces), ktorý má prístup k úložisku a fakty zapisuje prostredníctvom transakcií, ktoré spĺňajú ACID. Správca transakcií navyše oznamuje existenciu nových faktov všetkým peer procesom, aby si mohli nové fakty pridať do svojej cache pamäte.

3.2.2 Reprezentácia dát

Každý databázový systém je postavený okolo nejakej základnej jednotky, napríklad dokument, riadok, alebo relácia. Podľa [Dat], základná jednotka Datomic databázy sa nazýva datom, ktorý predstavuje jeden fakt. Datom je definovaný nasledujúcimi informáciami:

- **Entita** - unikátny identifikátor konkrétnej entity, ktorej sa fakt týka
- **Atribút** - identifikátor atribútu v tvare *:<namespace>/<name>*, napríklad *:post/author*. *Namespace* nie je povinná zložka identifikátora atribútu, avšak je odporúčaná, aby sa predišlo vzniku kolízií mien atribútov.

- **Hodnota** - hodnota atribútu
- **Transakcia** - identifikátor transakcie, ktorá datom pridala do databázy
- **Indikátor pridania** - boolean hodnota podľa toho, či bol datom pridaný ako nový, alebo či nahradil existujúci fakt

Schéma

Schéma v databáze Datomic určuje množinu atribútov, ktoré môžeme použiť v datomoch. Atribúty sú charakterizované menom, dátovým typom a kardinalitou. Detailnejší pohľad na vlastnosti atribútov môžeme nájsť [Dat]. Atribúty môžu byť v relácii s ľubovoľnými entitami, nakoľko Datomic naschvál neposkytuje spôsob, ako obmedziť použitie určitej množiny atribútov len pre niektoré entity. Schéma v Datomic databáze poskytuje preto väčšiu voľnosť pri návrhu aplikácie, ako napríklad v databázach založených na jazyku SQL, kde je potrebné explicitne definovať štruktúru tabuliek.

3.2.3 Dotazovací jazyk

V tejto sekcii sa pozrieme na jazyk, ktorým vieme formulovať dotazy pre prácu s Datomic databázou. Dotazovací jazyk Datomic databázy je postavený na deklaratívnom jazyku Datalog, ktorý typicky pozostáva z databázy známych faktov a pravidiel, ktorými vieme odvodzovať nové fakty z existujúcich. Dotaz v jazyku Datalog pozostáva z čiastočnej špecifikácie faktov a pravidiel, na základe ktorej je Datomic schopný nájsť všetky fakty, ktoré takému dotazu vyhovujú. Výber Datalogu ako dotazovacieho jazyka má nasledujúce výhody:

- **Jednoduchosť** - základom dotazu je zoznam výrazov, ktoré môžu obsahovať premenné a úlohou Datomic databázy je nájsť tie fakty, ktoré daným výrazom a hodnotám premenných vyhovujú
- **Deklaratívnosť** - Datalog je deklaratívny jazyk, podobne ako SQL, čo v praxi znamená, že špecifikujeme, aké fakty má výsledok dotazu obsahovať a nie spôsob, akým sa k výsledným faktom dopátrame - túto prácu prenechávame Datomic databáze, aby ju efektívne vykonala.
- **Logickosť** - Datalog je logický jazyk, založený na logických implikáciách. Výhodou je, že JOIN operácie sú implicitné a nie je potreba ich explicitne špecifikovať, ako napríklad v jazyku SQL. Operácia JOIN je implicitne použitá v prípade, že použijeme rovnakú premennú v aspoň dvoch výrazoch v dotaze, ako napríklad v ukážke 3.7.

Na ukážke 3.6 môžeme vidieť príklad databázy faktov, v ktorej sme nahradili identifikátory entít menami osôb pre zjednodušenie príkladu.

```
[
  [Adam   :age   35]
  [Martin :age   49]
  [Daniel :age   49]
  [Martin :likes pizza]
  [Adam   :likes pasta]
  [Daniel :likes sushi]
]
```

Obr. 3.6: Príklad databázy faktov v Datomic databázovom systéme

Štruktúra dotazu má nasledovnú formu: [*:find <zoznam premenných> :where <zoznam výrazov>*]. Premenné začínajú znakom *?*, napríklad *?e*. Pre ilustráciu dotazovacieho jazyka uveďme príklad dotazu, ktorý v databáze z ukážky 3.6 nájde všetky osoby vo veku 49 rokov spolu s preferovaným jedlom. Takýto dotaz môžeme vidieť v ukážke 3.7.

```
[:find ?e ?x :where [?e :age 49] [?e :likes ?x]]
```

Obr. 3.7: Príklad dotazu v Datomic databázovom systéme

Za kľúčovým slovom *find* definujeme premenné, ktoré tvoria výsledok dotazu a za kľúčovým slovom *where* vystupuje zoznam výrazov. Podľa očakávania, výsledok dotazu z ukážky 3.7 je zoznam dvojíc [[*Martin, pizza*][*Daniel, sushi*]].

V tejto sekcii sme ukázali základnú funkcionalitu dotazovacieho jazyka v Datomic databáze. Detailný prehľad všetkých možností, ktoré sú v dotazovaciom jazyku Datomic databázy dostupné, nájdeme v [Dat].

3.2.4 Synchronizácia dát

Pozrime sa, ako by Datomic databáza mohla byť užitočná pri riešení našeho problému synchronizácie dát. Na úvod by sme chceli spomenúť, že využitie Datomic databázy kladieme len do teoretickej roviny, nakoľko vznikla v dobe, keď už naše riešenie bolo navrhnuté a implementované. Ďalší faktor, negatívne ovplyvňujúci jej využitie v tejto práci, je uzavretosť zdrojového kódu, nakoľko naše riešenie je open-source.

Databázový systém Datomic považujeme za zaujímavý práve pre možnosť jednoduchej a efektívnej práce s históriou dát. Ako ukážeme v nasledujúcej kapitole, naše riešenie je postavené na budovaní histórie zmien v databáze, ktorú nám Datomic už automaticky poskytuje. Pre lepšiu predstavu, história zmien v našom riešení je štruktúrovo podobná oplogu v MongoDB, ktorý sme opísali v prvej časti tejto kapitoly. Na základe informácií, ktoré sme doteraz o Datomicu uviedli je evidentné, akú obrovskú silu Datomic poskytuje pri práci s historickými dátami. Pri synchronizácii výsledkov dotazu na klienta si stačí na klientovi pamätať čas poslednej synchronizácie a následne zohľadňovať len datomy, ktoré boli pridané po tomto čase. Datomic bol navrhnutý práve pre zjednodušenie takýchto operácií v porovnaní s ostatnými databázami.

