UNIVERZITA PAVLA JOZEFA ŠAFÁRIKA V KOŠICIACH
PRÍRODOVEDECKÁ FAKULTA

NOSQL DATABÁZY A PODPORA OFFLINE REŽIMU

2017 **Bc. Jozef DŽAMA**

UNIVERZITA PAVLA JOZEFA ŠAFÁRIKA V KOŠICIACH
PRÍRODOVEDECKÁ FAKULTA

NOSQL DATABÁZY A PODPORA OFFLINE REŽIMU

DIPLOMOVÁ PRÁCA

Študijný program:	Informatika
Pracovisko (katedra/ústav):	Ústav informatiky
Vedúci diplomovej práce:	RNDr. František Galčík, PhD

Košice 2017

Bc. Jozef DŽAMA

Univerzita P. J. Šafárika v Košiciach
Prírodovedecká fakulta

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jozef Džama
Študijný program: Informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: Diplomová práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: NoSQL databázy a podpora offline režimu.

Názov EN: NoSQL databases and offline mode.

Cieľ: (1) Preskúmať a analyzovať existujúce databázové (SQL a NoSQL) riešenia (s dôrazom na mobilné platformy) a porovnať ich z hľadiska výkonnosti, použiteľnosti API a podpory pre synchronizáciu a offline režim.
(2) Preskúmať a analyzovať obvyklé spôsoby implementácie offline režimu a cachovania pri prístupe k údajom vo vzdialených dátových úložiskách.
(3) Navrhnúť, implementovať a overiť prototyp riešenia pre podporu transparentného offline režimu a cachovania údajov zo vzdialeného dátového úložiska.

Literatúra:
[1] Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C., & Tomasic, A. (2008). Scalable query result caching for web applications. Proceedings of the VLDB Endowment, 1(1), 550-561.
[2] Benson, E., Marcus, A., Karger, D., & Madden, S. (2010, April). Sync kit: a persistent client-side database caching toolkit for data intensive websites. In Proceedings of the 19th international conference on World wide web (pp. 121-130). ACM.
[3] Yap, J. (2014). Transparent Consistency in Cache Augmented Database Management Systems

Vedúci: RNDr. František Galčík, PhD.
Oponent: RNDr. Peter Gurský, PhD.
Ústav : ÚINF - Ústav informatiky
Riaditeľ ústavu: prof. RNDr. Viliam Geffert, DrSc.

Dátum schválenia: 10.04.2017

prof. RNDr. Viliam Geffert, DrSc.
riaditeľ ústavu

Univerzita Pavla Jozefa Šafárika v Košiciach
Prírodovedecká fakulta
Ústav informatiky

Poďakovanie

Moja vďaka patrí predovšetkým RNDr. Františkovi Galčíkovi, PhD, vedúcemu tejto práce, za všetku ochotu, trpezlivosť, cenné rady a nápady.

Abstrakt

V našej práci sme sa venovali porovnaniu databázových systémov a analyzovaniu rôznych spôsobov riešenia offline režimu a cachovania. Na základe tejto analýzy sme navrhli a implementovali riešenie pre podporu transparentného offline režimu a cachovania údajov zo vzdialeného dátového úložiska. Vytvorené riešenie je založené na princípe rozširovateľnosti. Na otestovanie nami vytvoreného systému sme implementovali lokálne úložisko používajúce databázu SQLite a vzdialené úložisko komunikujúce pomocou DDP protokolu.

Abstract

In our thesis, we worked on comparing database systems and analyzing various ways of resolving offline mode and caching. Based on this analysis, we designed and implemented a solution to support transparent offline mode and data caching from remote data storage. The solution created is based on the principles of extensibility. To test the system we created, we implemented a local storage using the SQLite database and a remote storage communicating through the DDP protocol.

Obsah

Úvod

„Sieť nie je k dispozícií“. Jedna z neobľúbených správ zobrazovaných na obrazovke laptopu alebo mobilného zariadenia.

Byť „online“ sa v poslednej dobe stáva samozrejmosťou, až nutnosťou. Sieť internet ponúka stále viac služieb, pribúda webových aplikácií. Okrem pevného pripojenia alebo wifi prístupových bodov, sa rozširuje aj pokrytie mobilnou sieťou a rýchlosť tejto siete.

Napriek tomu vo Facebooku pred istým časom zaviedli takzvané utorky pomalého pripojenia. Vývojárom to má simulovať situáciu, v ktorej sa nachádzajú mnohí ich používatelia z rozvojových krajín. Aj na Slovensku sú oblasti, kde je pripojenie pomalšie ako by si používatelia želali. Taktiež situácie, kde je pripájať sa k sieti zakázane, napríklad v niektorých nemocniciach alebo na palube lietadla.

Spracovanie dát v 21. storočí prináša tiež nové výzvy – pribúda veľké množstvo (neštruktúrovaných) údajov (takzvané BigData). Okrem klasických relačných databáz sa začínajú používať aj iné, neštandardné typy spoločne označované ako NoSQL.

V našej práci chceme porovnať rôzne databázové systémy, ako pristupujú k synchronizácii, či podporujú offline režim a ak áno, tak ako. Tiež nás zaujíma aké riešenia boli vytvorené na synchronizáciu údajov cez sieť a aké stratégie sa pri tom používajú.

Následne si popíšeme ako sme navrhli a implementovali riešenie pre podporu offline režimu a cachovania údajov zo vzdialeného dátového úložiska. Naše riešenie je inšpirované modelom dokumentových databáz, nie je však zamerané na konkrétne úložisko dát u klienta, konkrétny protokol na komunikáciu so serverom, či serverové riešenie. Vytvorené riešenie je založené na princípoch rozširovateľnosti a skladá sa z 3 častí: hlavného úložiska starajúceho sa o základnú logiku, modulárneho lokálneho úložiska s databázou a modulárneho vzdialeného úložiska komunikujúceho so serverom.

Vývojárov často nezaujímajú podrobnosti synchronizácie; od úložiska zvyčajne očakávajú, že doň môžu uložiť dáta a potrebné dáta z neho získať. Naše riešenie preto vykonáva synchronizáciu lokálnych údajov so serverom. Počas chýbajúceho pripojenia k serveru možno vykonávať operácie lokálne a so serverom sa dáta zosynchronizujú po obnovení pripojenia. Výsledkom je transparentný offline režim.

Napokon sme implementovali lokálne úložisko pracujúce s databázou SQLite a vzdialené úložisko komunikujúce cez DDP protokol a otestovali sme naše riešenie na jednoduchom príklade.

V prvej kapitole tejto práce vysvetľujeme základné pojmy a uvádzame našu motiváciu. Druhá a tretia kapitola sa venujú iným riešeniam. V štvrtej kapitole popisujeme návrh a v piatej implementáciu nášho riešenia. Šiesta kapitola obsahuje návod a príklad použitia. V siedmej kapitole sme zhodnotili našu prácu a uviedli možné vylepšenia.

1 Motivácia a východiská

1.1 Offline režim aplikácií

O offline režime hovoríme v kontexte klient–server architektúry. Tá je charakteristická tým, že sa klient zvyčajne cez sieť internet pripája k serveru, či už vo forme dlhšie trvajúceho spojenia alebo na krátky čas, počas ktorého sa zrealizuje odoslanie požiadavky klienta na server, jej spracovanie serverom a následne zaslanie odpovede späť klientovi. Offline režimom myslíme schopnosť klientskej aplikácie pracovať aj bez pripojenia k serveru (či už s plnou alebo aspoň obmedzenou funkcionalitou). Dôvodom prerušenia komunikácie so serverom je najčastejšie chýbajúce pripojenie k sieti internet, ale problémom môže byť aj nefunkčnosť servera, napríklad kvôli výpadku.

Napriek stále sa rozširujúcemu pokrytiu sieťou internet, či už pomocou mobilných sietí alebo wifi prístupových bodov, pribúda aj aplikácií, ktoré ponúkajú offline režim. Hovorí sa o iniciatíve „offline first“ a vyvíjaní takzvaných „občasne pripojených“ aplikácií. Spomedzi mnohých príkladov možno spomenúť aplikácie s poznámkami/TODO úlohami, ako Evernote, ale aj sociálne siete, napríklad Facebook, tiež služby ako Spotify (ponúkajúca hudbu), Gmail aplikácia a ďalšie.

Offline režim sa však netýka iba smartfónov. Väčšina moderných webových prehliadačov tiež ponúka nejakú formu offline režimu. Rovnako spomenuté služby – Spotify a Gmail – oba ponúkajú aj desktop aplikáciu, ktorá funguje aj v offline režime. Dôvodom môže byť, že ľudia si svoje laptopy stále častejšie berú všade so sebou.

Okrem spomenutých typov aplikácií, možno sa zaoberať aj prípadom pracovníka, ktorý potrebuje mať laptop so sebou v teréne, kde môže byť rýchlosť pripojenia k internetu veľmi pomalá, prípadne dokonca žiadna. Ak v takomto prípade potrebuje pracovať s aplikáciou, nejaká forma offline režimu môže byť veľmi nápomocná.

Ozývajú sa aj hlasy proti offline režimu. Zvyčajne ide o názor, že offline režim je proti internetu, ľudia chcú byť predsa stále pripojení, nie offline. A s rozvojom pokrytia sieťou internet čoskoro nebude potrebný. V praxi sa však ukazuje, že opak je pravdou - popularita offline režimu rastie s popularitou (a teda aj rozvojom) internetu. Stále sa totiž nachádzajú situácie, krajiny alebo miesta, kde je pripojenie buď pomalé alebo chýba úplne.

1.2 Komunikačné siete s vysokou latenciou

Potrebujeme si tiež uvedomiť, že pre prácu v offline režime musíme mať na offline zariadení údaje, ktoré na takúto prácu potrebujeme. Táto vlastnosť môže byť výrazne nápomocná aj v prípade, že sa nachádzame v sieti s vysokou latenciou, príkladom sú GPRS siete.

Namiesto čakania na veľa údajov od servera, totiž už môžu všetky potrebné dáta byť na klientovi. Aplikácia teda nemusí čakať na to, aby jej server dáta posielal, čo v takomto prípade môže nejakú dobu trvať, ale môže hneď pracovať. Takto vieme získať rýchly offline režim, pričom v prípade potreby môžeme dáta synchronizovať v sieti s vysokou latenciou. Navyše, ak sa na serveri zmenili dáta, prípadne sa zmena stane počas používania aplikácie, server môže klientovi poslať iba danú zmenu. Takýmto zredukovaním množstva komunikácie cez sieť a tým, že údaje vieme udržiavať stále aktuálne skrze synchronizáciu, ktorá prebieha na pozadí, sa dá pozitívne ovplyvniť rýchlosť aplikácie. To vo výraznej miere zlepšuje aj skúsenosť používateľa, ktorý s ňou pracuje.

1.3 Databázy ako úložiská údajov aplikácii

Väčšina aplikácií pracuje aj s údajmi, ktoré si potrebuje ukladať perzistentne. Častokrát ide naviac o veľa dát, chceme ich ukladať a upravovať efektívne, prípadne chceme aj kontrolovať ich správnosť alebo ich mať uložené bezpečne, či už z pohľadu prístupu alebo zotavenia sa z pádov systému. Na to sa najčastejšie využívajú databázové systémy. Databáza je organizovaná množina dát. O jej definíciu, vytvorenie, úpravy, čítanie a administráciu sa stará systém pre správu databázy.

Z pohľadu na vzťah databázy a aplikácie, ktorá ju používa, môžeme hovoriť o databázach, ktoré sú súčasťou aplikácie (takzvaných vstavaných databázach) a o vzdialených/autonómnych databázach (typu klient-server), ku ktorým sa aplikácie musia pripájať.

Výhodou prvého prístupu je relatívna jednoduchosť použitia, koncový používateľ nemusí mať nainštalovaný databázový systém, samotná databáza vyžaduje minimálnu až žiadnu údržbu. Slúži ako jednoduché a rýchle úložisko údajov. Hodí sa pre mobilné aplikácie alebo všeobecne aplikácie, kde nepotrebujeme ukladať príliš veľa dát a stačí nám ich mať uložené lokálne. Navyše pri tomto prístupe nie je typicky potrebné zdieľať dáta s inými používateľmi, keďže každá aplikácia má vlastnú databázu údajov.

Oproti tomu databázové riešenia typu klient-server sú zvyčajne robustnejšie, výkonnejšie a ponúkajú viacero možností na prácu, no za cenu náročnej údržby a istej nemotornosti. Naviac, ak sa klientska a serverovská časť nachádza na rôznych zariadeniach, výpadky spojenia sú problémom, keďže sa môže stať, že stratíme prístup k dátam.

Toto rozdelenie veľmi úzko súvisí s offline režimom, keďže väčšina klient-server aplikácii využívajúcich internet, si ukladá údaje na serveri. Server sa teda často stáva akousi databázou zhromažďujúcou dáta a poskytujúcou ich svojim používateľom. Obsahuje zvyčajne veľké objemy dát a potrebuje s nimi efektívne pracovať, na čo sú vhodné autonómne (klient-server) databázy. Na druhej strane, konkrétna klientska aplikácia takmer vždy využíva len určitú podmnožinu týchto dát (napríklad v TODO aplikácii ma zaujímajú len moje úlohy, nie všetkých používateľov aplikácie), no ak chce pracovať aj offline, musí si ich niekde ukladať. Tu aplikácii stačí jednoduchá databáza, ktorá bude jej súčasťou.

Iné delenie databáz je delenie podľa spôsobu práce s dátami. Zároveň to odzrkadľuje isté historické etapy ich vývoja. Nezaoberajúc sa diernymi štítkami, ukladaním dát v súborovom systéme alebo historicky prvými modelmi (hierarchickým a sieťovým), ale databázovými systémami, aké sa dnes používajú, ostávajú nám dve veľké skupiny a to relačné a takzvané NoSQL databázy. (1)

1.3.1 Relačné databázy

Relačné databázy sú principiálne postavené na relačnom modeli. Hovorí sa im aj SQL databázy, pretože databázové systémy, ktoré ich spravujú, využívajú jazyk SQL. Tieto databázy ukladajú dáta do tabuliek. Tabuľka reprezentuje určitý celok dát, istú entitu a skladá sa z riadkov a stĺpcov. Stĺpce predstavujú atribúty dát, v riadkoch sú uložené samotné dáta (záznamy, n-tice).

Tieto databázy vykonávajú úpravy pomocou transakcií, ktoré zvyčajne striktne dodržiavajú vlastnosti ACID – atomicita, konzistentnosť, nezávislosť a trvanlivosť.

Relačné databázy sú dnes pravdepodobne najrozšírenejšie spomedzi databáz. Medzi najznámejšie z nich patrí databázový server MySQL alebo databázový systém SQLite, ktorý sa naopak nepoužíva ako vzdialená, ale ako vstavaná databáza.

1.3.2 NoSQL databázy

Relačný model databáz stojí na matematických základoch, konkrétne z teórie množín a zaoberá sa len ukladaním dát. S nástupom objektovo-orientovaného programovania, ktoré je oproti relačnému modelu postavené na princípoch softvérového inžinierstva a pracuje s objektami, ktoré majú dáta a správanie, sa začína hovoriť o komplikáciách pri ich súčasnom použití. Tieto komplikácie sa zvyknú označovať názvom objektovo-relačný nesúhlas impedancie (object-relational impedance mismatch), termínom prevzatým z elektro-inžinierstva. (2) Ako reakcia na tieto nepríjemnosti vznikli objektové databázy, kde sú dáta reprezentované formou objektov.

21. storočie sa okrem iného sa vyznačuje aj obrovským nárastom objemu údajov. V roku 2013 sa odhadovalo, že 90% zo všetkých dát na svete vzniklo počas posledných 2 rokov. (3) Hľadajú sa nové modely, ktoré by zvládali stále väčšie množstvá dát, dali sa lepšie škálovať (najmä vzhľadom na počet pracovných staníc), prípadne dokázali pracovať aj s neštruktúrovanými dátami. Relačné databázy sa dokážu s týmito problémami vysporiadať, ale často je to zložité, pretože väčšina nebola navrhnutá s dôrazom na riešenie týchto problémov. Vplyvom tejto situácie začínajú vznikať databázy iných typov, spoločne označované ako NoSQL.

Okrem rozdielneho pohľadu na dáta oproti relačným databázam, viaceré NoSQL riešenia upúšťajú od dôrazu na ACID, konkrétne najmä prísnej konzistentnosti, za účelom získania lepšej dostupnosti k dátam. A to aj v prípade, že databázy sa nachádzajú na viacerých serveroch a niektorý z nich nie je dostupný. Tomuto modelu sa hovorí model eventuálnej konzistentnosti. Takýto systém konverguje ku konzistentnému stavu, ale pokým v konzistentnom stave nie je, čítanie môže vrátiť rôzne hodnoty. Špeciálnym prípadom je model silnej eventuálnej konzistentnosti, ktorý naviac garantuje, že uzly, ktoré prijali rovnakú (neusporiadanú) množinu úprav, budú po ich aplikovaní v rovnakom stave. Takýto systém je monotónny (nepotrebuje sa vracať k staršiemu stavu).

Z tohto príkladu môže byť vidno, že NoSQL databázy vo všeobecnosti nechcú nahradiť relačné, ale zameriavajú sa aj na iné prípady použitia. Tak treba chápať aj názov NoSQL – z anglického „not only SQL“ (nie len SQL).

Zameriavajúc sa na rôzne problémy, NoSQL databázy môžeme podľa ich pohľadu na dáta rozdeliť do niekoľkých veľkých skupín. Najväčšie z nich sú:

- Kľúč-hodnota (key-value) – jeden z najjednoduchších modelov, často tvoria základ pre ďalšie typy. Princípom je hash tabuľka s kľúčom a ukazovateľom na dáta. Príkladom je Redis.
- Stĺpcovo orientované – podobné prvému typu, ale zamerané na veľa dát rozdelených na viacerých strojoch. Kľúče obsahujú ukazovatele na viacero stĺpcov. Známe a často používané riešenia sú napríklad Cassandra a HBase.
- Dokumentové – dáta uchovávajú v dokumentoch. Dokumenty obsahujú dáta v nejakom štandardnom formáte (JSON, XML, BSON) a sú adresované pomocou unikátnych kľúčov. Tieto databázy taktiež ponúkajú API alebo dopytovací jazyk na prácu s dokumentmi. Príklady: Couchbase, MongoDB.
- Grafové – určené pre dáta, ktorých vzťahy sa dajú popísať grafom. Pozostávajú z vrcholov (predstavujúcich entity), hrán (tvoriacich väzby) a ich vlastností. Príkladom môže byť AllegroGraph alebo OrientDB.

1.4 Offline režim z pohľadu vývojára

Okrem úložiska pre údaje, klient-server aplikácia musí nejakým spôsobom riešiť aj synchronizáciu dát medzi klientom a serverom. Dá sa to riešiť rôznymi spôsobmi, podľa toho, kto sa stará o samotnú synchronizáciu môžeme hovoriť o 3 skupinách.

Synchronizáciu môže riešiť samotný databázový systém. Výhodou je jednoduchosť použitia, synchronizácia sa tak stáva pre vývojára prakticky neviditeľná. Na druhej strane, nevýhodou je, že autor aplikácie stráca možnosť prispôsobenia niektorých vecí, môžu to byť napríklad serverové úložisko alebo spôsob prenosu dát cez sieť.

Druhou možnosťou ako riešiť synchronizáciu je použitie knižnice. Príklady môžeme nájsť u mobilných zariadení. Napríklad Android ponúka na túto prácu Sync adapter framework. Nevýhodou tohto prístupu môže byť, že väčšina knižníc podporuje len istú množinu databáz u klienta a len určitý protokol na komunikáciu so serverom. Zvyčajne sa však dá nájsť knižnica podporujúca kombináciu žiadúcich technológií.

Okrem spomenutých možností, kde sa použije nejaké hotové riešenie alebo kombinácia technológií, vývojár si môže synchronizáciu vyriešiť aj sám, na mieru pre svoju aplikáciu. Nevýhodou je v tomto prípade najmä množstvo práce a s tým súvisiaca časová náročnosť, ale takto prispôsobené riešenie môže mať rôzne výhody, ako

napríklad lepšiu efektívnosť, prípadne sa môže stať, že to je jediné možné riešenie pre danú úlohu.

2 Databázy a podpora offline režimu

U rôznych typov databáz sa podpora pre offline režim, či vhodnosť ich vlastností pre použitie pri riešení offline režimu líšia.

