VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO ŘÍZENÍ DATOVÉ SYNCHRONIZACE V PROSTŘEDÍ APLIKACÍ APPLE

FRAMEWORK FOR DATA SYNCHRONIZATION IN THE CONTEXT OF APPLE USER APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. FILIP KLEMBARA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MARTIN HRUBÝ, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Klembara Filip, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Knihovna pro řízení datové synchronizace v prostředí aplikací Apple**
Framework for Data Synchronization in the Context of Apple User Applications
Kategorie: Databáze
Zadání:

1. Prostudujte způsoby lokálního a serverového ukládání aplikačních dat v prostředí aplikací pracujících pod operačními systémy iOS, macOS a ipadOS. Prostudujte knihovny CoreData, Realm, CloudKit apod.
2. Navrhněte metodiku synchronizace dat mezi zařízeními v rámci logického celku (např. zařízení jednoho uživatele, zařízení pracovní skupiny apod.). Navrhněte protokol předávání zpráv o událostech v lokálních databázích zapojených zařízení.
3. Implementujte metodiku v podobně knihovny, která bude snadno integrovatelná do uživatelských aplikací. Implementujte centrální prvek systému jako serverovou službu.
4. Knihovnu včetně serverové části testujte se zapojením více zařízení. Zaměřte se na kontrolu integrity dat a provozní efektivitu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- První dva body.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Hrubý Martin, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 31. října 2019

Abstrakt

Cieľom tejto práce je navrhnúť a implementovať knižnicu pre zaistenie synchronizácie dát medzi viacerými klientskými zariadeniami v kontexte ekosystému jednej aplikácie podporujúcej operačný systém iOS, iPadOS alebo MacOS. Práca sa zameriava na jednoduchú integráciu knižnice do existujúcich aplikácií využívajúcich databázový systém Realm a na jednoduché prepojenie používateľského rozhrania so synchronizačným procesom.

Zvolený problém som vyriešil navrhnutím synchronizačného protokolu primárne využívajúceho synchronizáciu založenú na logoch pre distribúciu zmien medzi klientom a serverom, implementovaním knižnice pre monitorovanie zmien v klientskej databáze a ich distribúcie na server a implementovaním serverovej knižnice pre integrovanie prijatých zmien a pre ich rozdistribuovanie pomocou notifikácií.

Vytvorené riešenie poskytuje jednoduchý spôsob ako implementovať synchronizáciu dát medzi viacerými zariadeniami s využitím vlastného databázového serveru a s možnosťou definovania prístupu k novým zmenám počas synchronizácie objektov prezentovaných pomocou používateľského rozhrania. Vytvorenú knižnicu je možné rýchlo a jednoducho integrovať, a tým efektívne zrýchliť vývojový proces aplikácie.

Abstract

The goal of this thesis is to design and implement a library for data synchronization between multiple client's devices in the context of the ecosystem of a single application targeting the iOS, iPadOS or MacOS operating system. The work focuses on the simple integration of the library into existing applications using the Realm database system and on the simple way to connect the user interface with the synchronization process.

I solved the chosen problem by designing a synchronization protocol primarily based on log synchronization to distribute changes between the client and the server, implementing a library to monitor changes in the client's database and distribute them to the server, and implementing a server library to integrate received changes and distribute them with help of notifications.

The solution provides an easy way to implement data synchronization between multiple devices using custom database server and with the ability to define how to handle new changes of objects presented in the user interface during the synchronization. Created library can be quickly and easily integrated and thus effectively speed up the application development process.

Kľúčové slová

Synchronizácia dát, Synchronizácia, Synchronizačný protokol, CoreData, Realm, iCloud, CloudKit, Combine, SwiftUI, MeerkatSync, Swift, Synchronizácia dát v mobilných zariadeniach, iOS, iPadOS, MacOS, Apple

Keywords

Data synchronization, Synchronization, Synchronization protocol, CoreData, Realm, iCloud, CloudKit, Combine, SwiftUI, MeerkatSync, Swift, Mobile data synchronization, iOS, iPadOS, MacOS, Apple

Citácia

KLEMBARA, Filip. *Knihovna pro řízení datové synchronizace v prostředí aplikací Apple.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Hrubý, Ph.D.