Ostáva nám problém, ako zohľadniť, ktoré pridané datomy sú relevantné pre konkrétny dotaz. Riešenie tohoto problému ukážeme v nasledujúcej kapitole, avšak naše riešenie je vhodné len pre NoSQL databázy ako napríklad MongoDB, nakoľko pracujeme so slabším dotazovacím jazykom (nepracuje s reláciami). Datomic sa však opiera o Datalog, ktorý je založený na reláciách a tie nám spôsobujú problém.

Pozrime si príklad, v ktorom ukážeme, aký problém relácie spôsobujú. Na ukážke 3.8 máme databázu faktov, v ktorej platia nasledujúce vzťahy a podmienky:

- **osoba - jedlo** je many-to-many relácia

- **jedlo - cena** je one-to-one relácia
- mená osôb a jedál sú náhrady za identifikátory entít pre jednoduchšiu prácu s databázou

```
[
  [Martin :likes pizza]
  [Adam   :likes pasta]
  [Daniel :likes sushi]
  [pizza  :costs 34]
  [pasta  :costs 23]
  [sushi  :costs 24]
]
```

Obr. 3.8: Príklad databázy faktov v Datomic databázovom systéme

Majme dotaz z ukážky 3.9, ktorého výsledky chceme priebežne synchronizovať na klientov. Synchronizačný protokol, ktorý detailne predstavíme v nasledujúcej kapitole, pracuje na základe analýzy zmien v databáze od času poslednej synchronizácie klienta. Skúsme aj v tomto prípade vychádzať len z datomov, ktoré pribudnú do databázy z ukážky 3.8.

```
[:find ?e :where [?e :likes ?x] [?x :costs 34]]
```

Obr. 3.9: Príklad dotazu v Datomic databázovom systéme

Dotaz 3.9 hľadá tie osoby, ktoré majú radi nejaké jedlo, ktoré má cenu 34. Na základe databázy 3.8 je výsledkom dotazu [[*Martin*]], pričom takýto

výsledok má momentálne aj klient. Iný klient do databázy 3.8 zapíše datom [*sushi :costs 34*], čím sa zmení výsledok dotazu 3.9 na [[*Martin*], [*Daniel*]].

Problém spočíva v tom, že analýzou samotného datomu [*sushi :costs 34*] nevieme posúdiť, že do výsledku dotazu pribudla ďalšia osoba bez toho, aby sme vykonali celý pôvodný dotaz (čo samozrejme nechceme, nakoľko ten pracuje nad celou databázou). Abstraktnejšie formulované, problém spôsobujú dotazy obsahujúce výrazy v tvare [*?a* <*atribút*> *?b*], ktoré zväzujú premenné *?a* a *?b* do relácie. V takomto dotaze môže zmena atribútu entity A (zmena ceny jedla *sushi* na 34) ovplyvniť prítomnosť entity B (osoba *Martin*) vo výsledku dotazu a túto skutočnosť nevieme rozhodnúť len na základe analýzy datomu, ktorý spôsobil zmenu atribútu entity A.

V prípade, že by sme chceli Datomic použiť v spolupráci s naším synchronizačným protokolom, museli by sme obmedziť silu dotazovacieho jazyka. Obmedzenie dotazovacieho jazyka by spočívalo v dvoch podmienkach:

1. **V dotaze za kľúčovým slovom *find* môže byť definovaná najviac jedna premenná.** V praxi to znamená, že vo výsledku dotazu by mohol byť maximálne jeden typ entity. Pod typom entity rozumieme sémantický typ entity, ako napríklad *osoba*, alebo *jedlo* v našom prípade.

2. **Premenná uvedená za kľúčovým slovom *find* je jediná premenná, ktorá vystupuje v celom dotaze a smie sa nachádzať len na prvej pozícii výrazu.** Touto podmienkou sme zamietli výrazy, ktoré zväzujú premenné do relácii, pretože nedovoľujeme použitie premennej na tretej pozícii výrazu (na druhej pozícii výrazu je meno atribútu). Prakticky umožňujeme len filtrovanie entít podľa hodnôt atribútov.

Databázový systém Datomic je inovatívny projekt, ktorý prináša nový pohľad na uchovávanie dát a architektúru databázového systému svojou de-

centralizáciou komponentov. Prístup k stavu databázy v ľubovoľnom čase by mohol byť výraznou pomocou pre naše riešenie problému synchronizácie, avšak je nutné myslieť na problémy, ktoré spôsobuje sila dotazovacieho jazyka Datalog. Prekážkou použitia Datomic databázy pre naše riešenie by mohli byť aj licenčné podmienky, nakoľko Datomic nie je open-source softvér.

3.3 Firebase

V tejto sekcii sa zoznámime s posledným projektom, ktorý je zaujímavý pre túto prácu v ohľade na synchronizáciu dát. Podľa [Fir], Firebase je hostingové riešenie poskytujúce databázu ako službu (database-as-a-service). Hlavnou myšlienkou je poskytovať databázu vo forme plnohodnotného backendu pre vývojárov, ktorí sa môžu vo svojej aplikácii plne sústrediť len na vývoj klientskej časti aplikácie. Firebase vznikol z projektu Envolve, ktorý umožňoval vývojárom integrovať do ich webových aplikácií online chat. Firebase je proprietárne riešenie (closed-source), ktoré je dostupné len vo forme služby, čím odbremeňuje vývojárov od nutnosti riešiť serverovú časť aplikácie z hľadiska hostingu, nasadzovania a správy aplikácie. V súčasnej dobe nie je dostupný žiadny spôsob, akým by bolo možné nasadiť Firebase v privátnom prostredí v prípade potreby kontroly nad dátami.

Firebase nie je limitovaný výhradne pre webové aplikácie, nakoľko okrem podpory REST a JavaScript API pre webové aplikácie poskytuje API aj pre mobilné platformy Android a iOS v natívnych jazykoch, ako nájdeme v [Fir]. Uzavretosť kódu neumožňuje podrobnejšiu analýzu spôsobu, akým Firebase funguje, avšak pre túto prácu je zaujímavý z pohľadu rozhrania, ktorým komunikuje s klientami. Klienti si udržujú lokálnu cache pamäť synchronizovaných dát a server im posiela len zmeny oproti ich lokálnemu stavu, čím principiálne navonok dostávame podobnú funkcionalitu ako pri našom synchronizačnom protokole.