2.1 Relačné databázy

Relačné databázy sa vyvíjali dlhý čas. Pôvodne sa nezameriavali na synchronizáciu medzi viacerými replikami. Predpokladaný model využitia bol spočiatku tvorený serverom s databázou, ku ktorému sa klienti museli pripojiť a potom mohli pracovať s dátami. S nástupom internetu sa z viacerých dôvodov (najmä kvôli veľkému objemu dát a kvôli dostupnosti) databázové systémy začali meniť. Dnešný model je často tvorený viacerými databázovými servermi, ktoré si dáta podelené na časti rozdeľujú medzi seba (fragmentácia) alebo kopírujú takým spôsobom, že sa údaje nachádzajú na viacerých serveroch (replikácia). To súvisí s offline režimom, ktorého podstatou je, že klient je istá replikácia podmnožiny dát zo servera. Vo všeobecnosti sú však u relačných databáz synchronizácia a offline režim podporované skôr na aplikačnej úrovni (viac v kapitole 3) ako priamo u databáz.

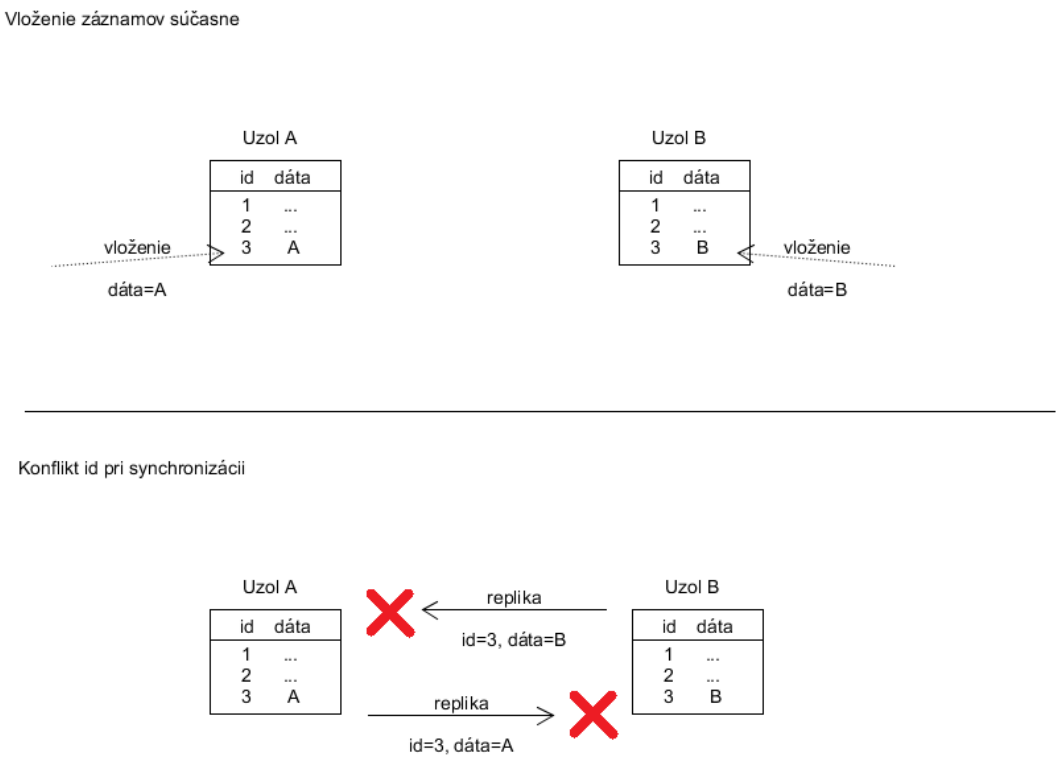
2.1.1 MySQL

MySQL je relačný databázový server. Nemá priamu podporu pre offline režim, ale podporuje replikáciu, ktorá zabezpečuje, že uzol sa dá do istej miery použiť aj bez pripojenia k zvyšku systému.

U MySQL replikácia funguje spôsobom master-slave, teda je tvorená nadriadeným uzlom (master) a replikami, ktoré držia aktuálnu kópiu dát nadriadeného uzla (slave). Master uzol zapisuje informácie o zmenách do takzvaného binary logu. Ak sa pripojí podriadený uzol, master vytvorí nové vlákno pre to pripojenie a môžu komunikovať - zvyčajne ide o preposielanie zmien zo spomenutého binary logu. Ak pripojený slave uzol má aktuálne dáta, posielajú sa len nové zmeny. Na podriadenom uzle sa v jednom vlákne čítajú dáta od nadriadeného uzla a zapisujú sa do lokálneho log súboru (nazývaného relay log) a v druhom vlákne sa číta tento relay log a zmeny sa aplikujú na príslušnú lokálnu databázu.

Toto riešenie je však limitované jednosmernou synchronizáciou. Zápisy u replík sú síce povolené, ale nebudú synchronizované s nadriadeným uzlom, ani s ostatnými replikami. Repliky sa dajú použiť ako offline databáza, avšak v takejto podobe len na čítanie. Pre získanie obojstrannej synchronizácie musíme tento model doplniť pridaním

ďalšej synchronizačnej vrstvy, v ktorej bude nadriadeným uzlom ten, ktorého zápisy chceme synchronizovať a ostatné budú podriadené. Týmto spôsobom však získame viacero uzlov, kde môžeme zapisovať dáta. Predpokladajme model s 2 master uzlami. Ak vložíme záznam do oboch master uzlov do rovnakej tabuľky naraz, oba uzly priradia záznamu rovnaké ID a pokúsia sa poslať tento záznam podriadeným uzlom, čo znamená, že oba uzly pošlú záznam s rovnakým ID tomu druhému uzlu, to je však problém, keďže takto nastane konflikt jedinečnosti ID (viď Obr. 1). Tento problém sa u MySQL rieši rozdelením čísel 1 až počet master uzlov medzi tieto uzly a následným zvyšovaním ID o počet master uzlov na každom takomto uzle. V prípade *n* uzlov bude platiť, že ID(mod *n*) bude na vybranom uzle konštantné. V spomenutom prípade 2 nadriadených uzlov by jeden z nich generoval ako identifikátory párne a druhý nepárne čísla.



Obr. 1 Neriešený konflikt ID pri replikácii typu master-master u MySQL databázy.

Podobné prístupy podporujú aj ďalšie veľké databázové systémy ako Microsoft SQL Server, PostgreSQL a Oracle databázový systém.

2.1.2 SQLite

SQLite je v súčasnej dobe pravdepodobne najpoužívanejší vstavaný databázový systém. Používa sa ako natívna databáza na platforme Android. SQLite je relačná

databáza, používajúca sa ako súčasť aplikácií. Oproti klient-server databázovým systémom je cieľom SQLite poskytnúť jednoduché, nezávislé a efektívne úložisko údajov pre konkrétnu aplikáciu alebo zariadenie. Nemá zabudovanú podporu pre synchronizáciu s inými databázami, ani pre offline režim (riešiť sa to dá inými knižnicami, viď Zumero v kapitole 3.2.2), ale svojou povahou je vhodná na použitie na mobilných zariadeniach alebo ako súčasť rôznych aplikácií. Často sa používa ako perzistentné úložisko na klientovi pri implementovaní offline režimu.

2.2 NoSQL databázy

Napriek tomu, že NoSQL databázy sú oproti relačným podstatne mladšie, respektíve sa do ich vývoja zatiaľ investovalo menej času oproti relačným, boli vytvorené s ohľadom na škálovanie na viacero serverov. Viaceré z NoSQL databáz majú aj priamu podporu pre offline režim.

2.2.1 CouchDB

CouchDB je dokumentová NoSQL databáza. Dáta ukladá v dokumentoch typu JSON. Podporuje škálovanie, replikáciu s obojsmernou synchronizáciou, pričom garantuje eventuálnu konzistentnosť.

Protokol, ktorý sa stará o synchronizáciu, bol v CouchDB navrhnutý aj s dôrazom na fungovanie v stave, kedy sú niektoré uzly offline. Tento protokol funguje cez http protokol používajúc CouchDB REST API. Stal sa základom pre generáciu takzvaných „Offline first“ aplikácií, ktoré kladú dôraz na offline režim. Ďalšie systémy, ktoré sú kompatibilné s CouchDB protokolom pre replikáciu sú PouchDB, Cloudant a Couchbase.

Pre pochopenie synchronizácie a riešenia konfliktov si treba povedať, ako vyzerá dokument v CouchDB. Dokument obsahuje okrem svojho ID aj aktuálnu verziu a všetky staré verzie zmien (ak nastal konflikt, história verzií vytvorí strom), samotné dáta a prípadne ďalšie príznaky, napríklad či bol dokument zmazaný.

Pri použitý na jednom uzle sa CouchDB vyhýba konfliktom použitím spomenutej verzie dokumentov. Dokument môže byť upravený len tým používateľom, ktorý poskytne aj verziu zhodnú s tou na serveri. To znamená, že ak dvaja používatelia získali rovnaký dokument zo servera, upraviť ho môže len ten, koho požiadavka príde na server prvá. Ďalšie požiadavky nebudú mať správne číslo verzie, keďže tá sa medzitým

zmenila. Títo používatelia musia najprv požiadať o aktuálny dokument a až potom ho upravený poslať na server.

Na viacerých uzloch je situácia zložitejšia. Pri replikácii typu master-master sa môže stať, že viaceré servery prijali úpravu toho istého dokumentu. Pri následnej replikácii tohto dokumentu na ostatné uzly nastane konflikt. Ten CouchDB rieši len tak, že vyberie na všetkých uzloch niektorú (všade tú istú, vyberá podľa dĺžky histórie, prípadne lexikograficky) verziu dokumentu za víťaza a ostatné označí ako konflikt. Používateľ si môže následne vyhľadať konflikty, zobraziť históriu daného dokumentu a ľubovoľne vyriešiť nájdené konflikty.

CouchDB podporuje offline režim, ale nevýhodou tohto riešenia môže byť v istých prípadoch http protokol, ktorý nepatrí medzi najefektívnejšie protokoly a potreba riešenia konfliktov. Tým, že konflikty musí riešiť používateľ, synchronizácia nie je úplne transparentná. Na druhej strane, ak sa konflikty neriešia, veľkosť dát zbytočne narastá kvôli nepoužitým verziám dokumentov. Tieto problémy zdieľajú aj ďalšie riešenia odvodené z CouchDB.

2.2.2 PouchDB

PouchDB je tiež dokumentová databáza. Jej cieľom je byť čím podobnejšia CouchDB, s tým rozdielom, že je napísaná v jazyku Javascript a má slúžiť ako databáza pre prehliadače.

PouchDB je zameraná na klienta v klient-server modeli, ponúkajúc lokálne úložisko aj v čase bez pripojenia na server. Používajúc synchronizačný protokol CouchDB, PouchDB sa po pripojení na server zosynchronizuje či už s CouchDB alebo ostatnými kompatibilnými databázami použitými v danom systéme.

2.2.3 Couchbase

Couchbase systém ponúka viacero softvérových riešení, pre nás sú najzaujímavejšie Couchbase Server, Sync Gateway a Couchbase Lite, ktoré spolu zabezpečujú plnú podporu pre offline režim. Okrem toho Couchbase ponúka podporu pre viacero ďalších technológií.

Couchbase Server je NoSQL databáza, používajúca buď dokumentový model alebo model kľúč-hodnota. Tento databázový systém je navrhnutý ako distribuovaný systém zameraný na prácu s veľa dátami, škálovanie a dostupnosť.

Couchbase Lite je dokumentová NoSQL databáza, používaná ako súčasť aplikácie, zameraná na mobilné zariadenia. Slúži ako trvalé úložisko, odkiaľ môžu aplikácie získavať dáta, aj v čase bez pripojenia k serveru.

Sync Gateway je synchronizačný nástroj, ktorý spája Couchbase Lite a Couchbase Server. Serverová časť Sync Gateway funguje ako replikačný uzol. Počúva na http požiadavky a ukladá dáta do Couchbase Server databázy. Stará sa o to, aby klienti mali aktuálne dokumenty a z klienta posiela lokálne zmeny na server. Naviac používa takzvané kanály, ktoré zabezpečujú, že každý klient dostane preňho relevantné a prístupné dáta. Kanály tvoria prepojenie medzi užívateľmi a dokumentmi. Každý dokument patrí do istej množiny kanálov a každý používateľ má prístup k nejakej množine kanálov. Z pohľadu dokumentu sú kanály jeho tagom. Užívateľ má zas prístup k niektorým kanálom a pri požiadavke o dáta vie špecifikovať, ktoré kanály ho zaujímajú. Konflikty sa riešia rovnako ako v CouchDB.

Silnou, ale aj zraniteľnou stránkou Couchbase sú dopyty. Dopyty sa v Couchbase dajú zadávať aj pomocou nimi vytvoreného jazyka N1QL, ktorý sa používa takmer identicky ako SQL. Rýchlosť vyhodnotenia dopytov je však u Couchbase výrazne ovplyvnená použitím view (databázovými pohľadmi), ktoré tu slúžia ako indexy. Dopyty nad dátami vo view sú rýchle, ale aj view majú svoje limity a nemusia stačiť pre komplexné dopyty.

2.2.4 MongoDB

MongoDB je dokumentový databázový systém typu klient-server. Podporuje replikáciu aj fragmentáciu, map-reduce, agregáciu a indexy pre rýchle vyhľadávanie dát. Údaje ukladá do dokumentov podobných JSONu s dodatočnými dátovými typmi, nazývaných BSON (Binary JSON).

Replikáciu MongoDB rieši do istej miery podobne ako MySQL (viď kapitola 2.1.1). Taktiež používa log (nazývaný oplog), z ktorého kopíruje zmeny na ostatné uzly, kde sa zmeny aplikujú do databázy.

MongoDB databáza hovorí aj o podpore pre mobilné aplikácie, ponúka však len serverové úložisko pre údaje. Nerieši priamo problémy offline režimu, ani neponúka jednoduchšiu databázu pre mobilné zariadenia.

2.2.5 Realm

Realm bola pôvodne len objektová databáza, alternatíva k SQLite pre mobilné zariadenia. Okrem mobilnej databázy bol však koncom roka 2016 pridaný aj synchronizačný server s objektovou databázou ako perzistentným úložiskom. Mobilná databáza je vhodná aj samotná pre použitie ako úložisko dát pre aplikácie, ktoré budú fungovať aj v offline režime, ale pri použití s Realm serverom sa získa zadarmo aj obojsmerná synchronizácia. V čase bez pripojenia sa použijú dáta z mobilnej databázy, po navrátení spojenia sa dáta automatický zosynchronizujú.

Konflikty rieši Realm automaticky, ale dovoľuje určiť aj vlastné pravidlá. Zmeny sa zlúčia podľa určitých pravidiel (preddefinovane zmazanie víťazí, pri zmene vlastnosti sa použije posledná zo zmien a vloženia do listu budú tiež usporiadané podľa časového poradia).

Realm dokáže zjednodušiť mnoho práce, ale na druhej strane potrebuje špeciálny prístup pri definovaní tried, ktoré majú slúžiť ako vzor pre objekty, ktoré bude ukladať do databázy. V aktuálnom čase Realm ešte stále pracuje na istej funkcionalite a dolaďuje niektoré veci.

3 Podpora offline režimu na aplikačnej úrovni

Databázy riešia zvyčajne jednu z požiadaviek offline režimu a to perzistentné úložisko. Niektoré databázové systémy majú aj zabudovanú synchronizáciu, prípadne ich výrobcovia ponúkajú systém, ktorý sa o synchronizáciu stará, ale často tomu tak nie je.

Vývojár si môže synchronizáciu vyriešiť aj sám, na mieru aplikácie. Má to výhody, napríklad optimalizovanie pre konkrétne použitie, ale vyžaduje to čas, prípadne aj isté skúsenosti.

Z tejto situácie vychádzajú riešenia na aplikačnej úrovni, ktoré ponúkajú efektívne vyriešenú synchronizáciu pomocou vhodných protokol na komunikáciu a spôsobu na prácu s konkrétnymi databázovými úložiskami. Okrem toho sa často zameriavajú aj na jednoduchosť použitia, čo vedie k zjednodušeniu a skráteniu času vývoja aplikácií.

3.1 Podpora pre Cloudové aplikácie

Cloud je systém poskytujúci sieťový prístup k zdieľateľným výpočtovým zdrojom ako napríklad servery, úložiská, aplikácie alebo služby, ktoré môžu byť jednoducho použité používateľmi, pričom je vyžadovaná minimálna interakcia s poskytovateľom daných služieb. Základné charakteristické vlastnosti cloudu sú: samoobsluha podľa používateľovej potreby, dobrá dostupnosť cez sieť, zdieľanie prostriedkov, pružnosť (škálovanie) a meraná spotreba. Služby sa dajú rozdeliť do 3 skupín alebo modelov: softvér ako služba (SaS), platforma ako služba (PaS) a infraštruktúra ako služba (IaS). (4)

Cloudové systémy spravidla ponúkajú serverové úložisko s relačnými alebo NoSQL databázami, ktoré môžu vývojári použiť v aplikáciách určených pre daný cloud. Viaceré cloudové systémy vytvorili aj synchronizačný softvér a lokálne úložisko pre mobilné alebo webové aplikácie. Väčšinou ide o jednoduché úložiska, ktorých cieľom je najmä spomínané zjednodušenie vývoja. Okrem toho zvyčajne ponúkajú aj ďalšiu funkcionalitu ako autentifikácia a podobne. V nasledujúcej časti uvedieme 3 príklady na takéto úložiska a synchronizačný softvér.

3.1.1 Microsoft Azure Mobile Apps

Azure Mobile Apps je platforma ako služba od Microsoftu. Daná služba poskytuje široký výber serverových úložísk. Okrem toho podporuje aj synchronizáciu a offline režim. Na klientovi je lokálne úložisko založené v prípade Androidu alebo Windowsu

na SQLite, v prípade iOS na natívnej databáze Core Data. Synchronizácia je kontrolovaná klientom pomocou príkazov push a pull. Push operácia pošle všetky doteraz neposlané zmeny na server pomocou REST volaní. Pomocou pull operácie zas klient požiada o dáta server a po ich prijatí ich uloží do lokálnej databázy.

3.1.2 SmartStore

Spoločnosť salesforce.com ponúka podobnú podporu offline režimu pomocou jednoduchej databázy nazývanej SmartStore. SmartStore ukladá dáta ako JSON dokumenty. Podporuje použitie indexov a napriek tomu, že ukladá všetky dáta do jednej kolekcie, podporuje použitie takzvaných soups, ktoré fungujú ako logické kolekcie.

O synchronizáciu so serverovým úložiskom sa stará knižnica SmartSync. Zmeny aktualizuje transparentne (úroveň transparentnosti sa dá pre špeciálne požiadavky nastaviť). Lokálne úložisko umožňuje využiť aj ako cache.

3.1.3 Amazon Cognito

Ďalším podobným riešením je Cognito od Amazonu. Pri použití v aplikácii umožňuje jednoducho riešiť autentifikáciu, či už pomocou vlastného systému, SAML (bezpečnostný protokol pre web) alebo pomocou sociálnych sietí (Facebook, Twitter a iné).

Dáta ukladá v lokálnej cache pamäti ako kľúč-hodnota dvojice. Aplikácia komunikuje s touto cache, čo zabezpečuje fungovanie aplikácie aj bez pripojenia k serveru. Zmeny sa aktualizujú po zavolaní synchronizačnej funkcie. Najprv prídu zmeny zo servera, vyriešia sa prípadné konflikty a potom sa pošlú lokálne zmeny na server.

3.2 Nástroje na synchronizáciu databázových systémov

Väčšina veľkých databázových systémov priamo podporuje replikáciu a fragmentáciu na viacero úložísk. Avšak problém môže nastať, ak chceme takto pracovať s rôznymi databázami. Na to sa používajú rôzne nástroje. Najčastejšie ide najmä o replikáciu, niektoré z nástrojov však podporujú aj iné spôsoby rozloženia dát v systéme.

Spoločnými vlastnosťami týchto riešení zvyčajne sú: efektívna synchronizácia pomocou využitia viacerých vlákien a efektívnych prenosov dát, širšie spôsoby využitia (master-slave replikácia, obojsmerná replikácia a iné konfigurácie), schopnosť vyriešiť aj prípady, keď je niektorý uzol nedostupný.

Takýchto nástrojov je viacero. Príkladmi môžu byť DBConvert alebo SymmetricDS. Do tejto skupiny sa dá zaradiť aj Zumero, riešenie takéhoto typu, ale so špecifickým zameraním.