Knihovna pro řízení datové synchronizace v prostředí aplikací Apple

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Martina Hrubého, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Filip Klembara
30. mája 2020

Poďakovanie

Chcel by som poďakovať Ing. Martinovi Hrubému, Ph.D. za odbornú pomoc poskytnutú pri riešení tejto diplomovej práce.

Obsah

Kapitola 1

Úvod

Táto práca sa zaoberá podporou pre správu používateľských dát v kontexte ekosystému zariadení od spoločnosti Apple. Vychádza zo situácie, v ktorej používateľ vlastní a používa viaceré zariadenia s operačným systémom iOS, iPadOS alebo MacOS. Na všetkých týchto zariadeniach je možné využívať konkrétnu aplikáciu spravujúcu používateľské dáta. Používateľ predpokladá, že daná aplikácia technicky zaistí, aby sa všetky požadované dáta priebežne rozosielali na jeho ostatné zariadenia tak, aby v dôsledku dochádzalo k permanentnému zrkadleniu (synchronizácii) používateľských dát. Situácia sa komplikuje pokiaľ nie je možné permanentne garantovať sieťové pripojenie.

Spoločnosť Apple neposkytuje vývojárom nástroje priamo umožňujúce jednoduchú implementáciu synchronizácie dát v plnom rozsahu, a teda je potrebné, aby ju vývojár navrhol a implementoval pre každú aplikáciu sám. V tejto práci predkladám softwarovú infraštruktúru pre vývojárov aplikácií, ktorá tento problém rieši. Cieľom a výsledkom predkladanej práce je knižnica MeerkatSync, ktorú je možné jednoducho integrovať do vyvíjanej aplikácie a ktorá úplne automatizuje proces synchronizácie dát v kontexte ekosystému aplikácie.

Vypracované riešenie využíva pre synchronizáciu centrálny prvok (server) držiaci masterkópiu aplikačnej databázy voči ktorej sa jednotlivé zariadenia synchronizujú. Narozdiel od existujúcich riešení, ponúka možnosť mať v plnej správe serverovú databázu, a tým mať možnosť ju priamo integrovať spolu s vlastnou databázou (napr. aby správa používateľov bola iba na jednom mieste). Ďalšou významnou výhodou je otvorené serverové rozhranie založené na REST API, vďaka ktorému je možné využívať synchronizáciu aj z iných aplikácií (napr. webové aplikácie, aplikácie pre android, apod.). Navrhnutý synchronizačný protokol nie je viazaný na žiadny konkrétny databázový systém, a teda je možné ho využívať aj pri iných databázových systémoch. V neposlednom rade ponúka vytvorené riešenie aj rozhranie pre jednoduché napojenie synchronizovaných objektov na používateľské rozhranie s podporou bezpečného prístupu k atribútom uložených objektov, určenia kódu, ktorý sa automaticky zavolá pri zmazaní objektu (napríklad zmena obrazovky pre zabránenie interakcie používateľa so zmazaným objektom), jednoduchej správy skupín používateľov, ich práv a objektov v nich.

Práca je rozdelená celkovo do siedmich kapitol. Základné koncepty vývoja aplikácií pre platformu Apple a existujúce synchronizačné metódy sú popísané v kapitole 2. Návrh synchronizačného protokolu a návrh jednotlivých rozhraní je popísaný v kapitole 3. Dôležité implementačné detaily sú popísané v kapitole 4. Testovanie a overenie funkčnosti je popísané v kapitole 5. Predposledná kapitola 6 sa zaoberá krátkou demonštráciou použitia vytvorenej knižnice v reálnej aplikácii. Zhodnotenie dosiahnutých výsledkov je popísané v záverečnej kapitole 7.

Kapitola 2

Východiská práce

Táto kapitola prepája problematiku programovania mobilných aplikácií využívajúcich lokálnu databázu, napojenie databázy na používateľské rozhranie aplikácie a protokoly pre sieťovú synchronizáciu dát medzi viacerými zariadeniami. V nasledujúcich sekciách sú uvedené základné prvky týchto aspektov. Najprv je popísaný programovací jazyk Swift, ktorý je v komunite programátorov pomerne nový, a preto sú objasnené jeho základné rysy. Následne sú popísané základné koncepty aplikácií pre zariadenia od firmy Apple a možnosti lokálnych databáz v týchto aplikáciach. Posledná sekcia popisuje existujúce spôsoby synchronizácie dát spolu s ich výhodami a nevýhodami.