Veľkým pozitívom celého riešenia z hľadiska vývoja je možnosť sústrediť sa len na vývoj klientskej časti aplikácie bez nutnosti riešiť serverovú časť. Nakoľko Firebase je komerčné riešenie, táto pohodlnosť má však svoju cenu, doslovne.

Nedostatky vidíme v schopnostiach dotazovacieho jazyku, ktorý ponúka slabšie možnosti práce s dátami v porovnaní s MongoDB. Naše riešenie je schopné pracovať s ohľadom na všetky možnosti dotazovacieho jazyka MongoDB. Pozrime sa na niektoré dotazy, ktoré nám spôsobujú problémy:

- **Logické spojky AND a OR** - nevieme tvoriť dotazy, ktoré filtrujú dáta na základe viacerých atribútov pomocou logický spojok AND a OR
- **Regulárne výrazy** - vyhľadávanie dát pomocou regulárnych výrazov nie je podporované
- **Triedenie výsledkov** - triedenie je podporované len pre jeden atribút na dotaz

Firebase považujeme za prínosné a pohodlné riešenie pre tvorbu webových a mobilných aplikácií, nakoľko rieši vývojárom podstatné množstvo problémov týkajúcich sa synchronizácie klienta so serverom a vývoja serverovej časti aplikácie. Táto pohodlnosť má však svoju cenu - finančnú a aj funkčnú v podobe obmedzeného a ťažkopádneho dotazovacieho jazyka. Dôležitý je aj fakt, že vývojár nemá absolútne žiadnu kontrolu nad dátami, nakoľko Firebase funguje v podobe služby.

Kapitola 4

Synchronizačný protokol

V štvrtej kapitole tejto práce sa detailne zameriame na priebeh samotnej synchronizácie. V úvodnej sekcii tejto kapitoly ilustrujeme priebeh inicializácie protokolu (handshake) a jeho nasledovné správanie. V ďalšej sekcii vysvetlíme čo je to história kolekcie a detailne opíšeme jej štruktúru pre lepšie pochopenie výhod, ktoré prináša. Následne analyzujeme algoritmy, ktoré sú zodpovedné za výpočet rozdielov medzi pohľadom klienta a aktuálnejším pohľadom servera.

4.1 Správanie protokolu

Komunikačný protokol si môžeme predstaviť ako systém pravidiel a postupov, ktoré musia komunikujúce strany dodržať, aby spolu mohli úspešne nadviazať a udržiavať komunikáciu. V súčasnej verzii je synchronizačný protokol postavený na kombinácii štandardného HTTP 1.1 protokolu a websocketov. Komunikácia na spodnej vrstve je iniciovaná zo strany klienta vo forme HTTP požiadavky okrem jediného prípadu, v ktorom server upozorňuje klienta na aktuálnejšiu verziu pohľadu. V tomto prípade server iniciuje komunikáciu na klienta prostredníctvom websocketov. Synchronizácii klienta predchádza jej inicializácia, ktorú predstavíme v nasledujúcej podsekcii.

4.1.1 Inicializácia protokolu

Komunikáciu otvára klient požiadavkou o pridelenie klientského UID (unikátny identifikátor). Klientské UID je generované na strane servera a jeho unikátnosť je zaručená súhrou viacerých faktorov. Generátor skladá UID z nasledujúcich troch zložiek:

1. čas pridelenia UID v milisekundách
2. náhodné počiatočné číslo generátora (seed) z intervalu $\langle 0, 2^{16} - 1 \rangle$
3. automaticky inkrementované počítadlo po každom vygenerovaní jedného UID

Následne je každá zložka prekonvertovaná do hexatridecimálnej číselnej sústavy (sústava zo základom 36), ktorá okrem znakov 0 - 9 používa ešte všetkých 26 znakov latinskej abecedy. Zreťazením týchto troch zložiek v uvedenom poradí dostávame unikátne klientské ID.

Po prijatí svojho UID sa klient môže prihlásiť na odber nejakého vydaného klientského pohľadu (v zmysle modelu *publish - subscribe*, ktorý sme uviedli v prvej kapitole tejto práce). Klient má na výber dve možnosti, ktorými dokáže vyžiadať dáta od servera:

- **get_data** - klient požiada server o všetky aktuálne dostupné dokumenty v danom pohľade
- **get_diff** - klient si vyžiada len rozdiel (diff) medzi svojim klientským pohľadom a aktuálnym pohľadom servera

Majme kolekciu k a označme si aktuálny pohľad servera na kolekciu k symbolom α_k a označme postupnosť všetkých akcií, ktoré boli na kolekcii k vykonané, symbolom δ_k . Pod akciou rozumieme pridanie celého dokumentu,

odobratie celého dokumentu alebo modifikovanie ľubovolného počtu atribútov jedného dokumentu v jednom čase (v zmysle synchronizačného protokolu sa jedná o atomickú operáciu). Symbol α'_k označuje klientský pohľad na kolekciu k, ktorý sa synchronizuje s pohľadom α_k . V prípade, že α'_k ešte neobsahuje žiadne dokumenty, pretože ešte nikdy nebol synchronizovaný, klient pošle na server požiadavku **get_data** pre získanie všetkých dokumentov v pohľade α_k a verzie $v(\alpha_k)$.

4.1.2 Beh protokolu

Po prijatí celého pohľadu klient pokračuje jeho synchronizovaním na aktuálny pohľad. V prípade, že v pohľade α_k na strane servera nastali nejaké zmeny, server pošle notifikáciu o zmene všetkým klientom a tí si následne prostredníctvom požiadavky **get_diff** vypýtajú rozdiel pohľadov α_k a α'_k . Rozdiel $\alpha_k - \alpha'_k$ medzi pohľadmi α_k a α'_k definujeme ako podpostupnosť akcií $\delta' = a_0, a_1, .., a_n$ postupnosti δ_k , kde $n \in \mathcal{N} \wedge a_i$ predstavuje i-tu akciu vykonanú na nejakom dokumente. Ak aplikujeme podpostupnosť akcií δ' na α'_k , klientský pohľad α'_k bude totožný s aktuálnym pohľadom α_k . Zodpovednosťou serverovej aplikácie je vypočítať správny rozdiel δ' na základe postupnosti δ_k a verzie α'_k . Verziu $v(\alpha'_k)$ chápeme ako index poslednej akcie, ktorá bola aplikovaná na α'_k v postupnosti δ_k . V praxi to znamená, že rozdiel $\alpha_k - \alpha'_k$ chceme počítať len zo zmien, ktoré boli na kolekcii k vykonané až po poslednej synchronizácii klientského pohľadu α'_k . V nasledujúcej sekcii vysvetlíme, ako presne vyzerá postupnosť δ_k a aké benefity jej štruktúra prináša.