3.2.1 SymmetricDS

SymmetricDS je open-source (s otvoreným zdrojovým kódom) nástroj pre replikáciu. Podporuje veľa relačných aj NoSQL databázových systémov. Komunikácia s konkrétnou databázou je riešená cez jednoduchú vrstvu, ktorá je navrhnutá tak, aby pridávanie nových databáz nebolo náročné.

SymmetricDS sa dá použiť aj na mobilných zariadeniach, kde môže pomocou SQLite ako lokálneho úložiska vytvoriť replikáciu údajov zo servera. Takýmto spôsobom sa v istých prípadoch dá využiť na implementáciu offline režimu, prípadne cachovania.

3.2.2 Zumero

Spomedzi nástrojov na synchronizáciu sa oplatí špeciálne spomenúť Zumero. Zumero je synchronizačný nástroj pre SQLite databázu, zvyčajne použitú na mobilnom zariadení a vzdialeným MS SQL Serverom. Špecifické je, že sa oproti iným riešeniam zameriava viac na podporu offline režimu na zariadení, ako len na synchronizáciu medzi dvoma vzdialenými databázami.

Zumero udržiava aktuálnu replikáciu dát zo servera fungujúcu ako cache pamäť u klienta, klientska aplikácia teda môže pracovať s lokálnou databázou a synchronizácia sa deje na pozadí. Tento prístup zabezpečuje fungovanie aplikácie aj bez pripojenia k serveru. Synchronizácia so serverom prebehne po obnovení pripojenia.

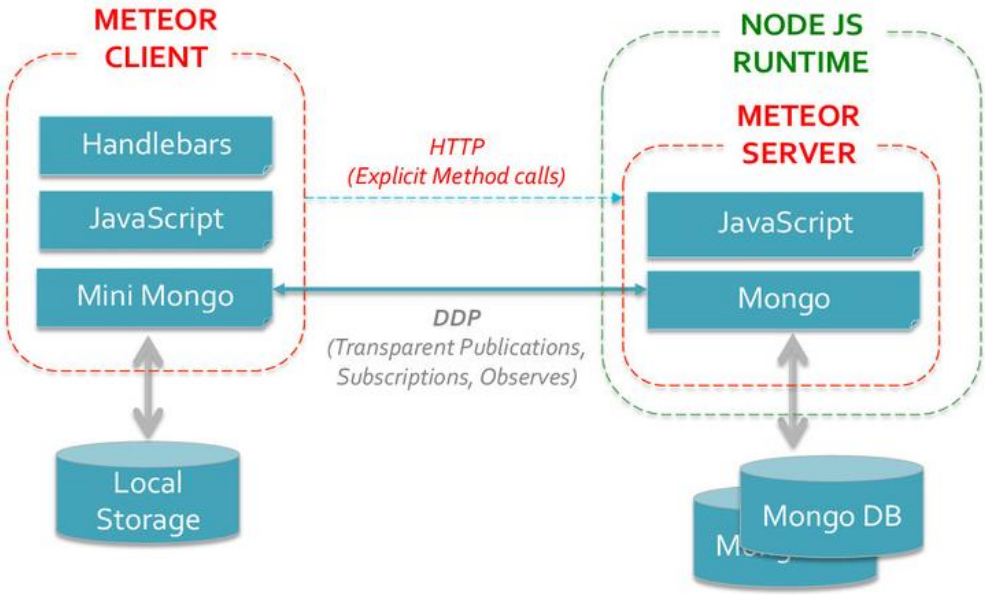
3.3 Meteor

Meteor je framework napísaný v jazyku Javascript, zameraný na rýchle vyvíjanie webových aplikácií. Je zložený zo servera, ktorý je postavený na Node.js a dáta ukladá do Mongo databázy. Konflikty v prípade Meteoru nerieši framework, ale použitá databáza, čo je v klasickom prípade Mongo. To znamená, že zvíťazí posledný zápis.

So serverom komunikuje (webový) klient pomocou DDP (distributed data protocol) protokolu. DDP protokol je klient-server protokol využívajúci návrhový vzor publish-subscribe (viac v kapitole 4.2.5). Používa sa na získavanie dopytov, posielanie zmien na server a ostatnú potrebnú komunikáciu medzi klientom a serverom.

Klient ma podobnú štruktúru ako server, dáta si ukladá do cache pamäte nazývanej Minimongo, ktorá štruktúrou a použitím reprezentuje akoby lokálnu Mongo databázu, avšak dáta sa ukladajú len v operačnej pamäti.

Vďaka Minimongu a podpore optimistic UI (viď kapitola 4.2.4) sa získava efektivita pri práci s dátami, keďže používateľ využíva lokálne prístupné dáta. Táto vlastnosť tiež zabezpečuje fungovanie aj bez pripojenia k serveru, keďže klient aj offline situáciu vyhodnotí len ako neskoré odpovede od servera. Po obnovení pripojenia Meteor pošle lokálne zmeny na server. Avšak táto forma offline režimu je limitovaná tým, že na klient funguje celý len v pamäti. V prípade webovej aplikácie, ak vykonáme napríklad obnovenie stránky bez pripojenia na server, pokračovať môžeme až po obnovení pripojenia.



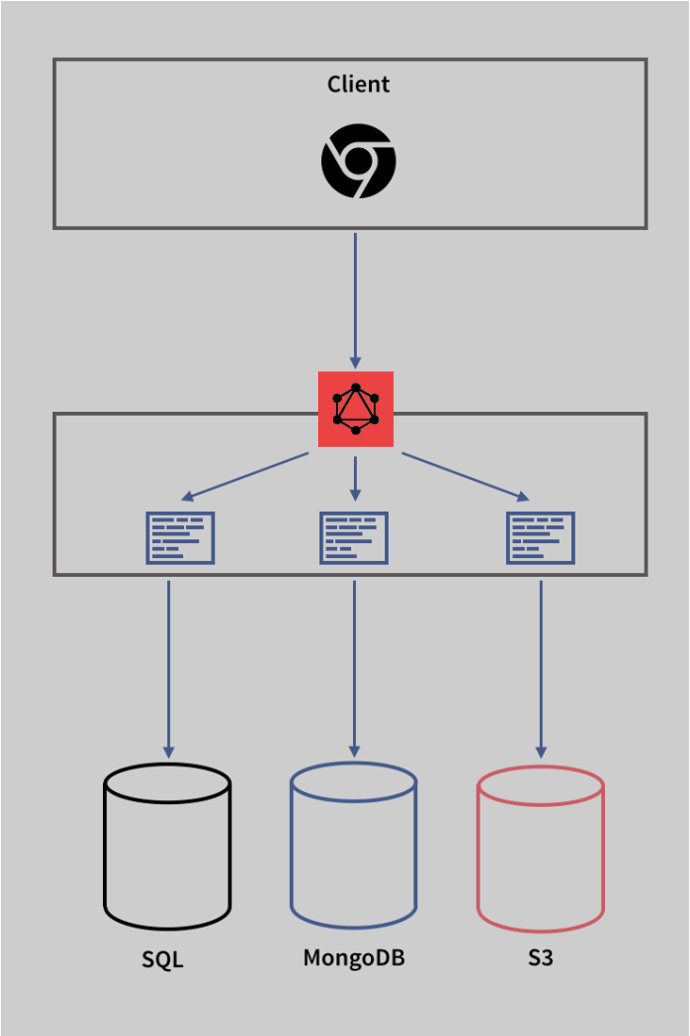
Obr. 2 Štruktúra Meteoru.

3.4 Apollo a GraphQL

GraphQL je dopytovací jazyk a behové prostredie pre tieto dopyty. Pôvodne bol vyvíjaný Facebookom. Umožňuje používať jednotné dopyty, bez ohľadu na výber konkrétneho dátového úložiska a vybrať iba tie polia z výsledku, ktoré potrebujeme.

Apollo framework používa GraphQL. Má 2 hlavné časti – klient a server. Apollo server prepája klienta a dátové zdroje. Tvorí akoby most medzi rôznymi databázami a aplikáciami na strane servera a sprostredkúva dáta klientovi. Apollo klient slúži ako rozhranie medzi klientskými aplikáciami a serverom. Využíva sa najmä

s javascriptovskými frameworkmi (primárne je to React), v aktuálnom čase sa pracuje aj na podpore pre Javu. Apollo klient používa GraphQL dopyty zadané používateľom na komunikáciu so serverom a prijaté dáta (výsledky dopytov) drží v dočasnej cache pamäti. Dokáže využiť aj ďalšie inteligentné techniky na načítavanie dát pre klienta ako prihlásenie sa na prijímanie zmien z vybraných kolekcií alebo optimistic UI. Na klientskej strane však priamo nepodporuje žiadne perzistentné úložisko.



Obr. 3 Náčrt architektúry Apollo. Dopyt z klienta (Google Chrome) poslaný na server (s troma úložiskami: relačnou databázou, Mongo databázou a úložiskom poskytovaným Amazon cloudom).

4 Návrh riešenia

Naším cieľom je navrhnúť prototyp riešenia pre podporu transparentného offline režimu a cachovania údajov zo vzdialeného dátového úložiska.

Transparentným offline režimom myslíme také využitie perzistentného lokálneho úložiska a pre používateľa prakticky neviditeľnú synchronizáciu, aby klient mohol výsledné riešenie používať aj bez pripojenia na server. Transparentnosť je v tomto kontexte vlastnosť hovoriaca o tom, že používateľ nášho riešenia nepotrebuje vedieť či má v danej chvíli pripojenie na server alebo nie. Používateľ len jednoducho pracuje s úložiskom: ukladá, mení, vymazáva a dopytuje dáta. O synchronizáciu so serverom sa bude starať nami navrhované riešenie.

Cachovanie má slúžiť k efektívnej práci s údajmi na klientskej strane, aby nebolo potrebné komunikovať so serverom pri každom dopyte o dáta. Pre dosiahnutie tohto cieľa potrebujeme mať potrebné dáta u klienta. Nie nutne celú databázu, stačia nám len tie dáta, ktoré daného používateľa zaujímajú.

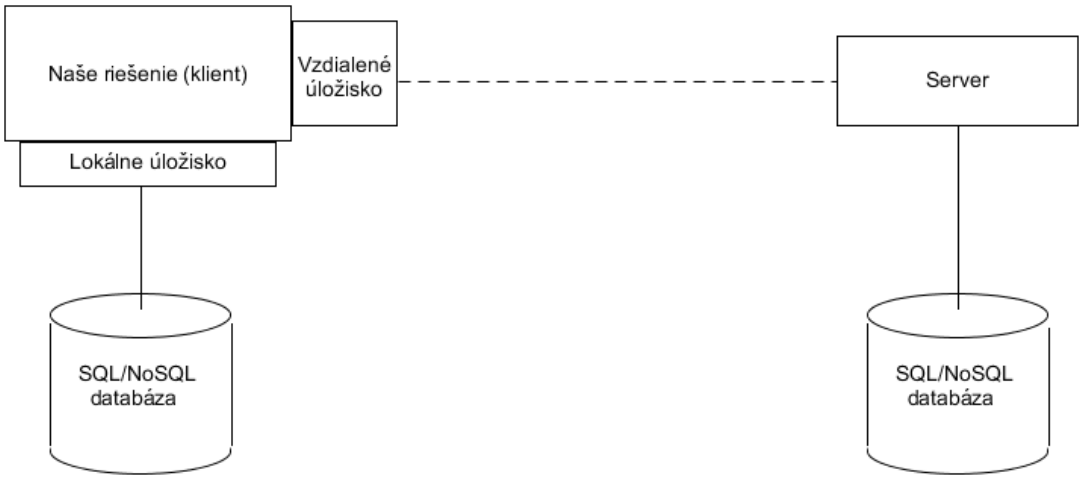
4.1 Východiská

Na dosiahnutie automatickej synchronizácie sa ako najvýhodnejšie zdá inšpirovať modelom dokumentových databáz, ktorého základnou jednotkou je dokument. Z výhod tohto modelu možno spomenúť flexibilnú schému a komplexnú štruktúru dokumentu, čo umožňuje pridávanie dátových položiek, aj vnorených (podporujú sa aj asociatívne polia). Vďaka takejto štruktúre získavame možnosť pridať potrebné metadáta priamo do dokumentu. Podstatné je však aj to, že na jednotlivé časti dokumentu sa vieme dopytovať pomocou názvov atribútov. Príklad dokumentu môžeme vidieť na Obr. 4.

```
{
  "id" : 1234,
  "name" : {
    "first" : "foo",
    "last" : "bar"
  },
  "topics": [
    "skating",
    "music"
  ]
}
```

Obr. 4 Príklad JSON dokumentu s vnoreným JSON objektom a s poľom.

Napriek použitiu modelu dokumentových databáz sme sa rozhodli neviazať lokálne ani vzdialené úložisko na konkrétny databázový systém, ani na typ (relačné alebo NoSQL) databázy. Konkrétne úložisko si vyberá vývojár pri používaní nášho riešenia. Modulárnosť úložísk rieši tenká vrstva, ktorá prepája naše riešenie a pripojenie na lokálne použitú databázu, respektíve pripojenie na server. Vhodné rozhrania navyše zabezpečia možnosť použiť aj nové, iné ako už implementované úložiská.



Obr. 5 Základný náčrt nášho riešenia s poukázaním na modulárnosť (so zobrazenými rozhraniami pre lokálne a vzdialené úložisko).

Logický model synchronizácie je inšpirovaný frameworkom Meteor (kapitola 3.3). Meteor používa návrhové vzory optimistic UI a publish-subscribe, aby sa jeho lokálne úložisko (Minimongo) dalo využiť ako rýchla cache pamäť. V princípe ide o to, že Meteor server pošle klientskej aplikácii všetky dáta, ktoré by klienta mohli zaujímať,

respektíve o ktoré prejavil záujem pomocou prihlásenia sa na odber (subscribe) na pomenovanú množinu dokumentov. Ak sa na serveri zmenia dáta, server pošle tieto zmeny všetkým klientom, ktorí sa o tieto údaje zaujímajú. Klient má takto zabezpečené, že ak vykoná dopyt pomocou svojho lokálneho úložiska, výsledok bude správny aj bez toho, aby posielal daný dopyt na server. Naviac ľubovoľné zmeny, pridávanie alebo odstránenie dokumentov sa môže diať aj lokálne, nielen na serveri, či už ako výsledok volania metódy alebo ako výsledok požiadavky klienta o zmenu konkrétnych dokumentov. Ak sa operácia udeje v lokálnom úložisku, klientska aplikácia môže hneď použiť výsledok, bez toho, aby čakala na odpoveď zo servera. Po prijatí odpovede zo servera tieto dáta nahradia lokálny výsledok. V prípade Meteoru vidno vhodnosť týchto konceptov v prípade správneho použitia, keďže viacero úspešných aplikácií bolo vytvorených pomocou tohto frameworku (napríklad Workpop alebo RentScene).

Naším cieľom bolo tento model rozšíriť najmä o to, aby výsledný systém nebol viazaný na konkrétne úložiska (respektíve protokol alebo technológiu v prípade vzdialeného úložiska).

4.2 Základné koncepty

4.2.1 Úložisko, kolekcia, dokument

V dokumentových databázach sa údaje ukladajú do dokumentov, z ktorých každý patrí do nejakej kolekcie. Rovnako je tomu v našom riešení – kolekcia a dokument sú prvkami úložiska. Úložisko sa zas stará o reálne ukladanie dát, v jeho pozadí je databáza alebo iné riešenie uchovávajúce dáta. Z pohľadu úložiska je kolekcia analógiou tabuľky v modeli relačných databáz, dokumentu zodpovedá jeden riadok tabuľky.

Dokumenty, aké používame, majú 3 dátové položky:

- jedinečné ID slúžiace na identifikovanie dokumentu,
- názov kolekcie, do ktorej patria,
- samotné údaje (obsah dokumentu).

Kolekcia nie je len pomenovanie pre množinu dokumentov, ale z implementačného hľadiska je to objekt, ktorý vykonáva potrebné operácie s dokumentmi, ktoré do tejto kolekcie patria. Okrem manipulácie nad nimi (pridanie, zmena, odstránenie), cez kolekciu sa zadávajú aj dopyty na vyhľadávanie dokumentov.

Špeciálnym prípadom kolekcie je lokálna kolekcia, t.j. kolekcia, ktorá obsahuje dokumenty, ktoré sa nesynchronizujú so serverom, ale dajú sa použiť lokálne.

4.2.2 Samoaktualizujúce sa dopyty (Live Query)

V klasických databázach, najmä v tých, ktoré sa používajú ako súčasť aplikácií, je zvyčajne výsledkom dopytu istá množina riadkov. Častokrát s danou databázou alebo vybranými dátami pracuje len jeden používateľ, dáta sa teda bez jeho pričinenia nemenia. V našom prípade je situácia oproti spomenutému jednoduchému prípadu zložitejšia, pretože navrhujeme transparentný offline režim. To znamená, že lokálne úložisko je istou replikáciou dát zo servera, ktorý nám opakovane posiela nejaké dáta (ide o zmeny vyvolané inými užívateľmi, prípadne našimi operáciami na serveri), pričom o ich spracovanie sa stará naše riešenie, používateľ o príchode konkrétnej aktualizácie dát nevie. Z toho vyplýva, že ak by používateľ zadal rovnaký dopyt dvakrát, ak sa medzitým udiala aktualizácia lokálnych dát, výsledné množiny dopytov budú rôzne, bez toho, aby o nejakých zmenách mohol vedieť. Na riešenie tohto problému používame takzvané samoaktualizujúce sa dopyty (Live Query).

Výsledkom dopytu pri použití samoaktualizujúcich sa dopytov je objekt s výslednou množinou dokumentov. Rozdielom je, že vždy po zmene dát v systéme sa aktualizujú aj dokumenty v tomto výsledku, čo zabezpečuje, že budeme mať stále aktuálne dáta. Výsledky takýchto dopytov sa teda menia v čase, rovnako ako dáta v lokálnom úložisku. Samotné dokumenty potom vieme získať aj ako jednoduchý nemeniaci sa zoznam alebo aj inými spôsobmi tak, aby sme vždy po zmene používali najnovšie dáta; závisí na tom, ktorú metódu použijeme.

4.2.3 Transakcia

Transakcie v našom systéme predstavujú najmenšiu jednotku zmien údajov. Každá modifikácia údajov je vykonávaná v transakcii. Transakcia potom obsahuje jednu alebo viac zmien dokumentov. Použitím návrhového vzoru reaktor sa následne zabezpečuje to, že transakcie (operácie nad dátami) sa vykonávajú serializovane.

Základný postup práce je taký, že používateľ zadá žiadosť o operáciu, vytvorí sa transakcia s potrebnými údajmi (napríklad parametre pre metódy) a vloží sa ako súčasť žiadosti do systému. Samotnú operáciu (pridanie, zmenu, odstránenie dokumentov) už vykonáva systém a zmeny sa uložia do transakcie.

Vďaka tomuto spôsobu vykonávania operácií sa vyhneme nechceným situáciám. Predstavme si napríklad prípad dokumentu, ktorý predstavuje žiaka. Daný dokument má medzi inými aj pole, kde je zapísaný priemer známok konkrétneho žiaka. Používateľ chce následne vykonať dve operácie – prvá zmení priemer známok daného dokumentu na 2, druhá operácia odstráni všetky dokumenty (všetkých žiakov) s priemerom väčším ako 3. Pri vykonaní týchto akcií na rozdielnych vláknach sa môže stať, že sa daný dokument najprv odstráni, až potom ho bude chcieť úložisko zmeniť. Druhá operácia by v takom prípade skončila neúspechom.