2.1 Programovací jazyk Swift a funkcionálne programovanie

V roku 2014 prišla spoločnosť Apple s novým programovacím jazykom primárne určeným pre ich platformu. Následne v roku 2015 spoločnosť otvorila kód tohoto jazyka celému svetu a stal sa projektom s otvoreným kódom (open-source). Od svojho vzniku prechádza jazyk pravidelnými zmenami, ktorými sa jeho rozhranie stále zlepšuje a integruje moderné trendy programovacích jazykov.

Podľa oficiálnej stránky s prehľadom jazyka Swift [7] je tento jazyk intuitívny a zameraný na bezpečnosť pri písaní kódu. Taktiež je vhodný ako prvý jazyk pre budúcich programátorov. Jazyk ponúka obmedzenú podporu aj pre iné platformy ako je Linux (Ubuntu, Android) alebo Windows. Táto podpora sa výrazne zlepšuje a smeruje k úplnej kompatibilite jazyka na týchto platformách. Jeho plná podpora je zaručovaná iba na operačných systémoch od spoločnosti Apple. Vďaka silnej podpore jednej distribúcie Linuxu (Ubuntu) sa tento jazyk využíva aj pri tvorbe serverových aplikácií.

2.1.1 Základy jazyka Swift

Syntax jazyka sa odvíja od skupiny jazykov inšpirovaných jazykom C. Bol navrhnutý ako náhrada za jazyk Objective-C. Autor knihy *Mastering Swift 4* [22] vytvoril následovné zhrnutie najzákladnejších výhod jazyka Swift oproti jazyku Objective-C:

- typová inferencia,
- generické programovanie,
- pozmeniteľnosť kolekcií,

- closure syntax (namiesto slovenského prekladu uzáver bude v texte používané slovo closure),
- podpora voliteľných typov (`Optional`),
- podpora konštrukcie `switch`,
- podpora pre dátový typ n-tica (anglicky tuple),
- preťaženie operátorov,
- výčet hodnôt (`enum`),
- protokoly a protokolovo orientované programovanie.

Keďže jazyk Swift využíva veľké množstvo konštrukcií známych z iných jazykov, v nasledujúcom texte sú uvedené len menej známe konštrukcie jazyka potrebné pre pochopenie textu v tejto práci.

Naviazané hodnoty v enum

Narozdiel od väčšiny jazykov s podporou typu `enum` umožňuje jazyk Swift určiť pre každú položku výčtu aj hodnoty, ktoré sú naviazané na ňu. V praxi to znamená, že je možné jednoducho udržiavať hodnoty viazané na konkrétny prípad výčtu priamo v ňom, čím sa sprehľadňuje kód. Vo výpise 2.1 je zobrazený príklad prevzatý z oficiálnej dokumentácie [2] využívajúci túto vlastnosť. Okrem naviazaných hodnôt je enum označený ako `indirect`, čo umožňuje jeho rekurzívnu definíciu.

```
1 indirect enum ArithmeticExpression {
2     case number(Int)
3     case addition(ArithmeticExpression, ArithmeticExpression)
4     case multiplication(ArithmeticExpression, ArithmeticExpression)
5 }
```

Výpis 2.1: Využívanie naviazaných hodnôt v type `enum`. Tento typ umožňuje reprezentáciu aritmetických výrazov tvorených sčítaním a násobením celých čísel. Kód prevzatý z oficiálnej dokumentácie [2].