4.2 História kolekcie

V predošlej sekcii sme definovali postupnosť δ_k ako postupnosť všetkých akcií, ktoré boli vykonané na kolekcii k. Postupnosť δ_k prakticky predstavuje

históriu zmien, ktoré sú uchované v rovnakom poradí, v akom nastali a ich aplikovaním na prázdnu množinu v zachovanom poradí dostaneme množinu k. Každá kolekcia k má svojú príslušnú históriu zmien δ_k , ktorá je uložená v databázovom úložisku pod menom kolekcie k so sufixom *_history*. Dokumentami kolekcie δ_k sú jednotlivé akcie a štruktúru akcií tvoria nasledujúce atribúty:

- `action` - typ akcie, môže byť pridanie dokumentu (+), odobratie dokumentu (-), alebo zmena ľubovoľného počtu hodnôt atribútov dokumentu (*)
- `id` - ID dokumentu, ktorého sa zmena týka
- `before` - kópia dokumentu pred aplikovaním akcie
- `after` - kópia dokumentu po aplikovaní akcie
- `v` - verzia (poradie) zmeny

Príklad obsahu histórie zmien δ_k môžeme vidieť v ukážke 4.1, na základe ktorej dostávame aktuálny pohľad α_k na celú kolekciu k v ukážke 4.2.

Ako sa dozvieme v nasledujúcich sekciách, takto navrhnutá štruktúra histórie zmien nám uľahčí prácu pri výpočte rozdieľov medzi dvomi pohľadmi. Ako nevýhoda sa môže javiť pamäťová redundancia, ktorú spôsobuje udržiavanie si v najhoršom prípade až dvoch úplných kópií v každom dokumente akcie. Pamäťová zložitosť pre udržanie $N = |k|$ dokumentov predstavuje $O(N + 2H)$, kde $H = |\delta_k|$. Argumentom proti tejto pamäťovej redundancii je fakt, že históriu zmien sa neoplatí udržiavať dlhšiu ako N, nakoľko je úspornejšie poslať nanovo celý pohľad α_k , ak $|\alpha_k - \alpha'_k| > N$, kde $|\alpha_k - \alpha'_k|$ predstavujú počet akcií v rozdieli medzi pohľadmi α_k a α'_k . Dostávame tým horné ohraničenie pre $H \leq N$ a teda dostávame lineárnu pamäťovú zložitosť $O(N + 2N) = O(N)$.

```
[
  action: +, id: #1, before: {},            after: {Adam, 20},   v: 1
  action: +, id: #2, before: {},            after: {Daniel, 32}, v: 2
  action: +, id: #3, before: {},            after: {Matej, 40},  v: 3
  action: *, id: #3, before: {Matej, 40},   after: {Martin, 45}, v: 4
  action: *, id: #1, before: {Adam, 20},    after: {Adam, 22},   v: 5
  action: +, id: #4, before: {},            after: {Peter, 20},  v: 6
  action: -, id: #4, before: {Peter, 20},   after: {},           v: 7
]
```

Obr. 4.1: História kolekcie *osoby*

```
[
  #1,  name: Adam,    age: 22
  #2,  name: Daniel,  age: 35
  #3,  name: Martin,  age: 45
]
```

Obr. 4.2: Kolekcie *osoby*

4.3 Synchronizácia filtrovaného pohľadu

Doteraz sme sa v tejto kapitole venovali synchronizačnému protokolu z hľadiska jeho správania a ukázali sme, ako si udržiavame pomocnú štruktúru zmien ku každej kolekcii v databázovom úložisku. V tejto sekcii uvedieme algoritmus, ktorý je zodpovedný za výpočet rozdielu medzi klientským pohľadom α'_k a aktuálnym pohľadom servera α_k . Vstupom algoritmu je kompletná história zmien δ_k a verzia klientského pohľadu $v(\alpha'_k)$. Výstupom je prirodzene postupnosť akcií $\delta' = a_0, a_1, .., a_n$, kde $i, n \in \mathcal{N} \wedge v(a_i) > v(\alpha'_k) \wedge v(a_i) < v(a_{i+1})$.

V prvej kapitole sme uvideli, že klientský pohľad môže byť definovaný tromi parametrami: filtrovaním, usporiadaním a limitovaním. V prípade, že klientský pohľad α'_k nie je definovaný žiadnym parametrom, výpočet rozdielu δ' je priamočiary. Z postupnosti $\delta_k = a_1, a_2, .., a_N$ stačí vybrať podpostupnosť akcií $a_i, a_{i+1}, .., a_N$, kde $i \in \mathcal{N} \wedge N = |\delta_k| \wedge i = v(\alpha'_k) + 1$.

Zaujímavejší je rozšírený algoritmus, ktorý berie do úvahy aj parameter filtrovania, teda vie vypočítať rozdiel dvoch pohľadov, ktoré podliehajú tomu istému filtru. Vstup algoritmu rozšírime o ďalší argument - filter pohľadu σ. Filter σ je reprezentovaný slovníkovou dátovou štruktúrou, v ktorej atribúty dokumentu predstavujú kľúče a ich hodnotami sú filtračné podmienky. Príklad reprezentácie filtra môžeme vidieť v ukážke 4.3.

```
{
  age: {
    $gt : 24
  }
}
```

Obr. 4.3: Filter pre pohľad *osoby staršie ako 24 rokov*

Algoritmus pre výpočet rozdielu medzi filtrovanými pohľadmi najprv nájde dve pomocné podpostupnosti δ_{before} a δ_{after}. Podpostupnosť δ_{before} je definovaná ako $\delta_{before} = \{a_i\}$, kde $i \in \mathcal{N} \wedge v(a_i) > v(\alpha'_k) \wedge a_i.before$ vyhovuje filtru σ (označenie $a_i.before$ znamená $before$ atribút akcie a_i). Akcie a_i z podpostupnosti δ_{before} predstavujú také akcie, ktoré vykonali zmenu na dokumentoch vyhovujúcich filtru σ. Analogicky je definovaná podpostupnosť δ_{after} s jedinou zmenenou podmienkou, kde $a_i.after$ vyhovuje filtru σ. Podpostupnosť δ_{after} obsahuje tie akcie, ktorých výsledkom sú dokumenty vyhovujúce filtru σ. Obidve podpostupnosti δ_{before} aj δ_{after} obsahujú len akcie s vyššou verziou, ako je verzia klientského pohľadu α'_k .