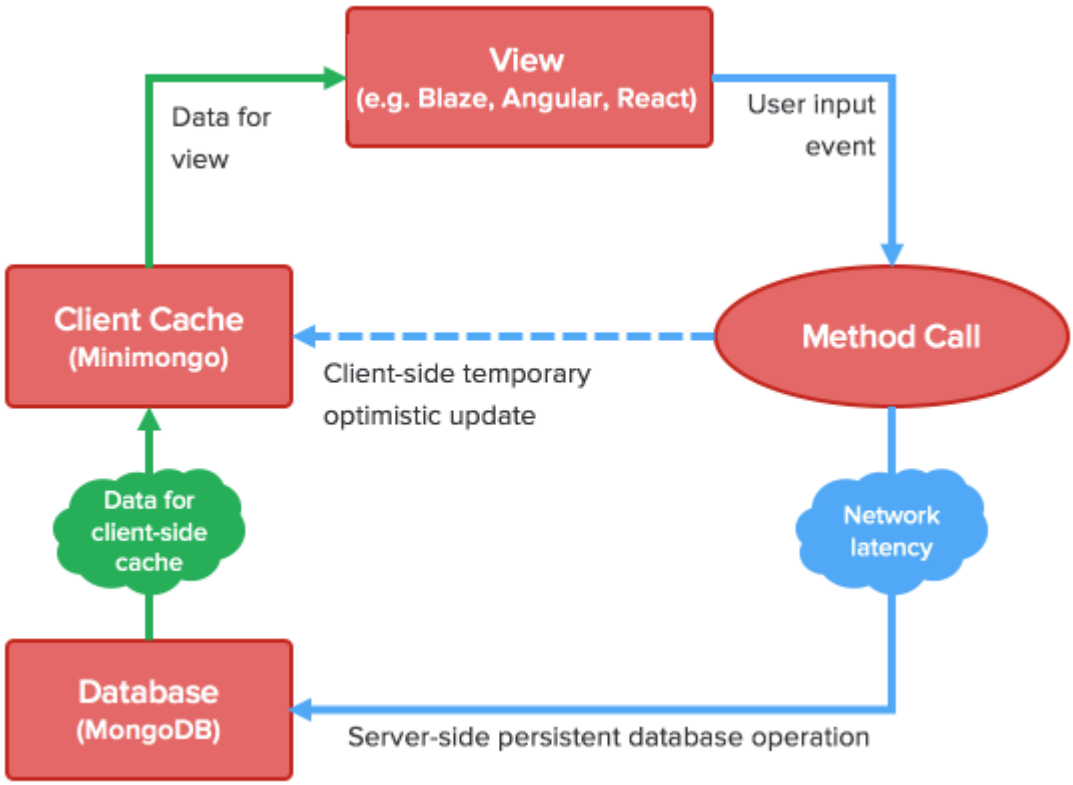
Použitie transakcií je jedno z riešení, ktoré garantuje, že transakcie sa vykonajú v rovnakom poradí ako boli zadané. Tento koncept používame nielen pre akcie týkajúce sa zmien dát v lokálnom úložisku, ale rovnako aj pre vzdialené úložisko.

4.2.4 Optimistic UI

Optimistic UI (voľne preložené ako „optimistické používateľské rozhranie“) je taký spôsob zmien užívateľského rozhrania, v ktorom sa používa odhadnutie a napodobenie výsledku pri volaní metód na serveri a zmena príslušného užívateľského rozhrania ešte pred prijatím zmien zo servera. Motiváciou je to, že používatelia očakávajú od aplikácií rýchle (okamžité) reakcie na ich vstup a pravdepodobnosť toho, že od servera prídu rozdielne ako lokálne dáta je v mnohých prípadoch použitia dostatočne malá. Tento model je úspešne používaný napríklad frameworkmi Meteor a Apollo.

Typický sled udalostí, ktoré sa dejú v aplikáciách podporujúcich Optimistic UI je takýto: užívateľ iniciuje nejakú akciu, zodpovedajúca požiadavka sa pošle na server, pričom lokálne sa vykoná príslušná metóda a aktualizuje sa užívateľské rozhranie. Po krátkej chvíli dôjde odpoveď zo servera, lokálne modifikácie dát sa zahodia a nahradí ich výsledok zo servera. Náčrt tohto postupu, konkrétne jeho použitie Meteorom, možno vidieť na Obr. 6.

Napriek tomu, že naše riešenie nemá priamo užívateľské rozhranie, je dosť pravdepodobné, že aplikácie, ktoré ho budú využívať, budú takéto rozhranie mať. Ďalším dôvodom pre využitie tohto vzoru na prácu s dátami je, že lokálne špekulatívne metódy umožňujú pracovať s dátami aj vtedy, keď nemáme pripojenie na server. Offline režim v tomto prípade zodpovedá situácií, kedy trvá čakanie na odpoveď od servera extrémne dlho.



Obr. 6 Použitie Optimistic UI vo frameworku Meteor (5). Po akcii používateľa sa daná metóda okrem poslania na server vykoná aj lokálne. Po prijatí výsledku zo servera sa ním nahradia odpovedajúce lokálne dáta.

4.2.5 Publish-subscribe

Publish-subscribe je návrhový vzor určený na posielanie správ. Je zložený z prvkov dvoch rolí – odosielatelia (publisher), ktorí ponúkajú správy zadelené do určitých tried (subscription) a odberatelia (subscriber), ktorí sa prihlásia na odber správ z vybraných tried. Odberatelia potom prijímajú len správy z tried, na ktoré sa prihlásili na odber.

V kontexte klient-server aplikácií zvyčajne server plní úlohu odosielateľa, na ktorého sa prihlásia klienti (odberatelia) a prijímajú dáta, ktoré im potom server posiela. Pri porovnaní napríklad s klasickým spôsobom pracovania s REST protokolom v aplikácii, kde chceme mať u klienta aktuálne dáta (pričom dáta sa môžu meniť na serveri) je medzi týmito spôsobmi niekoľko rozdielov. Najvýraznejšia zmena je v spôsobe komunikácie. Pokiaľ pri použití REST protokolu si opätovne pýtame pomocou http metód dáta od servera a server odpovedá (klient vykonáva takzvaný „pull“) v publish-subscribe modeli (predpokladáme, že server je odosielateľ) je server ten, kto automaticky posiela nové dáta klientovi (server vykonáva „push“).

4.2.6 Úložisko

Úložiskom budeme nazývať základnú časť nášho systému, starajúcu sa o interakciu s používateľom (vývojárom) a logiku celého systému, delegovanie operácií pre lokálne a vzdialené úložisko, hlásenie chýb a ďalšie veci.

4.2.7 Vzdialené úložisko

Vzdialené úložisko je časť systému zabezpečujúca komunikáciu a synchronizáciu so serverom. Tvorí vrstvu medzi základnou časťou systému a serverom. Jeho podstatnou črtou je, že je modulárne. Pomocou vhodného rozhrania umožňuje jednoducho implementovať a použiť verziu komunikujúcu so serverom cez REST protokol, DDP protokol alebo ľubovoľným iným spôsobom.

4.2.8 Lokálne úložisko

Lokálne úložisko slúži ako úložisko dát pre klientsku aplikáciu. Prepája systém s lokálnou databázou. Rovnako ako vzdialené úložisko, aj lokálne úložisko je modulárne – možno s ním použiť prakticky ľubovoľnú relačnú alebo NoSQL databázu, prípadne aj iné úložisko. Stará sa o vytvorenie potrebných kolekcií, upravovanie dokumentov a vykonávanie dopytov.

4.3 API vzdialeného úložiska

API vzdialeného úložiska súvisí s v súčasnosti využívanými modelmi ako publish-subscribe, vzdialene volanie metód a taktiež s tým, že nepodporujeme len jeden protokol.

Základné metódy, ktoré API ponúka sú prihlásenie sa na odber, zrušenie z prihlásenia sa na odber, pridanie poslucháča čakajúceho na správy od servera, (vzdialené) volanie metódy a aplikovanie lokálnych zmien.

4.3.1 Získanie dát

Jednou zo základných úloh vzdialeného úložiska je transparentne získať aktuálne dáta od servera. Vzdialené úložisko priamo podporuje model publish-subscribe pomocou metód prihlásenie sa na odber a zrušenie odberu zmien zo servera. Pri použití tohto modelu sme automaticky informovaný o zmenách. Ak server nepodporuje model publish-subscribe, musíme použiť iný spôsob, napríklad si periodicky pýtať od servera nové dáta.

Pre správne fungovanie celého systému je potrebné tiež pridanie poslucháča počúvajúceho na správy od servera, ktoré mu je vzdialené úložisko povinné poskytnúť. Týmto spôsobom komunikuje vzdialené úložisko so základným úložiskom. Ide konkrétne o správy týkajúce sa ukončenia požiadavky na server (prihlásenie sa na odber alebo vzdialené volanie metódy), prijatia dát od servera, zmeny pripojenia, odhlásenia sa z odberu, zneplatnenia kolekcie alebo prípadných chýb.

4.3.2 Posielanie dát na server

Z pohľadu, kde sa realizuje aplikačná logika, môžeme rozdeliť klient-server aplikácie do dvoch typov:

- inteligentný server vykonávajúci operácie a posielajúci zmeny klientom,
- server slúžiaci ako úložisko dát, databáza, bez aplikačnej logiky.

V prípade prvého modelu, kedy sa realizuje aplikačná logika na serveri (tomuto modelu sa hovorí aj tenký klient), sa využíva vzdialené volanie metód. Klient pošle požiadavku na vykonanie konkrétnej metódy a parametre potrebné na jej vykonanie. Po ukončení behu volanej metódy pošle server potrebné zmeny klientom. Volanie metód v offline režime zabezpečujeme v takomto prípade v našom riešení podporou simulovaného vykonávania potrebných metód lokálne, podobne ako je tomu u Optimistic UI.

Pri použití druhého modelu, ak sa uskutočňuje aplikačná logika na klientovi, je vykonávanie zmien v offline režim riešené automaticky, potrebné je riešiť len synchronizáciu so serverom. To sa robí jednoducho, poslaním zmien na server pomocou metódy na aplikovanie lokálnych zmien.

4.3.3 Ďalšie metódy

Ďalšie metódy vzdialeného úložiska sú metódy pre štart a stop tohto úložiska, slúžiace na prípravu potrebných zdrojov a vytvorenie pripojenia, respektíve ukončenie práce.

Pri lokálnom simulovaní vykonávania vzdialených metód, je ideálne v prípade vytvorenia dokumentu priradiť tomuto dokumentu rovnaké ID, aké bude príslušný novovytvorený dokument mať na serveri (napríklad kvôli odkazovaniu sa na dokumenty pomocou ID). Vzdialené úložisko umožňuje pridať doň ID generátor simulujúci generovanie ID na serveri. Tento generátor bude potom použitý v systéme na

generovanie ID pre vytvárané dokumenty pri lokálnom vykonávaní metód, ktoré boli zavolané aj na serveri.

Poslednou metódou vzdialeného úložiska je metóda na získanie jedinečného identifikátora relácie (session), ktorý sa dá použiť na obnovenie príslušnej relácie a pokračovanie od momentu prerušenia.

4.4 API lokálneho úložiska

Lokálne úložisko predpokladá databázu ako perzistentné úložisko údajov, čo ovplyvnilo návrh jeho API.

API lokálneho úložiska ponúka metódu na pripojenie k databáze (alebo prípadnému inému úložisku) a metódy pre prácu s kolekciami a dokumentmi. Okrem týchto metód obsahuje aj metódy pre štart a zastavenie úložiska (rozhranie net.jards.core.LocalStorage).

4.4.1 Kolekcie

Lokálne úložisko umožňuje používateľovi pridanie a odstránenie kolekcie iba pri štarte systému. Neskôr je pridanie možné len v prípade, že server pošle dokumenty s kolekciou, aká sa v lokálnom úložisku ešte nenachádza. Podobne môže dôjsť aj k odstráneniu všetkých dokumentov z kolekcie v prípade, že bola systémom zneplatnená.

4.4.2 Dokumenty

Vytvorenie, úprava a odstránenie dokumentu sú tri základné operácie pre ich modifikáciu. Tieto operácie sa používajú aj jednotlivo užívateľmi, aj systémom na zapísanie zmien, ktoré prišli zo servera.

Okrem modifikácií dokumentov zabezpečuje lokálne úložisko aj vyhľadávanie v dokumentoch pomocou dopytov. Podporujeme dva typy výsledkov dopytov – samoaktualizujúci sa dopyt a vrátenie prvého vhodného dokumentu.

4.4.3 Indexy

Databázové indexy sú v istej forme podporované prakticky vo všetkých relačných a aj vo väčšine NoSQL databáz. V našom lokálnom úložisku tiež podporujeme ich použitie. Pre každú kolekciu môže vývojár zadať ľubovoľný počet indexov, ktoré potom pri vkladaní alebo úprave dokumentu môže lokálne úložisko z tohto dokumentu vyextrahovať a použiť.

Jedným z možných využití tohto prístupu môže byť implementovanie rýchleho lokálneho úložiska používajúceho relačnú databázu s indexmi nad vybranými poľami dokumentov. Indexovanie zvolených polí dokumentov umožní rýchle vyhodnotenie (niektorých) dopytov bez toho, aby bol obsah dokumentu rozparsovaný. Dopyty používajúce tieto polia budú takto vykonané efektívne databázou napriek tomu, že náš systém používa dokumentový model, v ktorom sú klasicky všetky dáta v jednom reťazci.

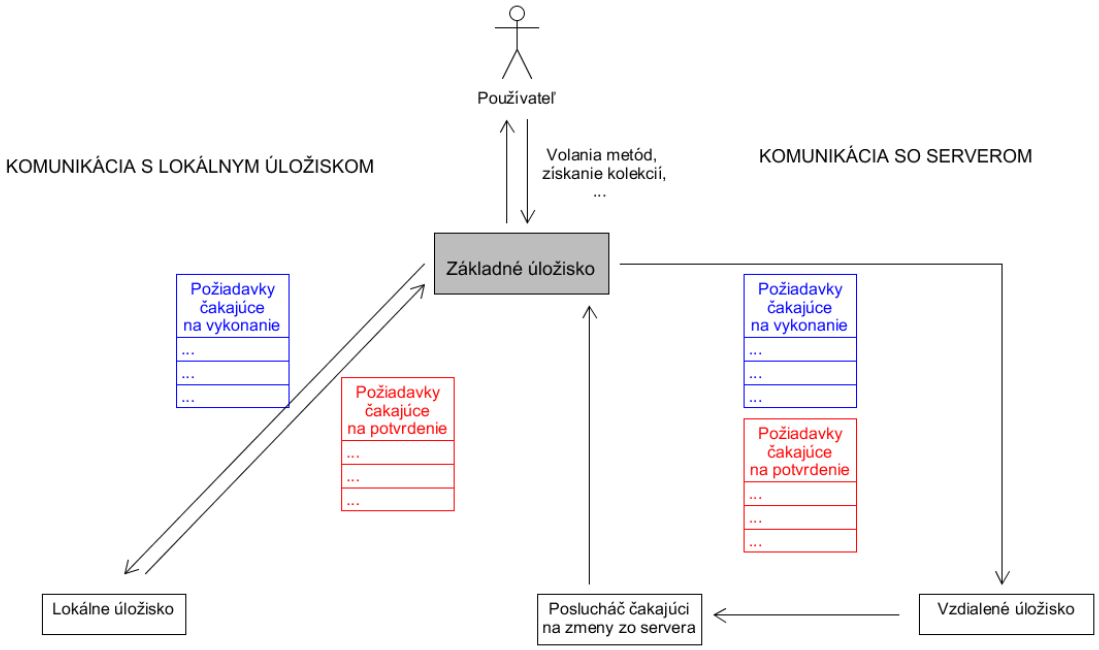
4.5 Úložisko

V tejto kapitole sa budeme venovať základnému úložisku, ktoré sa stará o interakciu s používateľom a vo výraznej miere zabezpečuje logiku nášho riešenia. Je jadrom celého systému.

4.5.1 Základná logika úložiska

Základné úložisko tvorí akýsi most medzi používateľom a oboma úložiskami (lokálnym aj vzdialeným). Užívateľ pomocou neho zadáva žiadosti na vykonanie metód (vzdialené volanie alebo lokálne vykonanie), pomocou ktorých upravuje dáta, pridáva simulácie zabezpečujúce lokálne vykonanie pri vzdialenom volaní metód a prihlasuje sa na odber pri publish-subscribe modeli, prípadne ruší tento odber.

Na komunikáciu s lokálnym a vzdialeným úložiskom používa základné úložisko transakcie. Užívateľ zavolá metódu, pre dané volanie sa vytvorí požiadavka s transakciou, ktorá sa vykoná príslušným úložiskom hneď, ako je to možné. Rovnako funguje aj prihlásenie sa na odber a odhlásenie sa z neho. Použitá architektúra systému umožňuje, že metódy základného úložiska sú realizované ako neblokujúce.



Obr. 7 Základné úložisko ako most medzi užívateľom a úložiskami, využívajúce požiadavky s transkaciami na serializované vykonávanie operácií nad úložiskami.

4.5.2 Dáta zo servera

Na získanie zmien zo servera základné úložisko používa poslucháča počúvajúceho na tieto zmeny. Metódy tohto poslucháča volá vzdialené úložisko, tým sa zabezpečuje prísun dát do jadra systému.

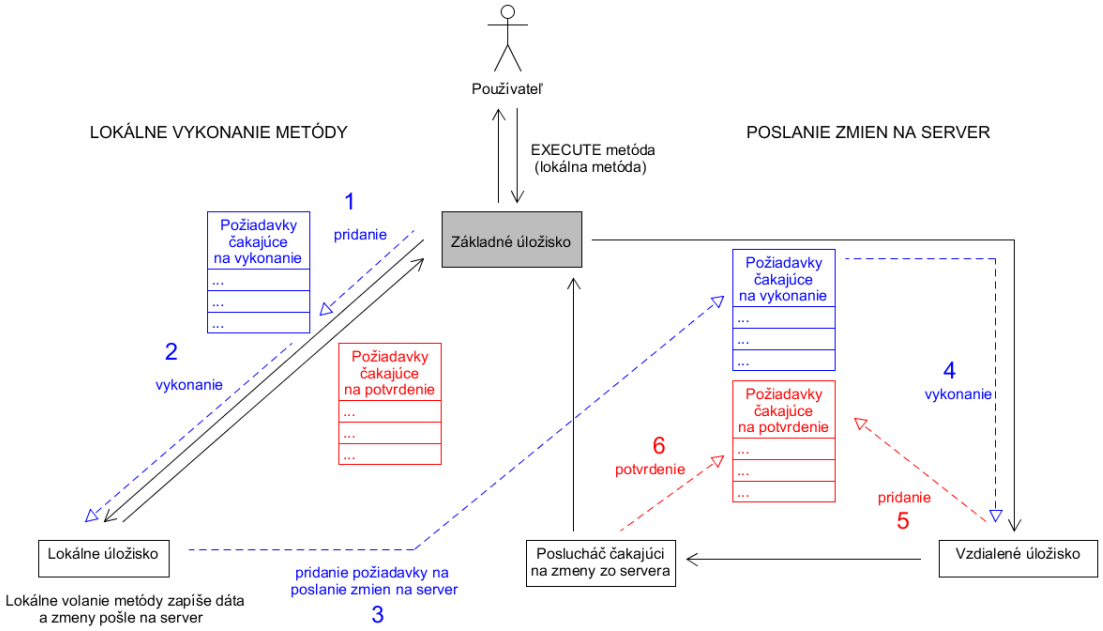
Základné úložisko je takto informované o ukončení volaní (potvrdení požiadaviek), zmene stavu pripojenia, prípadnom zneplatnení kolekcie alebo chybách a o prijatí nových dát od servera. Pripomíname, že server posiela nášmu systému dáta či už ako výsledok vzdialeného volania metódy alebo napríklad pri použití publish-subscribe modelu. V našom modeli takto prijaté dáta priamo zapíšeme do databázy pomocou lokálneho úložiska. Server je v tomto smere autoritou, ak nám prídu dáta od neho, prepíšeme podľa toho lokálne údaje.

4.5.3 Lokálne vykonanie metódy

Tento spôsob sa používa v prípade, keď je aplikačná logika realizovaná na klientovi a server slúži ako centrálne úložisko dát. Používateľ v tomto prípade zadá požiadavku o takéto vykonanie metódy, k požiadavke systém vytvorí transakciu a úložisko vykoná lokálne túto metódu hneď, ako je to možné. Pridané, zmenené a odstránené dokumenty sa uložia do transakcie a systém túto požiadavku so zmenami následne posunie vzdialenému úložisku, kde sa tieto zmeny aplikujú (sú poslané na

server). Transakcia je do času jej potvrdenia odložená medzi požiadavky čakajúce na potvrdenie od servera, kde sa v prípade straty pripojenia dá nájsť a poslanie zmien sa môže zopakovať po obnovení pripojenia. Metóda je ukončená po potvrdení jej vykonania serverom. Náčrt sledu týchto udalostí možno vidieť na Obr. 8.

Ak sme v čase volania v offline režime, mení sa len to, že vzdialené úložisko na klientovi čaká s poslaním lokálnych zmien, kým sa neobnoví pripojenie.



Obr. 8 Táto schéma znázorňuje dianie pri lokálnom volaní metódy; lokálne vykonanie operácií a poslanie zmien na server.