Algoritmus následne iteruje zjednotenie $\delta_{\cup} = \delta_{before} \cup \delta_{after}$ a vyhodnocuje prítomnosť akcie a (iterátora) v pomocných podpostupnostiach δ_{before} a δ_{after}. Nastávajú štyri možné prípady, z ktorých sú len tri zaujímavé pre náš algoritmus:

1. $a \in \delta_{before} \wedge a \in \delta_{after}$ - akcia a vykonala zmenu na dokumente, ktorý vyhovoval filtru σ a jej vplyvom sa táto vlasnosť nezrušila a dokument vyhovuje filtru σ aj po vykonaní akcie a. Táto akcia je vyhodnotená ako **zmena dokumentu**, ktorý patrí do pohľadu α'_k a preto je zahrnutá do výsledného rozdielu δ'.

2. $a \in \delta_{before} \wedge a \notin \delta_{after}$ - akcia a vykonala zmenu na dokumente, ktorý vyhovoval filtru σ a jej vplyvom sa táto vlasnosť zrušila a dokument už nevyhovuje filtru σ po vykonaní akcie a. Táto akcia je vyhodnotená ako **odobratie dokumentu**, ktorý patrí do pohľadu α'_k a preto je zahrnutá do výsledného rozdielu δ'.

3. $a \notin \delta_{before} \wedge a \in \delta_{after}$ - akcia a vykonala zmenu na dokumente, ktorý nevyhovoval filtru σ a jej vplyvom dokument vyhovuje filtru σ po vykonaní akcie a. Táto akcia je vyhodnotená ako **pridanie dokumentu**, ktorý nepatril do pohľadu α'_k a preto je zahrnutá do výsledného rozdielu δ'.

Po dokončení iterácie postupnosti $\delta_{\cup}$ algoritmus vypočítal výsledný rozdiel medzi aktuálnym pohľadom servera α_k a klientským pohľadom α'_k a môže svoju činnosť skončiť a vrátiť výslednú postupnosť δ'. V ďalšej sekcii predvedieme úpravy algoritmu, ktoré umožnia výpočet rozdielu pohľadov, ktoré okrem filtrovania využívajú aj usporiadanie a limitovanie - tzv. úplné pohľady, ako sme definovali v prvej kapitole.

4.4 Synchronizácia úplného pohľadu

Úplný pohľad predstavuje pohľad, ktorý okrem filtrovania dokumentov využíva aj usporiadanie a následné limitovanie počtu dokumentov. Príklad úplného pohľadu môžeme formulovať dotazom "desať najstarších dospelých osôb", v ktorom filtrujeme podmienkou dospelosti, triedime podľa veku a limitujeme počtom desať. Oproti predošlému algoritmu na výpočet rozdielu medzi filtrovanými pohľadmi rozšírime vstup algoritmu o ďalšie štyri argumenty.

1. **triedenie** γ
2. **limit** l
3. **hraničný dokument** *pivot* - predstavuje posledný dokument v klientskom pohľade α'_k . V zmysle tejto práce, všetky dokumenty, ktoré sú súčasťou úplného pohľadu sú menšie, alebo nanajvýš rovné ako pivot a všetky dokumenty, ktoré nepatria do pohľadu sú ostro väčšie ako pivot, vzhľadom na triedenie γ a limit l
4. **veľkosť klientského pohľadu** $|\alpha'_k|$ - počet dokumentov v α'_k

Tieto štyri argumenty dopĺňajú všetky predošlé vstupné argumenty - filter σ, $v(\alpha'_k)$ a δ_k . Argumenty *pivot* a $|\alpha'_k|$ posiela klient pri požiadavke **get_diff** pre úplný pohľad, nie je potreba uchovávať ich na strane servera.

Pri úplnom pohľade nemáme dostupné všetky dokumenty, ktoré vyhovujú filtru, ale len prvých l dokumentov, kde l je nejaká konečná hodnota určená pri definícii pohľadu. Triedenie určuje, ktorých l dokumentov má pohľad obsahovať. Triedenie a limit musia byť použité spolu, alebo vôbec. Použitie limitu bez triedenia považujeme za nezmyselné, nakoľko nevieme deterministicky určiť, ktoré dokumenty sa majú dostať do pohľadu. Použitie triedenia bez limitu je zbytočné, nakoľko samotný pohľad je množina, ktorá sama o

sebe nezachováva poradie. Z hľadiska obsahu klientského pohľadu nie je rozdiel medzi pohľadom bez triedenia a s triedením v prípade, že nie je použitý limit.

Pozrime sa, ako tento algoritmus pracuje. Prvý krok výpočtu je rovnaký ako v prípade filtrovaného pohľadu, ktorého výstupom je postupnosť δ' obsahujúca akcie a_i, ktoré určujú pridanie, odobratie, alebo zmenu nejakého dokumentu. Postupnosť δ' môže obsahovať aj akcie, ktoré nemajú žiadny vplyv na dokumenty v úplnom pohľade a týchto akcií sa chceme zbaviť. Algoritmus inicializuje novú lokálnu premennú c na hodnotu $|\alpha'_k|$, ktorú má k dispozícii od klienta. Majme tiež prázdnu postupnosť δ'', ktorá predstavuje výsledok algoritmu. Algoritmus potom iteruje postupnosť δ' a pre každú akciu a_i skúma nasledujúce možnosti:

- a_i **pridáva dokument**, atribút $a_i.after$ obsahuje celý dokument po pridaní, ako sme uviedli v sekcii o histórii kolekcie
  - $a_i.after \leq pivot$ - a_i pridáva dokument, ktorý by mal byť súčasťou α'_k , preto a_i je zahrnutá do výsledného rozdielu δ'' a zároveň c je zvýšená o 1.
  - $a_i.after > pivot$ - a_i pridáva dokument, avšak dokument je väčší ako $pivot$ a preto nepatrí do výsledného rozdielu δ''. Akcia a_i je ignorovaná.
- a_i **odoberá dokument**, atribút $a_i.before$ obsahuje celý dokument pred vymazaním
  - $a_i.before \leq pivot$ - a_i odoberá dokument, ktorý patril do α'_k , preto je a_i zahrnutá do výsledného rozdielu δ'' a zároveň premenná c je znížená o 1.
  - $a_i.before > pivot$ - a_i odoberá (vymazáva) dokument, ktorý nebol súčasťou α'_k , preto a_i je ignorovaná.