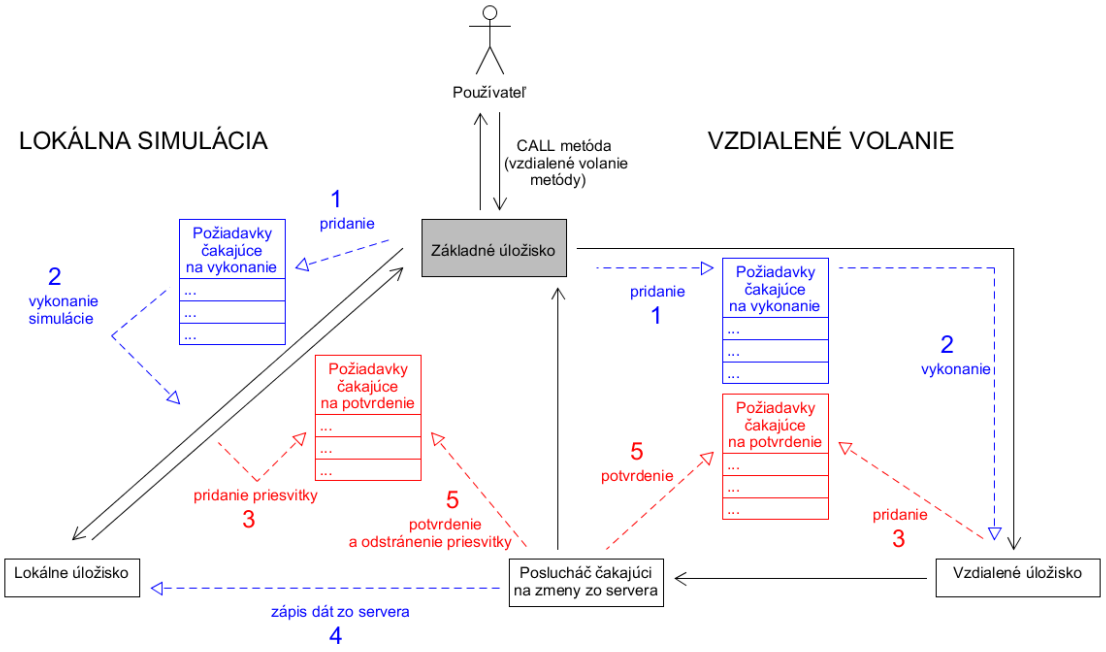
4.5.4 Vzdialené volanie metódy

Vzdialené volanie metód sa používa v modeli s tenkým klientom, kde vykonáva aplikačnú logiku server. Pre tento prípad podporujeme vykonanie lokálnej simulácie takýchto metód, podobne ako je tomu u Optimistic UI.

Za predpokladu, že metóda vykonávajúca simuláciu už bola do základného úložiska pridaná, je základný sled udalostí pri vzdialenom volaní metódy so simuláciou takýto: užívateľ zadá požiadavku o vykonanie metódy, k tejto požiadavke sa vytvorí transakcia a následne sa požiadavka pošle na server, pričom transakcia čaká na potvrdenie. Súčasne sa lokálne hneď, ako je to možné, vykoná príslušná simulácia. Lokálne sa však zmenené dáta nezapisujú do databázy (prípadne inej implementácie úložiska), ale uložia sa do transakcie, ktorá čaká v úložisku na potvrdenie zo servera, čím sa vytvorí takzvaná priesvitka. Fungovanie priesvitiek je podrobnejšie popísané

v ďalšej kapitole. Server zatiaľ vykoná volanie metódy a pošle nám dáta, ktoré systém zapíše do perzistentného lokálneho úložiska. Po prijatí potvrdenia o dokončení vykonávania metódy na serveri, sa odstráni príslušná transakcia v oboch úložiskách spomedzi čakajúcich na potvrdenie.

Možno poznamenať, že zápis dát od servera v tomto prípade nie je priamo súčasťou vzdialeného volania metódy, tieto dáta sú len jednoducho zapísané v lokálnom úložisku ako je to popísané v kapitole 4.5.2. Potvrdenie o dokončení metódy by však malo byť od servera prijaté až po prijatí všetkých zmien vykonaných danou metódou alebo súčasne s prijatím posledných zmien.



Obr. 9 Vzdialené volanie metódy so simuláciou; požiadavka o vykonanie metódy sa pošle súčasne na server aj sa vykoná lokálne. Lokálne sa vytvorí len odhad výsledku („priesvitka“), ktorý je nahradený dátami zo servera (po ich prijatí).

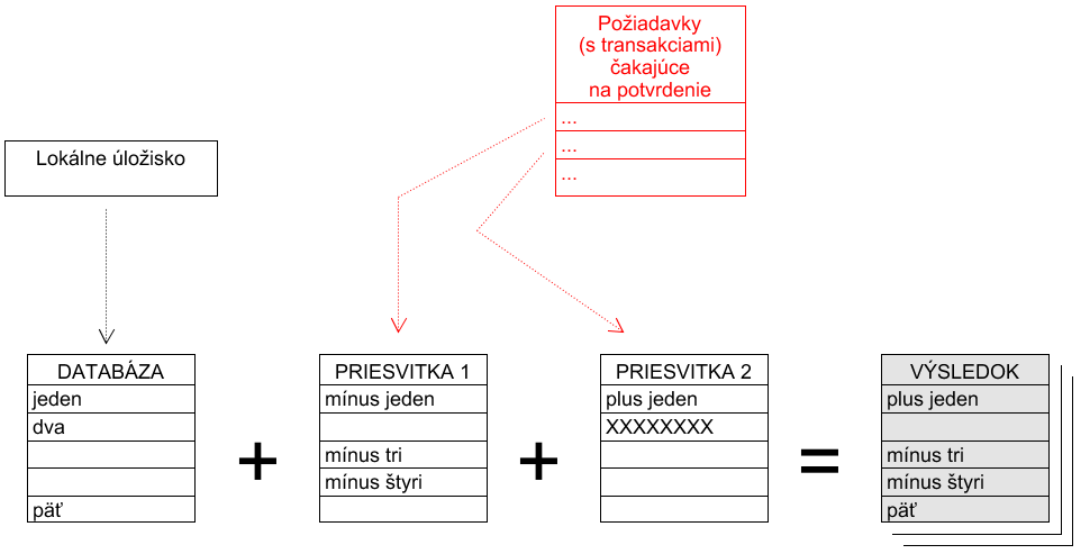
4.5.5 „Priesvitky“

O takzvaných „priesvitkách“ hovoríme v rámci lokálneho úložiska v súvislosti so simuláciou vzdialeného volania metód. „Priesvitkou“ myslíme priesvitný papier, ktorý má tú vlastnosť, že ak naňho niečo napíšeme a priložíme ho povedzme na stránku knihy, budeme vidieť stránku knihy, okrem miest kde je na priesvitke niečo napísane, tam budeme vidieť to, čo sme napísali (viď Obr. 10).

Ako už bolo spomenuté, simulácia vzdialeného volania metódy nezapisuje dáta priamo do databázy, zmeny sa uložia len do príslušnej transakcie. Táto transakcia sa zas

odloží v systéme, až pokým server nepotvrdí dokončenie vykonávania metódy. Takéto transakcie sú v našom systéme usporiadané podľa poradia, v akom boli im prislúchajúce simulácie vykonané a správajú podobne ako priesvitky. Keď používateľ požiada o dáta, lokálne úložisko nájde príslušné údaje v databáze, a tie sa potom postupne preložia priesvitkami, to jest aktualizujú sa podľa dokumentov v transakciách – dokumenty pridané simuláciami sa pridajú do výslednej množiny, zmenené sa vo výslednej množine nahradia zmenami a odstránené sa z výslednej množiny odstránia.

Dôvodom, prečo sme sa rozhodli pre tento model, je jednoduché zahodenie priesvitky (transakcie), po potvrdení ukončenia vykonávania metódy na serveri. V tom čase už máme lokálne dostupné všetky dáta zo servera, teda priesvitka (transakcia) už nie je potrebná. Ak by sme zmeny zo simulácie zapisovali do databázy, potrebovali by sme si stále viesť záznamy o nami vykonaných zmenách pre prípad, že zmeny zo servera budú rozdielne ako lokálne. Napríklad, ak by lokálna simulácia vytvorila dokument, ale na serveri by sa nevytvoril, potrebovali by sme tento dokument z databázy odstrániť. Použitie transakcií a ich odstránenie po potvrdení ukončenia metódy takéto situácie rieši automaticky.



Obr. 10 Ukážka princípu "priesvitiek" (priesvitných papierov, na ktoré môžeme niečo napísať) na názornom príklade.

4.5.6 Samoaktualizujúce sa dopyty (Live query)

Samoaktualizujúce sa dopyty využívame v našom systéme v kombinácii s priesvitkami spomínanými v predošlej kapitole. Výsledný model zabezpečuje získanie výsledku s aktuálnymi dátami, napriek prebiehajúcim zmenám z rôznych zdrojov.

Výsledkom samoaktualizujúceho sa dopytu nie je len jednoduchý zoznam s dokumentmi, ale objekt predstavujúci výslednú množinu. V tejto výslednej množine sú dáta z databázy a systém priesvitiek. Výsledok získame aplikovaním priesvitiek. Môžeme ho získať ako nemeniaci sa zoznam dokumentov, alebo aj ako v čase sa meniace dáta prihlásením pozorovateľa na zmeny.

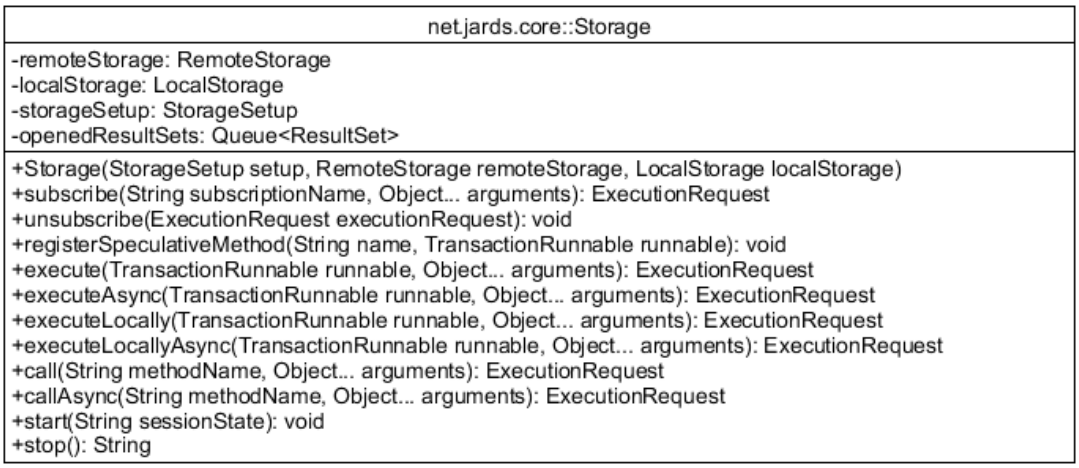
Ak ďalej spomenuté zmeny odpovedajú definícii dopytu, tak výslednú množinu dokumentov systém aktualizuje vždy po vykonaní simulácie (pridanie priesvitky do výslednej množiny), potvrdení ukončenia simulácie (odstránenie priesvitky), prijatí zmien od servera alebo po lokálnom vykonaní metódy (aktualizujú sa dáta z databázy) alebo po zneplatnení kolekcie (odstránenie dát z databázy).

5 Implementácia

Naše riešenie sme sa rozhodli implementovať ako knižnicu v programovacom jazyku Java. Tento programovací jazyk sme si vybrali z dôvodu, že je rozšírený, existuje preň mnoho knižníc, ktoré sa dajú použiť a nie je závislý na platforme. Zároveň je možné rýchle portovanie na mobilnú platformu Android.

5.1 Úložisko (Storage)

Centrálne úložisko, v implementácii nazývané `Storage`, slúži na prijímanie požiadaviek od používateľa, prepájanie lokálneho a vzdialeného úložiska a vo výraznej miere sa stará o logiku systému. Prehľad základných metód a premenných triedy základného úložiska možno vidieť na Obr. 11.



Obr. 11 Náčrt implementácie triedy Storage (hlavného úložiska).

5.1.1 Iniciácia

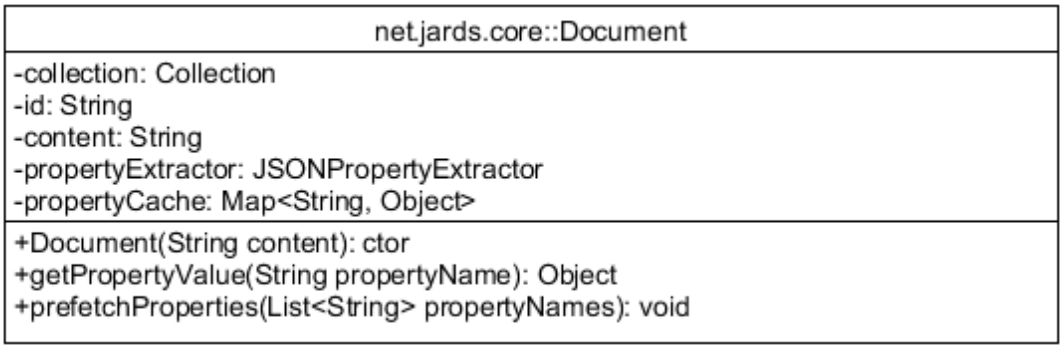
Modularita lokálneho a vzdialeného úložiska sa zabezpečuje pri vytvorení hlavného úložiska. Používajú sa na to príslušné rozhrania. Najprv sa vytvoria implementácie oboch úložísk a potom sa hlavnému úložisku nastavia do príslušných premenných pomocou konštruktora tejto triedy. V konštruktore triedy `Storage` sa tiež vytvorí pozorovateľ na zmeny od servera a pridá sa do vzdialeného úložiska (implementácia rozhrania `RemoteStorage`). Pomocou tohto pozorovateľa na zmeny sa dáta od servera dostanú cez vzdialené úložisko do hlavného úložiska a aplikujú sa lokálne (o pozorovateľovi na zmeny od servera viac v kapitole 5.2.3).

Metódou `start` sa spustí najprv lokálne a vzdialené úložisko, a po ich spustení sa začnú vykonávať jednotlivé požiadavky zadané používateľom.

5.1.2 Dokumenty a kolekcie

Dáta v našom systéme ukladáme do dokumentov. Tieto dokumenty obsahujú identifikátor, názov kolekcie, do ktorej patria, a samotné dáta tvoriace ich obsah. Okrem uchovávania údajov, dokumenty dokážu vyextrahovať z ich obsahu hodnoty podľa zadaných polí.

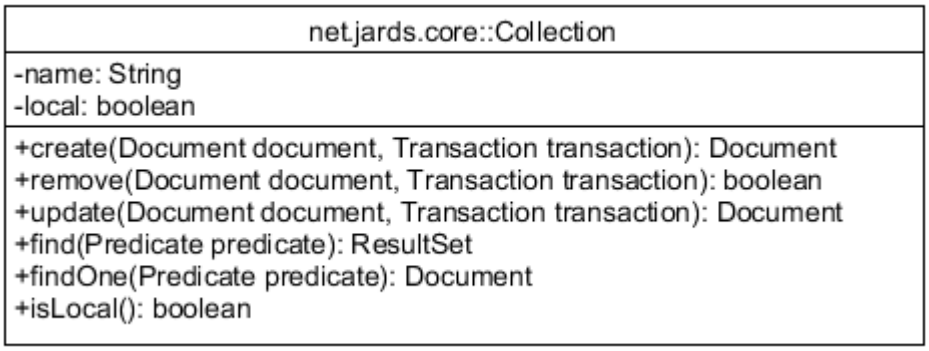
Vytvorenie dokumentu prebieha v dvoch fázach. V prvej používateľ vytvorí dokument s obsahom. Tento dokument ešte nemá identifikátor a nie je priradený žiadnej kolekcii. Následne používateľ pomocou metódy vybranej kolekcie zavolá metódu na pridanie ním vytvoreného dokumentu. Metóda vybranej kolekcie sa postará o vytvorenie nového dokumentu, priradí mu identifikátor, nastaví kolekciu na svoj názov a tento novovytvorený dokument vráti používateľovi.



Obr. 12 Trieda dokument v našom systéme.

Kolekcie slúžia na prácu s dokumentmi. Možno si ich predstaviť ako tabuľky v relačných databázach. Každá kolekcia má názov, pomocou ktorého sa dá získať zo systému. Špeciálnym prípadom kolekcie je lokálna kolekcia. Lokálna kolekcia obsahuje dokumenty, ktoré sa nesynchronizujú so serverom.

Pomocou kolekcie môžeme pridávať, meniť, odstraňovať a vyhľadávať dokumenty. Samotné operácie však nevykonáva kolekcia, ale transakcia, aby sa zabezpečilo serializované vykonávanie týchto požiadaviek.



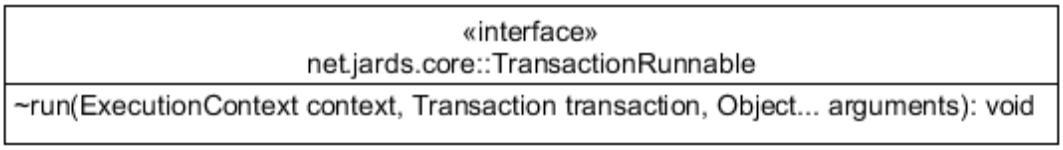
Obr. 13 Trieda reprezentujúca kolekcie v našom systéme.

5.1.3 Serializované vykonávanie operácií

Na získanie serializovaného vykonávania operácií v našom systéme sme sa inšpirovali návrhovými vzormi aktívny objekt (active object) a reaktor (reactor). Podľa vzoru aktívny objekt oddelíme vykonanie metódy od jej zavolania. Podľa vzoru reaktor zasa požiadavky z rôznych metód vkladáme do prislúchajúcich radov, odkiaľ sa vyberú vláknami, ktoré vykonávajú serializovane požiadavky pre lokálne a vzdialené úložisko.

Používateľ môže zadať jednoduchú požiadavku týkajúcu sa vykonania niektorej z metód vzdialeného úložiska alebo môže vykonať lokálne zmeny dát.

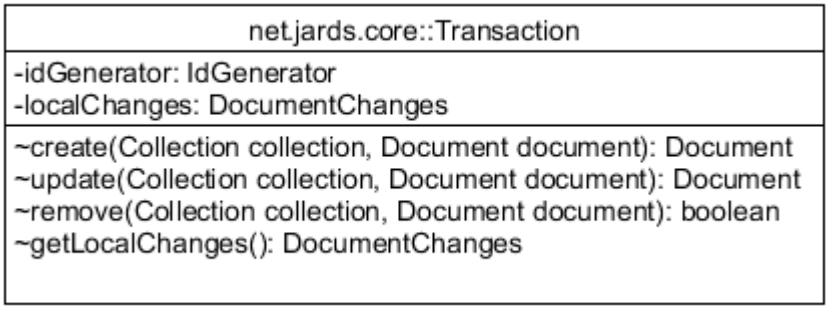
Meniť lokálne dáta môže používateľ len implementovaním metódy `run` rozhrania `TransactionRunnable` („spustiteľná transakcia“). Implementácie tejto triedy sa používajú na vykonávanie lokálnych metód a lokálnych simulácií. Pomocou parametra kontext, ktorý je v metóde `run` k dispozícii, môže používateľ získať kolekciu, pomocou ktorej možno potom pridávať, meniť alebo odstraňovať dokumenty.



Obr. 14 Rozhranie TransactionRunnable s metódou run, ktoré sa používa na vykonávanie lokálnych zmien.

Používateľ teda vo svojej implementácii `TransactionRunnable` volá metódy kolekcie na pridanie, zmenu alebo odstránenie dokumentu. Kolekcia deleguje tieto zmeny dát na transakciu, ktorá ich vykoná volaním metód lokálneho úložiska. Kedy dochádza k volaniu metódy `run` implementácie `TransactionRunnable` a aký je

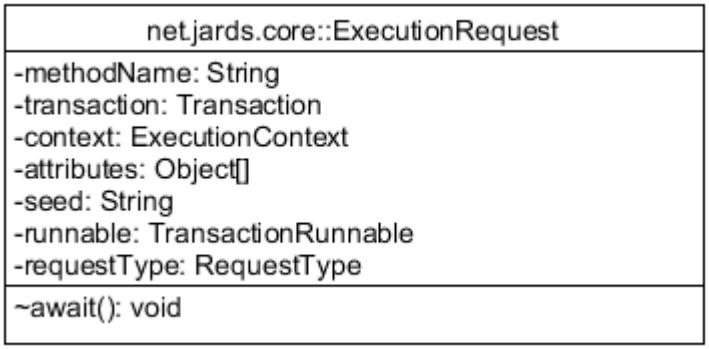
životný cyklus jednotlivých požiadaviek o vykonanie operácii je podrobnejšie rozpísané v nasledujúcej kapitole.