- a_i **mení dokument,** $a_i.before$ **a** $a_i.after$ **obsahujú dokument pred, respektíve po zmene**
  - $a_i.before \leq pivot \wedge a_i.after \leq pivot$ - a_i mení dokument, zmena neovplyvňuje prítomnosť v pohľade α'_k , a preto je a_i zahrnutá v nezmenenej podobe do výsledku rozdielu δ'' a premenná c tiež ostane rovnaká.
  - $a_i.before > pivot \wedge a_i.after \leq pivot$ - a_i síce mení dokument, avšak zmena spôsobila, že dokument, ktorý nepatril do α'_k by po zmene mal patriť do α'_k . Preto je a_i vyhodnotená ako pridanie dokumentu a je zahrnutá do výsledku rozdielu δ''. Premenná c je zvýšená o 1.
  - $a_i.before \leq pivot \wedge a_i.after > pivot$ - a_i síce mení dokument, avšak zmena spôsobila, že dokument, ktorý patril do α'_k by po zmene nemal patriť do α'_k . Preto je a_i vyhodnotená ako odobratie dokumentu a je zahrnutá do výsledku rozdielu δ''. Premenná c je znížená o 1.
  - $a_i.before > pivot \wedge a_i.after > pivot$ - a_i mení dokument, ktorý sa nachádzal mimo α'_k a po zmene ostal mimo α'_k , preto je a_i ignorovaná a premenná c ostane nezmenená.

Premenná c slúži ako počítadlo a po dokončení iterácie obsahuje počet dokumentov, ktoré obsahuje pohľad α'_k po aplikovaní nového rozdielu δ''. Nazvime tento pohľad α''_k. Môžu nastať tri prípady:

1. $c = l$ - počet dokumentov v α''_k je rovnaký ako limit l. V takomto prípade algoritmus končí a δ'' je odoslaná klientovi. Po aplikovaní δ'' na α'_k , pohľad $\alpha''_k = \alpha_k$, teda klientský pohľad je úspešne aktualizovaný na aktuálny pohľad servera.

2. $c > l$ - v tomto prípade algoritmus tiež skončí a postupnosť δ'' je výstupom algoritmu. Postupnosť je odoslaná klientovi, ktorý aplikuje δ'' na svoj klientský pohľad α'_k , čím dostane α''_k a nadbytočné dokumenty odstráni (klient pozná triedenie γ a limit l, takže vie určiť, ktoré dokumenty sú navyše). Po odstránení nadbytočných dokumentov opäť dostávame rovnosť $\alpha''_k = \alpha_k$. Je dôležité uvedomiť si, že server nemá dostatok informácií na to, aby povedal klientovi, ktoré dokumenty sú nadbytočné. Nevie totiž, aký bol pôvodný obsah α'_k a tým pádom na základe analýzy zmien nevie zistiť vo všeobecnom prípade, ktorý prvok je pivot (pri väčšine typov akcií sa pivot posúva). Táto práca je preto prenechaná klientovi, ktorý má k dispozícii celý obsah pohľadu, a preto pre neho nie je problém identifikovať nadbytočné dokumenty a nový pivot.

3. $c < l$ - počet dokumentov v α''_k je menší ako limit l a všetky dokumenty v klientskom pohľade sú nanajvýš tak veľké ako *pivot*. V tomto prípade nám ostáva doplniť δ'' akciami pre pridanie dokumentov, ktoré spĺňajú nasledujúce podmienky:

   - vyhovujú filtru σ
   - sú ostro väčšie ako *pivot*
   - potrebujeme ich presne $l - c$

   Dotaz, ktorý z kolekcie k vyberie dokumenty s uvedenými vlastnosťami vieme vytvoriť programaticky na základe známych informácii ako filter σ, triedenie γ, limit l a premenná c. V tejto chvíli algoritmus skončí a δ'' je odoslaná klientovi. Platí rovnosť $\alpha''_k = \alpha_k$. V tomto prípade je dôležité uvedomiť si, že klient si nevie "domyslieť" chýbajúce dokumenty, a preto mu ich server musí dopočítať. Server ich vie zistiť ľahko a vždy ich bude najviac l.

4.5 Efektívnosť synchronizačných algoritmov

Pozrime sa na uvedené dva algoritmy na výpočet rozdielu medzi pohľadmi z hľadiska časovej a pamäťovej zložitosti. Začneme algoritmom pre výpočet rozdielu medzi filtrovanými pohľadmi.

Prvý krok algoritmu potrebuje zistiť z databázy postupnosti δ_{before}, δ_{after} a $\delta_{\cup}$. Všetky tri postupnosti sa zisťujú z histórie kolekcie k, ktorá má H záznamov (akcií). Všetky tri postupnosti sa zisťujú až od verzie $v(\alpha' k)$. Atribút verzie je v histórii indexovaný, rovnako ako všetky atribúty z filtra σ, takže nájdenie postupností v databáze zaberie logaritmický čas vzhľadom na počet akcií v histórii - $O(log(H))$. Prípad môže skomplikovať filtrovanie pomocou regulárnych výrazov, kedy MongoDB robí kompletné prehladávanie kolekcie, takže sa zložitosť vyšplhá v tomto prípade na $O(H)$ pri filtroch s regulárnymi výrazmi.

Algoritmus následne iteruje celú postupnosť $\delta_{\cup}$ a v každom kroku robí len konštantné množstvo práce (zisťovanie príslušnosti v δ_{before} a δ_{after} je v $O(1)$, nakoľko v implementácii sú to množiny). Preto dostávame celkovú častovú zložitosť tohoto algoritmu lineárnu od počtu zmien v histórii kolekcie k, teda $O(H)$. Z tohoto dôvodu sa neoplatí udržiavať históriu dlhšiu ako N, kde N je počet dokumentov v kolekcii k. Pamäťová zložitosť je tiež lineárna od počtu zmien H, nakoĺko si pamätáme konštantný počet postupností o maximálnej dĺžke $O(H)$ a niekoľko pomocných premenných s konštatnou pamäťovou zložitosťou.

Pri úplnom pohľade sa situácia nezmení. Najprv použijeme algoritmus pre filtrované pohľady s časovou a pamäťovou zložitosťou $O(H)$. Jeho výstup preiterujeme a v každom kroku iterácie robíme len konštantné množstvo práce (porovnanie s pivotom je závislé len od počtu triediacich atribútov a tých je konštantný počet). Dopĺňanie dokumentov robíme len raz na záver v špecifických podmienkach a vieme ho spraviť v lineárnom čase od počtu zmien

H. Preto aj algoritmus pre výpočet rozdielu úplných pohľadov má linerárnu časovú a pamäťovú zložitosť $O(H)$, respektíve $O(N)$ ak berieme do úvahy, že väčšia história nedáva zmysel.