Obr. 15 Transakcia s metódami na pridanie, zmenu a odstránenie dokumentu.

5.1.4 Životný cyklus operácií (od volanie po vykonanie operácií)

Pri zavolaní jednotlivých metód hlavného úložiska (viď Obr. 11) sa vytvorí pre vybranú metódu požiadavka. Pri volaní metódy pre lokálne vykonanie (`execute` metóda), sa musí pridať medzi parametre tejto metódy implementácia rozhrania `TransactionRunnable`. Pri vzdialenom volaní metódy (metóda `call`) sa použije len názov metódy. Ak sa pred tým uložila príslušná simulácia (implementácia `TransactionRunnable`) pod názvom volanej vzdialenej metódy v úložisku, vykoná sa aj táto simulácia. Pre tieto dve volania metód (lokálne volanie a simulácia) pridá systém do požiadavky aj transakciu.



Obr. 16 Trieda pre požiadavku na vykonanie akcie lokálneho alebo vzialeného úložiska.

Požiadavka sa po vytvorení pridá do vláknovo bezpečného radu, kde čaká na vykonanie. V systéme používame 5 radov. Sú to rady pre:

- operácie vzdialeného úložiska čakajúce na vykonanie,
- operácie vzdialeného úložiska čakajúce na potvrdenie,

- operácie lokálneho úložiska čakajúce vykonanie,
- operácie lokálneho úložiska čakajúce na potvrdenie a
- zmeny zo servera čakajúce na zápis lokálnym úložiskom.

Operácie týkajúce sa vzdialeného úložiska rieši vlákno pre vzdialené úložisko. Operácie týkajúceho sa lokálneho úložiska tiež príslušné vlákno, s tým dodatkom, že zmeny zo servera majú pri vykonávaní prednosť (ak sú v systéme takéto zmeny, vykonajú sa pred ostatnými lokálnymi zmenami). Tieto vlákna kontrolujú im pridelené rady a ak sa v im prislúchajúcom rade nachádza požiadavka, tak ju podľa jej typu vykonajú (pomocou potrebného úložiska).

Prácu vlákien vysvetlíme podrobnejšie spolu so životným cyklom požiadaviek pre jednotlivé typy operácií.

- Zmeny od serveru sa podľa návrhu v kapitole 4.5.2 vkladajú pomocou pozorovateľa na tieto zmeny do im určeného radu. Z tohto radu ich vyberie vlákno určené na lokálne vykonávania a zapíše ich priamo do databázy. Zmeny na zápis od servera majú prednosť pred inými lokálnymi zmenami (vlákno vykonávajúce lokálne zmeny najprv skontroluje rad so zmenami od servera, vykoná zmeny z tohto radu a až potom prejde k radu s čakajúcimi lokálnymi operáciami).
- Požiadavky na server (prihlásenie sa na odber, zrušenie prihlásenia sa na odber, vzdialené volanie metódy a aplikovanie lokálnych zmien na server) patria k základným operáciám úložiska (návrh v kapitole 4.5.1). Po vytvorení požiadavky pre niektorú z týchto operácií, sa vytvorená požiadavka uloží do radu operácií pre vzdialené úložisko. Odtiaľ sa vyberie vláknom posielajúcim požiadavky na server, zavolá sa príslušná metóda vzdialeného úložiska a požiadavka sa vloží do radu nepotvrdených operácií vzdialeného úložiska, kde ostane, kým sa nepotvrdí. Ak stratí klient pripojenie k serveru, požiadavky z radu nepotvrdených operácií pre vzdialené úložisko sa vykonajú znovu po obnovení pripojenia.
- Pre lokálne vykonávanie metódy sa postupuje ako sa navrhlo v kapitole 4.5.3. Najprv sa vytvorí požiadavka, ktorá sa pridá do radu pre lokálne operácie. Z tohto radu sa vyberie vláknom vykonávajúcim lokálne zmeny, z požiadavky sa získa implementácia `TransactionRunnable` a vykoná sa metóda `run`

tejto implementácie. Operácie s dokumentmi volané metódami kolekcie v tejto metóde `run` sa vykonajú pomocou transakcie. Táto transakcia zavolá príslušné metódy lokálneho úložiska, ktoré zmeny zapíšu do databázy. Ak bola požiadavka len na lokálne vykonanie (`executeLocally`), vykonávanie sa končí. Ak ide o požiadavku pre lokálne vykonanie s poslaním zmien na server (`execute`), zmeny sa uložia do transakcie. Požiadavka s transakciou so zapísanými zmenami sa následne vloží do radu pre čakajúce operácie vzdialeného úložiska, kde sa vykoná.

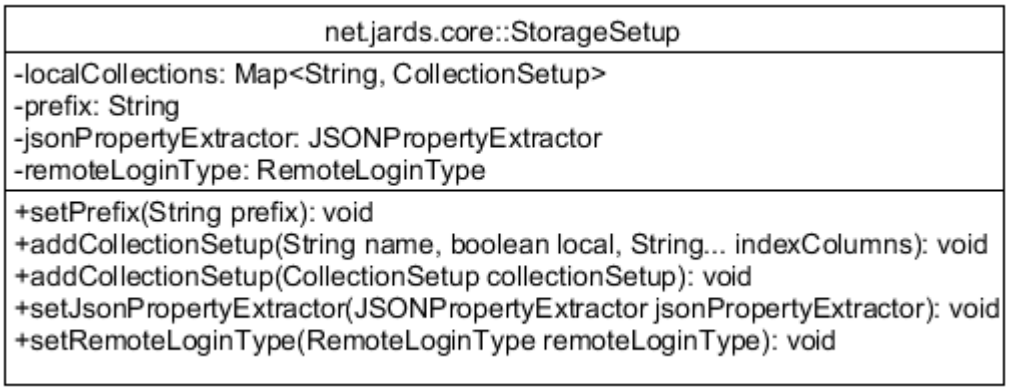
- Vzdialené volanie metódy prebieha podľa návrhu v kapitole 4.5.4. Vytvorí sa požiadavka, ktorá sa pomocou príslušného radu pošle vzdialenému úložisku na vykonanie (tento postup sme už opísali v prechádzajúcich bodoch – vzdialené úložisko zavolá vybranú metódu na serveri a prijaté dáta od servera sa zapíšu). Ak sa v základnom úložisku nachádza simulácia pre zavolanú metódu, do požiadavky sa pridá aj kód simulujúci vykonávanie metódy (implementácia `TransactionRunnable`) a táto požiadavka sa pridá do radu lokálnych operácií čakajúcich na vykonanie. Vlákno vybavujúce lokálne zmeny ju vykoná, avšak transakcia nepoužije lokálne úložisko na zapísanie zmien do databázy, tieto zmeny si len uloží. Požiadavka s touto transakciou sa následne pridá do radu požiadaviek čakajúcich na potvrdenie, kde ostáva, pokiaľ nie je pozorovateľom na správy od servera jej vykonanie potvrdené. Do toho času vytvárajú tieto zmeny takzvanú priesvitku (kapitola 4.5.5). Pre používateľa sú viditeľné rovnako ako keby boli zapísané v databáze.

5.1.5 Nastavenie úložísk (StorageSetup)

Na nastavenie rôznych parametrov a spôsobu práce jednotlivých úložísk sa používa trieda `StorageSetup`. Objekt tejto triedy sa vytvorí ešte pred vytvorením úložísk a cez ich konštruktory sa týmto úložiskám posunie na použitie.

Pomocou aktuálnej implementácie triedy `StorageSetup` môžeme nastaviť spôsob, akým sa bude server snažiť komunikovať so serverom (bez prihlásenia, prihlásenie sa, ak je to možné alebo nutne sa prihlásiť). Pre lokálne úložisko sa v tejto triede nastaví prefix pre kolekcie, ktorý slúži na oddelenie kolekcií jednotlivých používateľov a pridajú sa kolekcie, ktoré chceme mať v systéme.

Aj keď systém používa predvolený extraktor na extrahovanie vlastností z dokumentov, možno systému nastaviť aj iný extraktor pomocou odpovedajúcej metódy v triede `StorageSetup`.



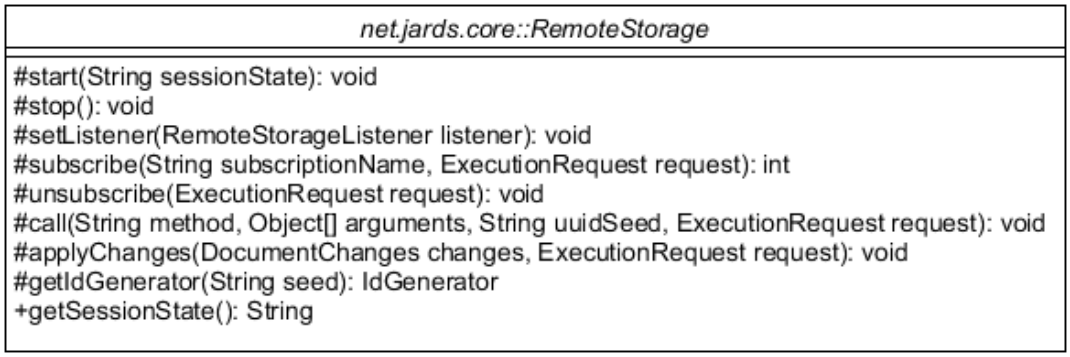
Obr. 17 Trieda StorageSetup s aktuálne implementovanými nastaveniami.

5.2 Synchronizácia so serverom

O synchronizáciu so serverom sa stará vzdialené úložisko. Pomocou rozhrania sa dá implementovať pre rôzne typy protokolov a teda aj serverov. My sme implementovali verziu komunikujúcu cez DDP protokol a jednoduchý server vo frameworku Meteor.

5.2.1 Rozhranie vzdialeného úložiska (RemoteStorage)

Rozhranie vzdialeného úložiska obsahuje všetky metódy potrebné na posielanie požiadaviek na server. Pred začatím práce sa zavolá metóda štart, kde môže prebehnúť pripojenie na server. Pomocou metódy na nastavenie pozorovateľa (`setListener`) sa pridá implementácii vzdialeného úložiska pozorovateľ na správy od servera, ktorého je pre správne fungovanie potrebné o správach od servera informovať. Vzdialené úložisko má tiež metódu na získanie generátora identifikátorov, ktorý sa s používa pri simuláciách.



Obr. 18 Rozhranie vzdialeného úložiska (RemoteStorage).

5.2.2 Implementácia vzdialeného úložiska

Vzdialené úložisko sme implementovali na komunikáciu so serverom pomocou DDP protokolu. Na komunikáciu so serverom sme použili knižnicu Java DDP Client[1].

Uvedená knižnica má metódy pre prihlásenie sa na odber dát zo servera podľa publish-subscribe modelu, zrušenie tohto odberu, vzdialené volanie metód a poslanie lokálnych zmien na server.

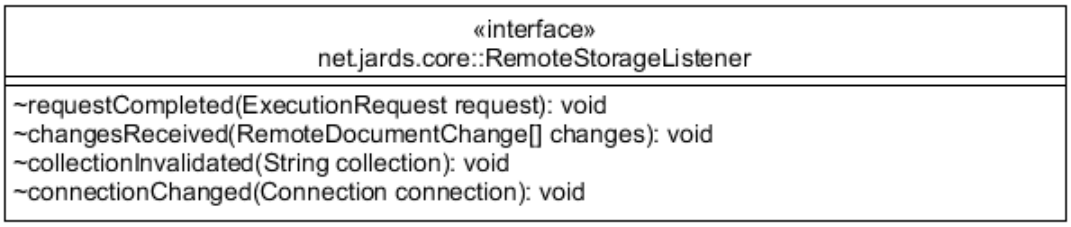
V našej implementácii vzdialeného úložiska si tiež pamätáme všetky prihlásenia sa na odber, aby sme ich mohli obnoviť pri obnovení pripojenia k serveru po prípadnej strate spojenia.

5.2.3 Pozorovateľ na zmeny od servera (RemoteStorageListener)

Rozšírením triedy `DDPListener` z použitej knižnice počúvame na správy od servera, sledujeme zmenu pripojenia a následne presúvame vhodne transformované správy pozorovateľovi na správy od servera uloženému v našej implementácii vzdialeného úložiska.

Pozorovateľ na zmeny ďalej vykonáva potrebné zmeny pomocou hlavného úložiska. Prijaté zmeny posiela do radu na zápis zmien od servera (rovnako zneplatnenie kolekcií), serverom potvrdené požiadavky odstráni z radov kde čakajú na potvrdenie alebo vykonanie (prípad simulácie).

[1] Knižnica je dostupná na: https://github.com/kenyee/java-ddp-client



Obr. 19 Základné metódy pozorovateľa na zmeny od servera.

5.2.4 Generovanie identifikátorov

Na lokálne generovanie identifikátorov používame generátor identifikátorov od vzdialeného úložiska. Tento generátor je zodpovedný za to, aby lokálne vygenerované identifikátory v simuláciách vzdialených volaní metód boli rovnaké ako identifikátory pridelené dokumentom na serveri. To nám zabezpečí, že ak sme sa odkazovali pomocou identifikátora na dokumenty vytvorené v simulácií, nestratíme tieto väzby ani nahradením dokumentov zo simulácie dokumentmi vytvorenými na serveri.

Pri volaní metódy sa požiadavke priradí náhodný reťazec (aktuálna verzia používa náhodné UUID). Tento reťazec sa potom pošle na server aj do lokálnej simulácie, kde slúži ako semienko pre algoritmus generujúci identifikátor pre pridaný dokument (lokálne je to algoritmus v generátore identifikátorov poskytnutý vzdialeným úložiskom). Pri použití s frameworkom Meteor na serveri generuje identifikátory takzvaný algoritmus alea[2], ktorý sme implementovali v Jave pre lokálne použitie.

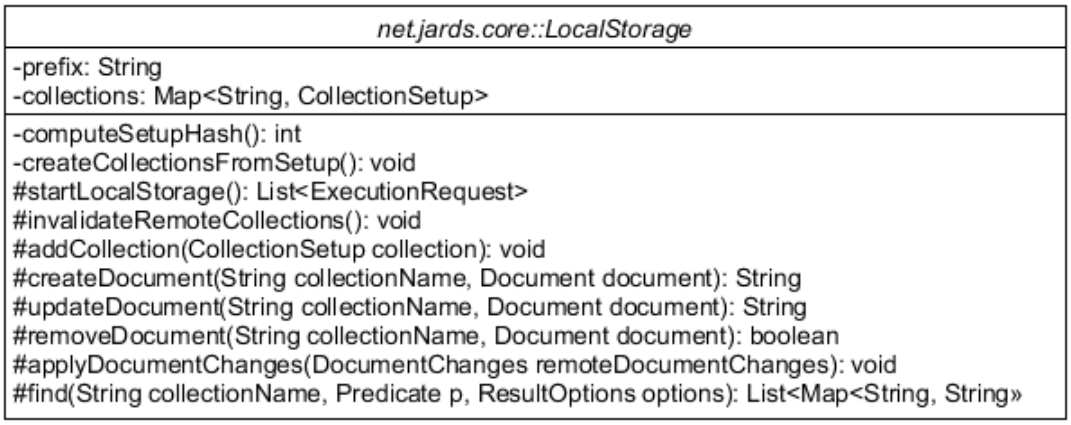
5.3 Lokálne úložisko

Lokálne úložisko sa stará o zápis a čítane dát z lokálnej databázy. Pomocou rozhrania sa dajú pridať lokálne úložiska podporujúce rôzne typy databáz. My sme vytvorili verziu lokálneho úložiska využívajúcu vstavanú databázu SQLite. Motivácia bola najmä tá, že táto databáza sa používa na platforme Android.

5.3.1 Metódy lokálneho úložiska

Trieda pre lokálne úložisko je vytvorená ako abstraktná trieda. Metódy lokálneho úložiska možno rozdeliť do 2 skupín: abstraktné metódy slúžiace ako rozhranie pre komunikáciu s databázou a metódy realizujúce pripravenie lokálneho úložiska na prácu.

[2] Kód algoritmu alea je dostupný na Github:
https://github.com/meteor/meteor/blob/devel/packages/random/random.js



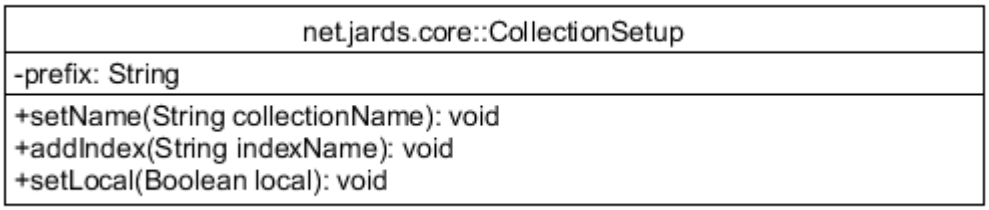
Obr. 20 Abstraktná trieda reprezentujúca lokálne úložisko s metódami tvoriacimi rozhranie pre rôzne implementácie a metódami slúžiacimi pre nastavenie kolekcií pri štarte úložiska.

5.3.2 Pridanie kolekcií (CollectionSetup)

Pri štarte lokálneho úložiska sa najprv interne vykoná nastavenie kolekcií. Používa sa na to špeciálna trieda `CollectionSetup` (Obr. 21). Každej kolekcii, ktorá má byť použitá v systéme, sa musia najprv pomocou tejto triedy nastaviť potrebné parametre. Tieto nastavenia kolekcií sa potom pridajú do objektu triedy `StorageSetup`, ktorý sa použije pri vytvorení lokálneho úložiska.

Pri vytváraní lokálneho úložiska sa najprv zo všetkých kolekcií, ktoré chce používateľ vytvoriť vypočíta hash. Pomocou zadaného prefixu sa potom úložisko pokúsi prečítať jeden riadok dát zo špeciálnej kolekcie definovanej systémom. Ak táto kolekcia pre daný prefix existuje a je v nej zapísaný rovnaký prefix, nové kolekcie sa nevytvárajú. Ak takáto kolekcia neexistuje alebo je v nej zapísaná iná hash hodnota, systém vytvorí takúto kolekciu s vypočítanou hash hodnotou a všetky kolekcie zadané používateľom. Prípadné existujúce konfliktné kolekcie odstráni. Týmto spôsobom sa nemusí pri každom štarte aplikácie vytvárať či kontrolovať a aktualizovať databázový model lokálneho úložiska. Aktuálne riešenie určuje hash len jednoduchým zreťazením nastavení.

Samotné vytvorenie zadaných kolekcií vykoná implementácia lokálneho úložiska, v našom prípade sa vytvorí pre každú kolekciu tabuľka v SQLite. Tabuľky obsahujú stĺpec pre identifikátor slúžiaci zároveň ako primárny kľúč, stĺpec pre dáta tvoriace obsah dokumentov a stĺpce pre zadané indexy.



Obr. 21 Trieda CollectionSetup používajúca sa na nastavenie kolekcií v lokálnom úložisku. Preddefinovaný typ indexu je reťazec, trieda obsahuje aj metódu pre číselný index.

5.3.3 Práca s dokumentmi a použitie indexov

Implementácia lokálneho úložiska pre SQLite sa stará o zapísanie dokumentov do príslušných tabuliek (kolekcií) databázy, úpravy dokumentov v databáze a odstraňovanie dokumentov z SQLite databázy.

Ak lokálne úložisko zapisuje alebo upravuje dokument v kolekcii, pre ktorú boli definované indexy, vyextrahuje tieto indexy z obsahu dokumentu a zapíše ich do im prislúchajúcich stĺpcov. Databáza následne vytvorí nad týmito stĺpcami vlastné indexy, čo zabezpečí, že vyhľadávanie pomocou týchto indexov bude rýchle. Bez extrahovania dát z obsahu dokumentu a ich použitia ako indexu by databáza musela pre dopyt používajúci tieto dáta čítať všetky riadky vybranej tabuľky.

ID	dáta	meno
1	{"meno":"Ján"}	Ján
2	{"meno":"Peter"}	Peter
3	{"meno":"Milan"}	Milan

Obr. 22 Príklad tabuľky (kolekcie) so stĺpcom pre index s názvom meno. Z vkladaných alebo menených dokumentov je potom položka meno vyextrahovaná a vložená do príslušného stĺpca.