Sila uvedených algoritmov spočíva v dvoch faktoch:

- sú navrhnuté tak, aby pracovali len so zmenami, ktoré sa udiali od poslednej synchronizácie klienta. V praxi sa púšťajú vždy, keď sa zmenia dáta v kolekcii k a v histórii kolekcie k, takže pracujú často a na relatívne malých postupnostiach, preto je lineárna zložitosť vítaná.
- nepotrebujú si udržiavať žiadne informácie medzi jednotlivými behmi, ako v prípade *poll-and-diff* algoritmu pri frameworku Meteor

Tieto vlastnosti prispievajú k efektivite serverovej časti aplikácie čo sa týka použitých zdrojov ako operačná pamäť a procesor. Bezstavovosť serverovej aplikácie je kľúčová pre umožnenie priamočiareho horizontálneho škálovania, ktoré je možné docieliť jednoducho spustením viacerých inštancií serverovej aplikácie a použitím load balancera.

Pri návrhu algoritmov sme mysleli aj na optimalizáciu dátového toku na klienta. Algoritmy sme navrhli tak, aby posielali klientom len nutné informácie k tomu, aby klienti boli schopní dosynchronizovať svoje klientské pohľady na základe výstupu algoritmov. Preto výstup pozostáva z troch typov akcií - pridanie, odobratie a zmena dokumentu. Takáto forma výstupu je optimálna v prípade, že synchronizujeme malé množstvo zmien často. Preto sa naše algoritmy výborne hodia na použitie v single page aplikáciách, kde chceme zabezpečiť aktuálnosť dát prostredníctvom malých rýchlych aktualizácií.

Kapitola 5

Implementácia

V tejto kapitole uvedieme niektoré detaily vzorovej implementácie, ktorá je produktom tejto práce. Pozrieme sa na úlohy a zodpovednosti niektorých zaujímavých komponentov. Celú vzorovú implementáciu sme vydali ako open-source knižnicu na verejnom repozitári Github pod licenciou MIT. Knižnicu je možné nájsť na https://github.com/Igi4/sync.

5.1 Knižnica sync

Medzi ciele tejto práce patrí aj vytvorenie vzorovej implementácie dvoch algoritmov, ktoré sme uviedli v predošlej kapitole. Výstupom tejto práce je open-source knižnica sync, ktorá zastrešuje celý synchronizačný protokol a je navrhnutá tak, aby bola vhodným základným kameňom pre tvorbu single page aplikácií. Ako databázové úložisko používa MongoDB. Je vyvinutá v jazyku Dart, ktorý predstavíme v nasledujúcej časti.

5.2 Dart

Dart je open-source programovací jazyk vytvorený spoločnosťou Google. Ambíciou Googlu bolo nahradenie jazyka JavaScript ako skriptovacieho jazyka, ktorý sa používa pri tvorbe webových aplikácií na strane klienta. Dart je

schopný fungovať priamo v prehliadači prostredníctvom virtuálneho stroja, ktorý musí byť súčasťou prehliadača (napr. upravená verzia prehliadača Chromium). V prípade, že prehliadač nepodporuje Dart priamo, je možné Dart skompilovať do Javascriptu a vykonávať ho v ľubovoľnom prehliadači, ktorý podporuje JavaScript.

Dart je rovnako vhodný na programovanie časti aplikácie na strane servera. Virtuálny stroj DartVM je schopný spúšťať zdrojový kód v jazyku Dart v klasickom prostredí príkazového riadku. Dart je oproti JavaScriptu objektovo-orientovaný - podporuje triedy a dedičnosť. Podporuje tiež anotácie typov, ktoré môžu slúžiť ako dokumentácia kódu. Syntax je podobná jazyku C a Jave.

Dôvod použitia Dartu ako vývojového jazyka knižnice sync spočíva v možnosti použiť na strane klienta aj servera rovnaký jazyk. Vyhli sme sa tak duplicite kódu v prípade rovnakej funkcionality na strane klienta a na strane servera a celá knižnica je napísaná v jednom jazyku.

5.3 Štruktúra knižnice sync

Knižnica sync pozostáva z dvoch hlavných častí, klientskej a serverovej:

- **Serverová časť** knižnice zodpovedá za komunikáciu s MongoDB a ukrýva v sebe HTTP server pre komunikáciu s klientami. V zmysle *publish-subscribe* modelu sa stará o publikovanie pohľadov.

- **Klientská časť** knižnice poskytuje nástroje pre odoberanie pohľadov (subscription) a stará sa tiež o komunikáciu s HTTP serverom. Integruje knižnicu *clean_data*, ktorá poskytuje dátové štruktúry pre ukladanie obsahu klientských pohľadov.

5.4 Server

V tejto časti predstavíme niektoré dôležité triedy serverovej časti aplikácie a uvedieme ich úlohu.

- **MongoProvider** (`lib/src/mongo_provider.dart`)
  Ako názov triedy napovedá, trieda MongoProvider je zodpovedná za celú komunikáciu s databázou MongoDB. Jej rozhranie poskytuje metódy na pridanie, odobratie a zmenu dokumentu pre zápis do databázy a metódy `data` a `diffFromVersion` pre čítanie z databázy. Metóda `data` vracia aktuálny pohľad na nejakú kolekciu a metóda `diffFromVersion` je zodpovedná za výpočet rozdielu medzi klientským pohľadom a aktuálnym pohľadom. Práve metóda `diffFromVersion` v sebe ukrýva algoritmy, ktoré sme predstavili v predošlej kapitole.

- **Publisher** (`lib/src/publisher.dart`)
  Trieda Publisher funguje ako správca publikovaných pohľadov. Poskytuje metódu `publish` pre vytvorenie nového pohľadu a udržiava si zoznam publikovaných pohľadov. Premosťuje komunikáciu medzi HTTP serverom a MongoProviderom - na základe požiadavky z HTTP servera určí pohľad a zavolá správnu metódu na MongoProviderovi. Publisher je navrhnutý ako singleton, takže na celú serverovú aplikáciu existuje presne jedna inštancia.

- **Backend** (`lib/src/backend.dart`)
  Backend zabezpečuje vytvorenie HTTP serveru a správu ciest (routing). Identifikuje požiadavky pre prácu s pohľadmi a tie odovzdáva inštancii triedy Publisher. Backend zabezpečuje aj poskytovanie statického obsahu klientom (súbory HTML, CSS, klientská časť knižnice sync, obrázky a iné). Poslednou, ale nemenej dôležitou úlohou triedy

Backend, je obsluha websocketov pre notifikáciu klientov. Notifikáciu iniciuje Publisher na základe detekcie zápisovej operácie.

5.5 Klient

V tejto časti prejdeme na stranu klienta a spomenieme niektoré vybrané triedy knižnice sync:

- **Subscription** (`lib/src/subscription.dart`)
  Inštancia triedy Subscription predstavuje odoberanie vybraného klientského pohľadu v zmysle *publish* - *subscribe* modelu. Uchováva si argumenty pohľadu v prípade, že pohľad je parametrizovaný a stará sa o aktuálnosť dát v pohľade. Zabezpečuje synchronizačný protokol zo strany klienta. Klientovi poskytuje kolekciu, ktorá v sebe uchováva obsah klientského pohľadu. V prípade zmeny dát v kolekcii zo strany klienta zabezpečuje odoslanie zmien na server. Príjma notifikácie o zmene dát od triedy Publisher a iniciuje synchronizáciu pohľadu. V prípade úplného pohľadu si udržuje informácie o hraničnom dokument (pivot), ktorý je potrebný k synchronizácii.

- **Subscriber** (`lib/src/subscriber.dart`)
  Úlohou triedy Subscriber je poskytovať rozhranie pre vytváranie nových inštancií triedy Subscription - na tento účel slúži metóda `subscribe`.

- **Connection** (`lib/src/connection.dart`)
  Trieda Connection sa stará o celú komunikáciu so serverom. Poskytuje možnosť odosielať HTTP požiadavky, rieši serializáciu dát a komunikáciu cez websockety na strane klienta.

Záver

Cieľom tejto práce bolo navrhnúť a implementovať protokol pre synchronizáciu klientských pohľadov medzi serverom a klientom s dôrazom na veľkosť toku dát a s ohľadom na veľký počet klientov. Súčasťou práce bolo vytvorenie vzorovej implementácie takéhoto návrhu. Cieľ tejto práce považujeme za splnený vo všetkých bodoch, nakoľko knižnica sync v súčasnej podobe poskytuje funkčné riešenie pre tvorbu single page aplikácií a odbremeňuje vývojárov od nutnosti riešiť komunikáciu so serverom a problém synchronizácie dát. Knižnica sync poskytuje jednoduché a intuitívne rozhranie pre publikovanie pohľadov a ich odoberanie na strane klienta. Klient pracuje len s lokálnou kolekciou (klientským pohľadom) a o aktuálnosť dát sa stará knižnica sync.

V dôsledku návrhu a efektívnosti algoritmov, ktoré sme predstavili v štvrtej kapitole tejto práce je dosiahnutá optimálna komunikácia medzi klientom a serverom, nakoľko sa posielajú len najnutnejšie informácie, z ktorých je klient schopný zrekonštruovať aktuálny pohľad. Serverová časť aplikácie je navrhnutá tak, aby si nepotrebovala uchovávať stav a bolo ju možné púšťať v ľubovoľnom počte inštancií bez koordinátora, čím vieme zabezpečiť obhospodárenie ľubovoľného počtu klientov pomocou priamočiareho horizontálneho škálovania.

Knižnicu sync sme vydali ako open-source pod licenciou MIT a môže byť rozšírená na plnohodnotný framework pre stavbu single page aplikácií, avšak takéto rozšírenie ďaleko presahuje rozsah tejto práce. Na záver uvedieme niekoľko hlavných funkcií, ktoré je potrebné vyriešiť pre produkčné nasadenie

tejto knižnice. Je potrebné myslieť na obnovu spojenia v prípade, že klient stratí spojenie so serverom a treba vyriešiť problém opravenia synchronizácie, ak spojenie padlo počas synchronizácie. Praktický výkon serverovej aplikácie je možné zvýšiť technikou vyrovnávacej pamäte (cache), ktorá by si uchovávala vypočítané rozdiely pohľadov v prípade, že by sa veľké množstvo klientov chcelo dosynchronizovať z rovnakej verzie pre rovnaký pohľad. Je možné integrovat do knižnice sync offline úložisko, nakoľko moderné prehliadače poskytujú rôzne formy perzistentných lokálnych databáz. Pomocou offline úložiska by sme vedeli tvoriť single page aplikácie, ktoré by dokázali pracovať plnohodnotne v offline režime a synchronizovali by sa v prípade dostupnosti internetového spojenia.

Literatúra

[Car91] Robertson G. G. Mackinlay J. D. Card, S. K. The information visualizer: An information workspace. *Proc. ACM CHI'91 Conf. (New Orleans, LA, 28 April-2 May)*, pages 181–188, 1991.

[Chr] Christof Strauch. NoSQL Databases. http://www.christof-strauch.de/nosqldbs.pdf. [Online; accessed 3-May-2014].

[Dat] Datomic. Datomic documentation. http://docs.datomic.com/. [Online, accessed 10-Mar-2015].

[Dav] David Glasser. David Glasser on scaling Meteor with the MongoDB oplog. https://www.meteor.com/blog/2013/12/18/david-glasser-on-scaling-meteor-with-the-mongodb-oplog. [Online, accessed 29-Jan-2015].

[Fir] Firebase. Firebase documentation. https://www.firebase.com/docs/. [Online, accessed 5-Feb-2015].

[Ing] Ingo Maier, Martin Odersky. Deprecating the Observer Pattern with Scala.React. http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf. [Online, accessed 28-Jan-2015].

[Met] Meteor. Meteor Documentation. http://docs.meteor.com/#/full/. [Online, accessed 28-Jan-2015].

[Mon] MongoDB. MongoDB Manual. http://docs.mongodb.org/manual/. [Online, accessed 3-May-2014].

[Nod] Node.js - Wikipedia. Node.js. http://en.wikipedia.org/wiki/Node.js. [Online, accessed 28-Jan-2015].

[Sin] Single page application - Wikipedia. Single page application. http://en.wikipedia.org/wiki/Single-page_application. [Online, accessed 2-Feb-2015].

[Tim] Tim Berners-Lee - Wikipedia. Tim Berners-Lee. http://en.wikipedia.org/wiki/Tim_Berners-Lee. [Online, accessed 28-Jan-2015].

[Web] Web page - Wikipedia. Web page. http://en.wikipedia.org/wiki/Web_page. [Online, accessed 27-Jan-2015].

[Wor] World Wide Web - Wikipedia. World Wide Web. http://en.wikipedia.org/wiki/World_Wide_Web. [Online, accessed 28-Jan-2015].