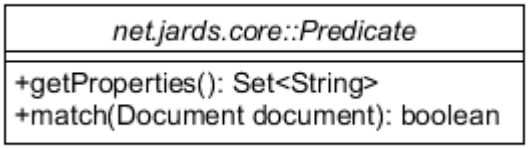
5.3.4 Dopyty pomocou predikátov

Na definovanie podmienok a filtrovanie dopytov sme vytvorili abstraktnú triedu reprezentujúcu predikát (`Predicate`). Trieda `Predicate` má dve metódy: abstraktnú metódu, ktorá vyhodnotí či zadaný dokument vyhovuje definovanému filtru a metódu, ktorá vráti množinu polí používaných týmto predikátom.

Trieda `Predicate` umožňuje vytvorenie ľubovoľných implementácií pre potrebné použitie. My sme implementovali niekoľko základných operácií (konjunkciu, disjunkciu, porovnanie dvoch polí a porovnanie poľa s hodnotou), ktoré sa dajú skladať

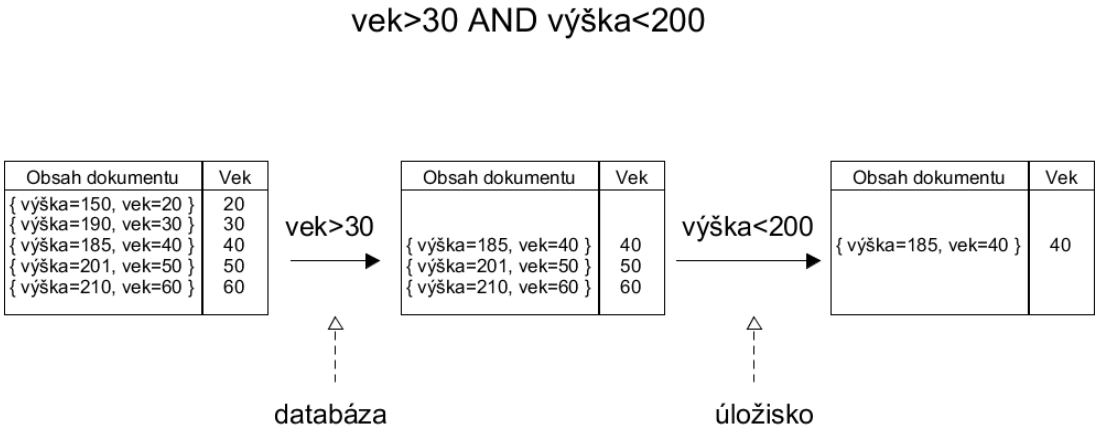
na vytvorenie komplexných podmienok (napríklad konjunkcia troch porovnaní poľa s hodnotou).

Lokálne úložisko prijme dopyt s implementáciou predikátu a ďalšími nastaveniami (napríklad usporiadanie výsledných dokumentov). Následne lokálne úložisko rekurzívne zhodnotí použitie predikátu (aj doňho vnorené predikáty) a vytvorí samotný dopyt. V našej implementácii pre SQLite použijeme predikát vtedy, ak používa len polia extrahované z obsahu dokumentu do stĺpcov (indexované polia). Ak je dopyt definovaný aj na iných ako do stĺpcov extrahovaných poliach, databáza vráti všetky dokumenty danej kolekcie.



Obr. 23 Trieda predikát používaná na definovanie a filtrovanie dokumentov pri dopytoch.

Koncept predikátov nám umožňuje podporiť aj iné stratégie použitia v databázovom úložisku. Napríklad ak je najvyššie postaveným predikátom konjunkcia, databáza môže použiť na filtrovanie všetky priamo v nej vnorené výrazy, ktoré používajú len polia extrahované do stĺpcov. Tým sa zmenší počet vrátených dokumentov aj v prípade, že dopyt nie je definovaný len na extrahovaných poliach.



Obr. 24 Filtrovanie pomocou konjunkcie.

5.3.5 Implementácia samoaktualizujúcich sa dopytov

Výsledkom dopytu v našom systéme je objekt obsahujúci naďalej sa meniacu výslednou množinou dokumentov (`ResultSet`).

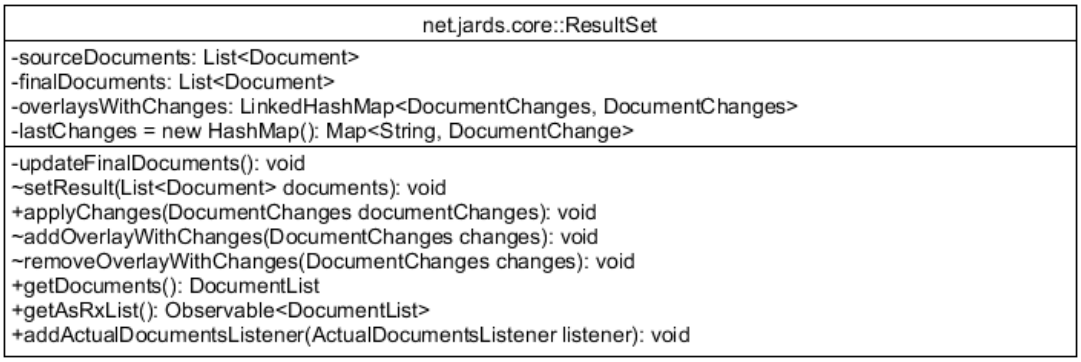
Dokumenty sa v triede `ResultSet` nachádzajú vo viacerých zoznamoch. Zoznam zdrojových dokumentov sa nastaví po vytvorení objektu dokumentmi z databázy. Pred vložením sa prefiltrujú pomocou definovaného predikátu (databáza nemusí nutne použiť každý predikát, napríklad kvôli nepodporovaným poliam). Zdrojové dokumenty sa menia tiež po aplikovaní zmien z lokálneho vykonania metód. Ak dokumenty v zmenách patria do danej kolekcie, tak sa príslušne aplikujú. Pridané dokumenty sa pridajú, ak odpovedajú predikátu. Ak zmenené dokumenty odpovedajú predikátu, tak nahradia pôvodné, ak neodpovedajú predikátu, pôvodný predikát sa odstráni. Odstránené dokumenty z prijatých zmien sa zo zdrojového zoznamu odstránia.

Po vykonaní lokálnej simulácie metódy, sa vytvorená priesvitka pridá do všetkých aktívnych výsledných množín dokumentov. Zmeny sa prefiltrujú podľa predikátu a vložia sa do usporiadanej mapy. Po potvrdení vykonania metódy prislúchajúcej simulácii sa príslušná priesvitka odstráni.

Z priesvitiek vytvárame mapu predstavujúcu výsledok aplikácie všetkých priesvitiek, teda mapu posledných zmien. Kľúčom je identifikátor dokumentu, hodnotou je zmena príslušného dokumentu. Pridanie priesvitky aplikuje zmeny tejto priesvitky na mapu posledných zmien. Odstránenie priesvitky spôsobí opätovné vytvorenie mapy s poslednými zmenami.

Ľubovoľná zo spomenutých metód môže spôsobiť zmenu vo výslednej množine dokumentov. Po každej zmene sa z toho dôvodu volá metóda na aktualizovanie zoznamu finálnych dokumentov. V tejto metóde sa najprv vytvorí nový zoznam, do ktorého sa pridajú dokumenty zo zoznamu zdrojových dokumentov. Potom sa na novovytvorený zoznam aplikuje mapa posledných zmien. Výsledný zoznam obsahuje sa priradí do premennej uchovávajúcej finálne dokumenty. Tento zoznam obsahuje aktuálne lokálne dostupné dáta.

Po vytvorení nového finálneho zoznamu sa súčasne informujú všetci prihlásení pozorovatelia na aktuálne dáta o novom zozname. Podobne sa pošle nový zoznam všetkým prihláseným na RxJava Observable. Inou možnosťou na získanie výsledku dopytu je metóda na získanie zoznamu dokumentov – táto metóda vráti nemeniaci sa zoznam vytvorený z finálnych dokumentov.



Obr. 25 Premenné a metódy triedy reprezentujúcej v čase sa meniacu množinu výsledných dokumentov (ResultSet).

6 Príklady použitia a overenie

6.1 Serverová aplikácia pre testy

Na umožnenie testovania našej knižnice sme vytvorili jednoduchú klient-server aplikáciu. Serverovú časť sme implementovali vo frameworku Meteor podľa oficiálneho tutoriálu na ich webovej stránke.

Aplikácia slúži na plánovanie úloh. V kolekcii „tasks“ ukladá dokumenty reprezentujúce jednotlivé úlohy. Podstatné polia úloh sú identifikátor, vlastník úlohy (autor) a samotný text úlohy.

Server odosiela podľa publish-subscribe modelu prihláseným prijímateľom ich aktuálne naplánované úlohy (podľa vlastníka). Na manipuláciu s úlohami má server metódy na pridanie, zmenu a odstránenie úlohy. My sme pridali metódu na pridanie úlohy aj so semienkom na generovanie identifikátora. Rovnako sme kvôli testom povolili posielanie lokálnych zmien na server (Meteor to štandardne nepovoľuje).

6.2 Návod k použitiu

V tejto podkapitole sa nachádza základný tutoriál popisujúci prácu s knižnicou. Najprv však uvedieme zmeny, ktoré je potrebné vykonať na serveri implementovanom podľa tutoriálu (ako je naznačené v kapitole 6.1). Bez vykonania nižšie uvedených zmien nebude fungovať ukážková klientska aplikácia ani niektoré metódy uvedené v tutoriáli.

6.2.1 Zmeny serverovej aplikácie

Najprv odporúčame implementovať serverovú aplikáciu. Postupujeme podľa pokynov uvedených v oficiálnom tutoriáli [3]. Funkčnosť aplikácie sa dá otestovať v prehliadači.

Po získaní funkčnej aplikácie môžeme vykonať potrebné zmeny. Aby nám fungovala testovacia aplikácia, stačí pridať metódu na pridanie plánovanej úlohy s názvom `insertWithSeed` s pridaným parametrom, ktorým je semienko pre generátor identifikátorov. Metóda je takmer identická s metódou `insert` vykonávajúcou jednoduché pridanie úlohy. Odporúčame neupravovať metódu `insert`, ale pridať novú metódu, inak prestane fungovať pridávanie úloh cez prehliadač. Metódu pridáme medzi ostatné metódy v súbore \simple-todos\imports\api\tasks.js.

[3] Tutoriál je dostupný na adrese https://www.meteor.com/tutorials/blaze/creating-an-app

```
'tasks.insertWithSeed'(text, seed) {
    //Kontrola či je používateľ prihlásený
    if (! this.userId) {
        throw new Meteor.Error('not-authorized');
    }
    //vygenerovanie id
    var id = Random.createWithSeeds(seed).id();
    //vloženie úlohy do databázy
    Tasks.insert({
        _id: id,
        text,
        createdAt: new Date(),
        owner: this.userId,
        username: Meteor.users.findOne(this.userId).username,
   });
}
```

Pre testovanie lokálneho vykonávania metód je potrebné povoliť aplikovanie lokálnych zmien na serveri. Meteor to umožňuje pomocou definovania a pridania pravidiel. Pre účely testovania nám stačí jednoducho povoliť všetky zmeny. To vykonáme pridaním nasledujúceho kódu do rovnakého súboru ako predchádzajúcu metódu (súbor \simple-todos\imports\api\tasks.js).

```
//Definovanie povolení pre kolekciu Tasks
Tasks.allow({
    insert: function () {
        return true;                                    //Povolenie pridaní úloh
    },
    update: function () {
        return true;                                    //Povolenie zmien úloh
    },
    remove: function () {
        return true;                                    //Povolenie odstránení úloh
    },
});
```

6.2.2 Štart systému

Pokračujeme návodom k použitiu našej knižnice. Ukážeme použitie základných metód. V ukážkach kódu budeme používať hodnoty a nastavenia odpovedajúce serverovej aplikácii popísanej v prechádzajúcich kapitolách.

Pred vytvorením úložiska mu musíme nastaviť potrebné parametre – minimálne prefix pre kolekcie uložené v databáze. Na tomto mieste je tiež potrebné pridať

kolekcie, ktoré chceme používať. Používa sa na to objekt triedy StorageSetup. S týmito nastaveniami môžeme vytvoriť lokálne úložisko.

```
StorageSetup storageSetup = new StorageSetup();
//nastavenie prefixu pre kolekcie lokálneho úložiska
storageSetup.setPrefix("váš_prefix");
//pridanie kolekcie tasks
storageSetup.addCollection("tasks");
```

Po pripravení týchto nastavení môžeme pridať lokálne úložisko. Pre SQLite verziu lokálneho úložiska je potrebné pridať aj adresu databázy.

```
//vytvorenie lokálneho úložiska
LocalStorage localStorage = new SQLiteLocalStorage(storageSetup,
        "jdbc:sqlite:test.db");
```

Pre vzdialené úložisko komunikujúce cez DDP protokol je potrebné pridať aj webovú adresu a port, pomocou ktorých sa môže pripojiť k serveru.

```
//údaje pre pripojenie
DDPConnectionSettings connectionSettings = new DDPConnectionSettings("localhost",
          3000);
//vytvorenie vzdialeného úložiska
RemoteStorage remoteStorage = new DDPRemoteStorage(storageSetup,
          connectionSettings);
```

Ak sa chceme prihlásiť je potrebné pridať aj spôsob prihlásenia a prihlasovacie údaje. V ďalšom príklade je vybraté prihlásenie pomocou používateľského mena (iné možnosti pre DDP sú pomocou emailovej adresy alebo tokenu).

```
//údaje pre pripojenie
DDPConnectionSettings connectionSettings = new DDPConnectionSettings("localhost",
          3000, Username, "meno", "heslo");
//vytvorenie vzdialeného úložiska
RemoteStorage remoteStorage = new DDPRemoteStorage(storageSetup,
          connectionSettings);
```

Po vytvorení triedy StorageSetup, lokálneho a vzdialeného úložiska môžeme vytvoriť hlavné úložisko a spustiť ho.

```
//vytvorenie hlavného úložiska
Storage storage = new Storage(storageSetup, remoteStorage, localStorage);
//spustenie hlavného úložiska
storage.start();
```

Ak chce používateľ pokračovať v rozrobenej práci a server to podporuje, môže použiť reťazec identifikujúci začatú reláciu (session).

```
storage.start("zadaná session");                    //spustenie so session
```

Po zavolaní metódy `start` začne úložisko pracovať. Vytvoria sa potrebné lokálne kolekcie a vzdialené úložisko sa pokúsi pripojiť k serveru.

Po štarte úložiska sa používateľ môže prihlásiť na odber na server. Použije na to metódu `subscribe` s reťazcom identifikujúcim dátovú sadu, o ktorú má záujem. V našom prípade je to množina plánovaných úloh uložených na serveri (označená „tasks“).

```
storage.subscribe("tasks");                         //prihlásenie sa na odber
```

Po prihlásení sa na odber nám server začne posielať dáta, ktoré sa ukladajú v príslušnej lokálnej kolekcii.

6.2.3 Vykonávanie metód

V knižnici podporujeme vykonávanie dvoch typov metód – vzdialené volanie pomocou metódy `call` (s možnou simuláciou) a lokálne vykonanie s poslaním zmien na server (úložisko ponúka aj možnosť len lokálneho vykonania). Všeobecne sa odporúča používať len jeden zo spomenutých typov na vykonávanie zmien v jednej aplikácii.

Najprv si musíme metódu vytvoriť. Metódu vytvoríme implementovaním metódy run rozhrania `TransactionRunnable`. V metóde run dostaneme k dispozícii argumenty, transakciu a kontext, z ktorého vieme získať vybranú kolekciu. Potom môžeme pomocou kolekcie vytvárať, meniť alebo odstraňovať dokumenty.

```
public class PridanieUlohy implements TransactionRunnable {
    public void run(ExecutionContext context, Transaction transaction,
          Object...  arguments) {
        String uloha = (String) arguments[0];       //získanie argumentu
        //získanie kolekcie tasks
        Collection collection = context.getCollection("tasks");
        Document d = new Document();                //vytvorenie nového dokumentu
        d.setJsonData("{\"text\":\""+uloha+"\"}"); //nastavenie obsahu dokumentu
        d = collection.create(d, transaction);      //zapísanie dokumentu
    }
}
```

Takto vytvorenú metódu môžeme napríklad lokálne vykonať. Pripomíname, že volanie metódy používateľom nevykoná priamo úlohu. Pre dané volanie sa vytvorí požiadavka, ktorá sa pridá do systému a bude vykonaná hneď ako je to možné.

```
storage.execute(new PridanieUlohy(), "text úlohy.");      //vykonanie metódy
```

Špeciálnym prípadom je lokálne vykonanie bez poslania zmien na server. Takéto volanie sa vykonáva pomocou metódy hlavného úložiska `executeLocally`.

Pri vzdialenom volaní stačí jednoducho použiť metódu `call` s názvom metódy ako parametrom. Pre vykonanie simulácie pri vzdialenom volaní metódy musíme najprv túto simuláciu pridať do úložiska. Podstatné je vložiť ju pod rovnakým názvom akým chceme volať. Pridanie simulácie a vzdialené volanie metódy (aj so simuláciou) potom vyzerá takto:

```
//pridanie simulácie
storage.registerSpeculativeMethod("tasks.insertWithSeed", new PridanieUlohy());
//volanie metódy
storage.call("tasks.insertWithSeed", "text úlohy.");
```

Všetky zo spomenutých možností vykonávania metód majú synchrónne a asynchrónne verzie. Spomenuté príklady boli synchrónne volania, asynchrónna verzia sa použije v prípade pridania Async za typ volania metódy (`callAsync`, `executeAsync`, `executeLocallyAsync`).

6.2.4 Získanie dokumentov

Dokumenty z úložiska získame pomocou vykonania metódy, v ktorej zadáme vyhľadanie dokumentov príslušnou kolekciou. Nasledujúce ukážky sa vykonávajú v metóde run implementovaného rozhrania `TransactionRunnable`.

Ako jednoduchý príklad najlepšie poslúži nájdenie jedného ľubovoľného dokumentu pomocou metódy `findOne`.

```
//nájdenie ľubovoľného dokumentu
Document najdenyDok = collection.findOne();
//získanie textu z obsahu dokumentu
System.out("Text najdeneho dokumentu je " + najdenyDok.getPropertyValue("text"));
```

Všetky dokumenty kolekcie získame volaním metódy `find`. Táto metóda vráti objekt s výslednou množinou dokumentov nazývaný `ResultSet`. Najjednoduchší spôsob na prístup k dokumentom je získať z objektu s výslednou množinou dokumentov jednoduchý nemeniaci sa zoznam s dokumentmi.

```
//získanie všetkých dokumentov kolekcie
ResultSet vyslednaMnozina = collection.find();
//získanie zoznamu dokumentov z ResultSet-u
DocumentList zoznamDokumentov = resultSet.getDocuments();
```

Trieda `ResultSet` ponúka viacero spôsobov na získanie dokumentov. Ďalšími sú pridanie pozorovateľov na zmeny alebo získanie zmien ako RxJava Observable. Ukážeme si druhý zo spomenutých príkladov a výpis zmien v pozorovateľovi vytvorenom v lambda výraze.

```
//získanie a prihlásenie sa na RxObservable
resultSet.getAsRxList().subscribe(noveZmeny ->
          //výpis nového zoznamu
          System.out.println("Zmeny: " + noveZmeny.toString()));
```

6.2.5 Použitie indexov a predikátov

Na vyhľadávanie dokumentov podľa filtrovacích dopytov (definované pomocou predikátov) je vhodné najprv ukázať použitie indexov.

Indexy pridávame pri vytvorení systému. Najjednoduchší spôsob je pridať indexy priamo pri pridávaní kolekcie do triedy `StorageSetup`. Interne sa na nastavenie kolekcie a uloženie jej indexov použije trieda `CollectionSetup`, ktorá reprezentuje kolekciu neskôr použitú v systéme. Jednoduché pridanie kolekcie „tasks“ s indexom nad poľom text vyzerá ako v nasledujúcom príklade.

```
String nazovKolekcie = "tasks";
boolean jeLokalna = false;                          //kolekcia bude mať dáta zo servera
String nazovPolaPreIndex = "text";
//pridanie nastavenia kolekcie
storageSetup.addCollectionSetup(nazovKolekcie, jeLokalna, nazovPolaPreIndex);
```

Poznamenávame, že metóda na pridanie kolekcie má parametre: názov kolekcie, určenie či je lokálna a ľubovoľný počet indexov. Indexy nemusíme použiť alebo ich môžeme použiť, či už 1, 2 alebo ľubovoľný počet.

Po nastavení kolekcie tasks s indexom text si ukážeme dopyt používajúci tento index. Dopyty nie sú viazané na indexované polia, ale ich vykonanie je takto efektívnejšie.

```
//(ideálne indexované) pole
String porovnavanePole = "text";
//hodnota porovnaná s hodnotou poľa
String hodnota = "text niektoreho dokumentu";
//vytvorenie podmienky „Equals“
Predicate podmienka = new Predicate.Equals(porovnavanePole, hodnota);
//nájdenie dokumentu so zadaným textom
Document najdenyDocument = collection.findOne(podmienka);
```

6.2.6 Pridanie verzií lokálneho a vzdialeného úložiska

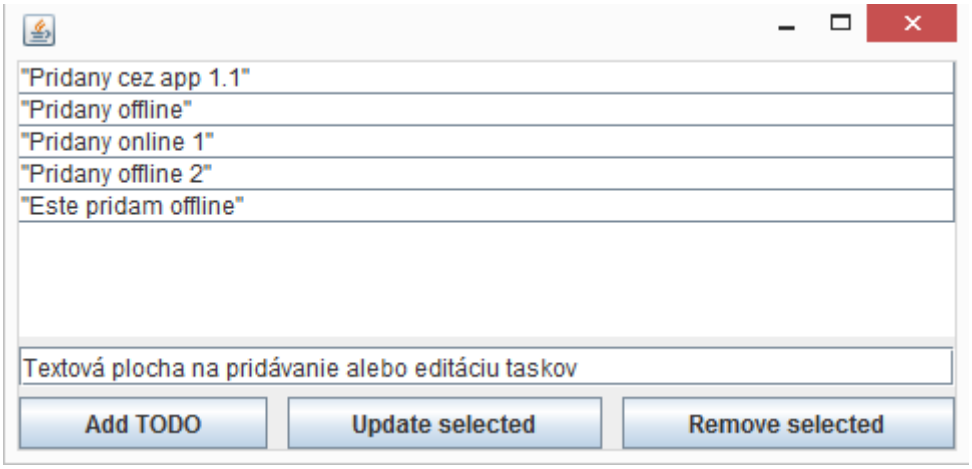
Na záver tutoriálu uvádzame stručné pokyny k implementácii vlastných verzií lokálneho a vzdialeného úložiska.

Vlastné lokálne úložisko vytvoríme rozšírením triedy `LocalStorage` a implementovaním potrebných metód. O vytvorenie potrebných kolekcií sa postará systém, pomocou metód vytvorenej verzie. Používateľ sa môže sám rozhodnúť nakoľko a aké predikáty bude v lokálnom úložisku podporovať vytvorením triedy `PredicateFilter` a použitím predikátov v dopytoch. Podpora indexov je rovnako na používateľovi. Potrebné nastavenia kolekcií s indexami (`CollectionSetup`) získa pomocou triedy `StorageSetup`.

Vzdialené úložisko využívajúce iný protokol na komunikáciu so serverom môžeme vytvoriť rozšíreným triedy `RemoteStorage`. V novej verzii implementujeme jednotlivé metódy a získame pozorovateľa na zmeny (`RemoteStorageListener`), ktorému následne posúvame potrebné správy od servera (nové dáta, potvrdenia volaní metód a ďalšie). Týmto spôsobom sa dostanú údaje do centrálneho a lokálneho úložiska. V niektorých prípadoch je tiež podstatné vytvoriť generátor identifikátorov odpovedajúci spôsobu generovania identifikátorov na serveri.

6.3 Klientska aplikácia

Použitím nami vytvorenej knižnice sme k serverovej aplikácii na plánovanie úloh vytvorili aj jednoduchú klientsku aplikáciu. Klientska aplikácia má demonštrovať použitie knižnice a zároveň sme ju využili na základné testy funkcionality s pripojením aj bez pripojenia k serveru. V jednoduchom okne aplikácia zobrazuje úlohy daného používateľa a umožňuje ich upravovať, odstraňovať alebo pridávať ďalšie.



Obr. 26 Klientska aplikácia použitá ako príklad použitia a otestovanie základnej funkcionality.

Používateľské rozhranie sme implementovali pomocou nástroja Swing. Skladá sa z jedného okna s tabuľkou, v ktorej sa zobrazujú úlohy. Pridávať, meniť alebo odstraňovať úlohy možno pomocou troch tlačidiel a textovej plochy, kde sa zadáva text upravovanej úlohy.

Pri vytvorení aplikácie sa vytvorí lokálne úložisko, ktoré používa jednu kolekciu slúžiacu na zápis úloh, zapisujúce údaje do vstavanej databázy SQLite a vzdialené úložisko s prihlasovacími údajmi používateľa. Obe úložiská sa pridajú do základného úložiska a zavolá sa metóda `start`. Úložisko teraz začne pracovať, lokálne úložisko skontroluje existenciu kolekcie pre úlohy; ak v databáze táto kolekcia neexistuje, vytvorí ju. Vzdialené úložisko sa pokúsi pripojiť sa k serveru. Po úspešnom pripojení a prihlásení zadaného používateľa sa prihlási na odber úloh (pre tohto používateľa).

Po vytvorení komponentov používateľského rozhrania sa pomocou kolekcie s úlohami lokálne vykoná dopyt na nájdenie všetkých dokumentov. Na získaný objekt s výslednou množinou dokumentov sa prihlási pozorovateľ na zmeny. Tento pozorovateľ potom pomocou EDT vlákna aktualizuje dáta v tabuľke vždy podľa najnovšieho výsledku.

Jednotlivé tlačidlá používajú vzdialené volania metód na serveri pre pridanie, zmenu a odstránenie dokumentu. Pridanie dokumentu využíva semienko na priradenie rovnakého identifikátora lokálne a na serveri. Pre pridanie dokumentu používame aj lokálnu simuláciu. V tejto simulácii sa vytvorí úloha (dokument) s rovnakým textom ako na serveri.

Pomocou aplikácie sme pri použití na názornom príklade otestovali funkcionalitu spomenutých operácií – štart systému, samoaktualizujúce sa dopyty, prihlásenie sa na

odber a vzdialené volanie metód. Pomocou webového klienta sme sledovali zosynchronizovanie sa po vykonaní zmien v ľubovoľnom klientovi. Vďaka prihláseniu sa na odber nám automaticky prišli od servera zmeny vykonané druhým klientom.

6.3.1 Správanie sa v offline režime

Pomocou našej aplikácie sme tiež testovali a dolaďovali správanie sa v offline režime. Pri spustení aplikácie počas vypnutého servera sa načítali lokálne dáta. Pridanie dokumentu fungovalo z používateľovho pohľadu rovnako ako pri pripojení na server vďaka lokálnej simulácii. Úprava alebo odstránenie dokumentu nevykonalo nič viditeľné, pokým sa nespustil server. Po spustení servera sa naňho klient automaticky pripojil a poslal čakajúce požiadavky na volania metód, ktoré sa vykonali. Používateľ videl zmenu v tom, že sa vykonali aj operácie bez lokálnej simulácie (operácie zadané počas offline režimu sa vykonali tiež).

Pri vypnutí servera počas už začatej komunikácie, klient pokračoval v práci podľa očakávaní ako v predošlom prípade. Po zapnutí servera sa komunikácia so serverom obnovila, čakajúce metódy sa vykonali a dáta sa zosynchronizovali.

6.4 Ďalšie testy

So serverovou aplikáciou venovanou plánovaniu úloh sme vykonali aj ďalšie testy našej knižnice. Sledovali sme zmenu stavu pripojenia pri pripájaní sa k serveru, prihlásení sa používateľa a pri vypnutí servera. Kontrolovali sme vykonávanie jednotlivých vlákien.

Lokálne vykonanie metódy zapísalo zmeny do databázy a aplikovalo zmeny dát na server. Na serveri sme pre tento účel povolili pridávanie lokálnych zmien. Zmeny boli na serveri správne zapísané a preposlané ostatným klientom (v našom prípade webovému klientovi).

Testovali sme aj vzdialené volanie metódy, so simuláciou aj bez simulácie. Pri použití simulácie sme použili implementovaný generátor a presvedčili sa o identickosti identifikátorov vygenerovaných v simulácii a na serveri.

Samoaktualizujúce sa dopyty sme testovali vykonávaním zmien v lokálnom úložisku, pomocou simulácií a vykonaním zmien vo vzdialenom úložisku. Predikáty použité v dopytoch filtrovali správne výsledné dokumenty.

Otestovali sme aj vytváranie kolekcií a ich odstraňovanie. Kolekcie používajú indexy pri vkladaní aj úprave dokumentov. Indexované polia definované v predikátoch sa pri dopytoch použijú ako sme navrhli.

7 Zhodnotenie

7.1 Možné vylepšenia

Základnú funkcionalitu nášho systému sme podľa návrhu implementovali. Stále však ostáva priestor pre ďalšie vylepšenia hotových častí systému alebo pridanie ďalšej funkcionality. Viaceré z vylepšení spomenutých v nasledujúcej časti už majú vybudovaný základ, na ktorom ich je potrebné len dotvoriť.

7.1.1 Reštart systému

Dokumenty zapísané v databáze sú uložené perzistentne. V aktuálnej verzii však po zastavení a opätovnom vytvorení systému stratíme čakajúce a nepotvrdené požiadavky z radov. Riešením by mohlo byť uložiť obsah radov (vhodne reprezentované čakajúce a nepotvrdené požiadavky) do lokálneho úložiska pri metóde `stop` hlavného úložiska. Pri metóde `start` by sa tieto požiadavky načítali, priradili do príslušných radov a práca by mohla pokračovať od bodu, kde bola prerušená.

7.1.2 Podpora objektového mapovania pre dokumenty

Dokumenty v aktuálnej verzii ukladajú svoj obsah ako textový reťazec, nastavenie a získanie obsahu tiež pracuje s textovým reťazcom. Jedným z možných vylepšení by bolo podporovať aj objektové mapovanie, aby používateľ mohol ukladať a získavať z dokumentov priamo objekty.

7.1.3 Usporiadané výsledky dopytov

Napriek podpore pre usporiadané výsledky dopytov v API a získanie príslušne usporiadaných dokumentov z databázy, usporadúvanie nie je implementované v objekte pre výslednú množinu dopytu. Dorobiť by sa to dalo viacerými spôsobmi. Najjednoduchší spôsob by bol usporadúvanie finálneho výsledku. Väčšou výzvou by bolo aplikovať prichádzajúce zmeny na zdrojové dokumenty tak, aby bol výsledný zoznam usporiadaný.

7.1.4 Nástroj na tvorbu dopytov

V nami vytvorenej knižnici sa na definovanie podmienok dopytov používajú predikáty. Používateľ knižnice má možnosť vytvárať aj vlastné predikáty. Zjednodušenie definovania dopytov by sa dalo docieliť vhodným nástrojom, podľa možnosti rozšíriteľným, aby ho používateľ mohol používať aj s jeho vlastnými predikátmi.

7.1.5 Konflikty

Spôsob akým riešime konflikty je následkom celého návrhu. Princíp, ktorý používame by sa dal jednoducho charakterizovať tým, že server je autoritou. Dáta, prijaté od servera sa priamo zapisujú, prepíšu lokálne zmeny. Riešenie konfliktov na serveri je už priamo v kompetencii servera. Dokumenty automaticky nepoužívajú časové pečiatky alebo číslované verzie podľa úprav, ale vďaka dokumentovému modelu to vývojár používajúci naše riešenie môže jednoducho pri práci s dokumentmi doplniť a použiť na serveri.

7.1.6 Posielanie špekulatívnych metód (simulácií) klientovi zo servera

Pri danej architektúre sa prirodzene núka rozšírenie, týkajúce sa posielania metód určených na vykonanie v lokálnej simulácií serverom klientovi. Je viacero možností ako toto rozšírenie implementovať, pri posielaní spustiteľného kódu od servera klientovi však treba dbať na bezpečnosť. Platí tiež, že už z povahy nášho riešenia nechceme upravovať protokol použitý na komunikáciu so serverom.

Jedným z vhodných spôsobov pre pridanie tejto funkcionality sa zdá byť pridanie súbor typu jar so želanými metódami na server. Klient by potom pri štarte skontroloval, či má lokálne aktuálny jar súbor (na kontrolu sa môže využiť napríklad hash). Ak je lokálny súbor neaktuálny, stiahol by si nový jar súbor od servera a nahradil ním lokálny. Pri tomto riešení by bolo tiež potrebné pamätať na architektúru na akej funguje konkrétny klient, keďže jeden jar súbor nebude fungovať aj v Java aplikácii aj na Android zariadení.

V prípade rozličných metód lokálne a na serveri by síce simulácia spôsobila iné lokálne zmeny dát, ale po príchode dát zo servera sa použijú tieto dáta a zmeny zo simulácie sa zahodia.

7.1.7 Podpora iných verzií servera pomocou vzdialeného úložiska

Jedným z možných vylepšení by bolo vytvoriť verzie vzdialeného úložiska podporujúce iné typy protokolov (a teda aj serverov vytvorených pomocou iných technológií). Príkladom by mohla byť implementácia vzdialeného úložiska komunikujúca pomocou http protokolu s REST serverom. Tento server by si mohol ukladať dáta v štruktúrach odpovedajúcich kolekciám a používať preddefinované filtre tvorené vybranou množinou dát. Takýmto spôsobom by mohol podporovať aj publish-subscribe model, pričom klienti by sa prihlasovali na filtre ponúkané serverom.

7.1.8 Použitie iných databáz u klienta pomocou lokálneho úložiska

Na ukladanie dokumentov u klienta sa ponúka viacero vstavaných databáz. Možným vylepšením by bolo implementovať ďalšie verzie lokálneho úložiska používajúce iné databázy.

Počas výberu databázy pre náš príklad použitia sme porovnávali niekoľko vstavaných databáz, ktoré sú použiteľné aj na mobilnej platforme Android. Konkrétne sme sa zaoberali relačnou databázou SQLite, objektovou databázou Realm a dokumentovou databázou Couchbase Lite. Okrem podpory synchronizácie (spomínanej v kapitole 2) nás zaujímala najmä ich výkonnosť a použiteľnosť API.

Z hľadiska API používajú všetky spomínané riešenia rozdielny prístup. Práca s dokumentmi je u Couchbase Lite z veľkej miery spontánna, definovanie indexov (pomocou view) vyžaduje isté pochopenie alebo skúsenosť s podobným prístupom. Realm využíva objektové mapovanie, rozširovanie nimi definovaných tried a anotácie. Základná prácu to veľmi zjednodušuje, ale môže sa to skomplikovať pri komplexnejších situáciách. S SQLite sa dá pracovať priamo pomocou jazyka SQL použitím rôznych knižníc. Druhou možnosťou sú knižnice s API ako StorIO, ktorá ponúka jednoducho použiteľný nástroj na budovanie a vykonávanie SQL dopytov.

Výkonnosť databáz Couchbase Lite, Realm a SQLite sme testovali 3 spôsobmi: vkladanie údajov, vyhľadávanie podľa identifikátora (primárneho kľúča) a vyhľadávanie zadané nerovnosťou. Entita, s ktorou sme pracovali bola tvorená identifikátorom, dvoma reťazcami a dvoma celými číslami.

Testy sme vykonali na zariadení LG L70 s operačným systémom Android 4.4.2. Použili sme nasledovné verzie testovaných databáz: Realm 0.84.1, Couchbase Lite 1.0.4 a SQLite použitú na danom Android zariadení. V Tab. 1 uvádzame výsledky testov pri použití dátovej sady 10000 údajov a vykonaní 1000 vyhľadávaní, respektíve vkladaní 10000 údajov.

Ako identifikátor sme použili v Couchbase Lite automaticky generovaný reťazec, v SQLite stĺpec s automaticky sa zväčšujúcim číslom. Použitá verzia databázy Realm priamo nepodporovala zabudovaný primárny kľúč, ktorý by sme mohli použiť. Pri pridaní náhodného UUID reťazca pre objekty ukladané do databázy Realm, sa čas vkladania do databázy oproti uvedenému času mierne zvýšil. Pri vyhľadávaní podľa nerovnosti sme pre Couchbase Lite vytvorili aj index (v Couchbase Lite nazývaný view). Čas vytvárania tohto indexu sme do výsledného času nezapočítavali.

My sme pre našu implementáciu vybrali ako perzistentné úložisko SQLite databázu. Hlavné dôvody pre to boli, že je to rozšírené riešenie, dostatočne výkonné a najmä, že SQLite je natívne úložisko v mnohých mobilných zariadeniach.

Tab. 1 Porovnanie času vkladania a vyhľadávania v troch databázach. Časy sú uvedené v sekundách.

	Vkladanie	Vyhľadávanie podľa ID	Vyhľadávanie podľa nerovnosti
Realm	2,57	0,19	0,59
Couchbase Lite	35,01	0,50	13,04
SQLite	3,22	0,85	0,92

Záver

V práci sme sa venovali podpore offline režimu a databázam, špeciálne viacerým NoSQL riešeniam. Na porovnanie výkonu niekoľkých databázových systémov sme vykonali jednoduché testy. Jednotlivé databázy sme tiež porovnávali podľa použiteľnosti API. Osobitne sme sa pri porovnávaní zamerali na podporu offline režimu a synchronizácie s vzdialeným úložiskom.

Zaoberali sme aj spôsobmi implementácie offline režimu a cachovania u rôznych technológií. Rozobrali sme si aktuálne používané spôsoby na synchronizáciu údajov so vzdialeným úložiskom, napríklad publish-subscribe model a optimistic UI.

Na základe vykonanej analýzy sme následne navrhli riešenie podporujúce transparentný offline režim a synchronizáciu so serverovým úložiskom. Naše riešenie sa skladá z 3 častí: hlavného úložiska starajúceho sa o interakciu s používateľom a základnú logiku systému, rozšíriteľného lokálneho a rozšíriteľného vzdialeného úložiska. Lokálne úložisko sa stará o zápis a čítanie dát z databázy, vzdialené o komunikáciu so serverom pomocou vybraného protokolu.

Podľa nášho návrhu sme implementovali jadro systému a verziu riešenia s lokálnym úložiskom používajúcim SQLite a vzdialeným úložiskom komunikujúcim cez DDP protokol. Na otestovanie nášho riešenia sme tiež vytvorili jednoduchý server vo frameworku Meteor a klientsku aplikáciu používajúcu našu knižnicu. V práci je uvedený aj návod k použitiu nášho riešenia.

Stále ostáva priestor pre vylepšenia funkcionality, najmä možnosť pokračovať po reštarte systému od bodu, kde sme prestali a rozšírenie funkcionality pre dopyty a dokumenty. Zaujímavými rozšíreniami by boli aj posielanie metód klientovi a pridanie ďalších implementácií lokálneho a vzdialeného úložiska. Podstatné je tiež spomenúť prispôsobenie a otestovanie nášho riešenia na platforme Android.

Zoznam použitej literatúry

1. **Fortune, Stephen.** A Brief History of Databases. *Avant.org.* [Online] Avant.org, 27. 2 2014. [Dátum: 10. 4 2017.] http://avant.org/project/history-of-databases/.

2. **Scott W., Ambler.** Mapping objects to relational databases. *http://caminotics.ort.edu.uy/.* [Online] 7 2000. [Dátum: 11. 4 2017.] http://caminotics.ort.edu.uy/innovaportal/file/2032/1/mappingobjectstorelationaldatabases.pdf.

3. **Dragland, Åse.** Big Data – for better or worse. *www.sintef.com.* [Online] Sintef, 22. 5 2013. [Dátum: 11. 4 2017.] http://www.sintef.no/en/latest-news/big-data--for-better-or-worse/.

4. **NIST.** The NIST Definition of Cloud Computing. *http://nvlpubs.nist.gov.* [Online] 9 2011. [Dátum: 15. 4 2017.] http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

5. **Stubailo, Sashko.** Optimistic UI with Meteor. *blog.meteor.com.* [Online] 25. 5 2015. [Dátum: 28. 4 2017.] https://blog.meteor.com/optimistic-ui-with-meteor-67b5a78c3fcf.

6. **Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C., & Tomasic, A.** *Scalable query result caching for web applications.* s.l. : Proceedings of the VLDB Endowment, 2008. 1(1), 550-561..

Prílohy

Príloha A: CD médium – diplomová práca v elektronickej podobe, zdrojové kódy knižnice, javadoc, testovacia aplikácia.