Pomenovanie parametrov vo funkcii

Pri definícii funkcií môže programátor priradiť každému parametru vnútorné aj vonkajšie meno. Vonkajšie meno musí byť uvedené pred každým argumentom pri volaní funkcie a vnútorným sa pomenováva parameter v rámci funkcie. Programátor má možnosť určiť ako vonkajšie pomenovanie symbol `_` (podčiarkovník) vďaka ktorému nie je potrebné využívať vonkajšie názvy. Vďaka týmto pomenovaniam je možné písať hlavičky funkcií s veľkou výpovednou hodnotou blížiacou sa k bežným vetám ako je zobrazené vo výpise 2.2.

```
1 func send(data: Data) {
2     // send data
3 }
4
5 send(data: Data())
```

```
6
7  func divide(_ a: Int, by b: Int) -> Int {
8      a / b
9  }
10
11 divide(20, by: 5) // 4
```

Výpis 2.2: Príklad pomenovaných parametrov funkcií.

Z pomenovania parametrov funkcie vyplýva aj signatúra funkcie. V jazyku Swift sa v signatúre okrem názvu funkcie objavujú aj vonkajšie názvy parametrov (`send(data:)`, `divide(_:by:)`).

Closures – uzáver nad blokom kódu

V moderných jazykoch sa vyskytujú spôsoby, ako určitý blok kódu zavolať z iného miesta programu. V Pythone sa pre tento účel používajú lambda výrazy, Javascript ponúka možnosť definovať anonymné funkcie. Podobne ako Javascript, ponúka Swift zaobaliť určitý blok kódu do anonymnej funkcie, ale narozdiel od anonymnej funkcie v Javascripte, uzavrie aj kontext prostredia, v ktorom sa closure nachádza.

Jazyk Swift má silnú syntaxovú podporu pre prácu s closures. Pokiaľ funkcia berie ako posledný argument closure, môže programátor tento argument vynechať a za volanie funkcie definovať blok kódu, ktorý bude reprezentovať closure predanú týmto argumentom. Tento spôsob syntaxe sa anglicky nazýva trailing closure syntax.

Closures môžu obsahovať parametre a programátor môže využívať skrátenú syntax parametrov (anglicky shorthand arguments name). Namiesto celých identifikátorov parametrov môže k parametrom pristupovať pomocou špeciálnych premenných zložených zo symbolu $ (dolár) a čísla reprezentujúce index parametru.

Výpis 2.3 zobrazuje využívanie trailing closure syntax a skrátené pomenovanie parametrov pri aplikovaní bloku funkcie na každý prvok v poli.

```
1 [1, 2, 3, 4].map({ (element: Int) -> Int in element * 2 })
2
3 [1, 2, 3, 4].map { (element: Int) -> Int in element * 2 } // trailing
      closure syntax
4
5 [1, 2, 3, 4].map { element in element * 2 } // type inference
6
7 [1, 2, 3, 4].map { $0 * 2 } // shorthand argument names
```

Výpis 2.3: Príklad používania closures v jazyku Swift pre zdvojnásobenie každého prvku v poli.

2.1.2 Funkcionálne programovanie

Väčšina bežne používaných jazykov[1] spadajú pod imperatívne jazyky, nakoľko výpočet programu pozostáva z príkazov, ktoré menia globálny stav programu. Funkcionálne jazyky, ako napríklad Haskell, sa naopak snažia vyjadriť výpočet programu pomocou výrazov, a tým typicky zakryť prípadnú zmenu vnútorného stavu [20]. Swift je imperatívny jazyk,

[1] C, C++, Java, Python, Javascript a podobne.

ktorý ponúka dostatok prostriedkov pre podporu funkcionálneho programovania. Narozdiel od čisto funkcionálnych jazykov, Swift nevynucuje dodržovanie funkcionálnych princípov a ostáva to na programátorovi.

Funkcie

Pojem funkcia vo funkcionálnom programovaní znázorňuje čisto matematickú reprezentáciu funkcie. To znamená, že funkcia môže závisieť iba na hodnote parametrov a nie na prostredí alebo globálnom stave. Inak povedané, je jedno, koľko krát zavoláme jednu konkrétnu funkciu s rovnakými argumentami, výsledná hodnota bude vždy identická.

Nemennosť premenných

Vo funkcionálnom programovaní sa nevyužívajú premenné ale iba konštanty. Kvôli tomuto princípu bežné konštrukcie pre tvorbu cyklov ako je `for`, alebo `while` strácajú význam a musia byť nahradené rekurziou.

Funkcie vyššieho rádu

Doug Galante vo svojom článku [18] označuje každú funkciu, ktorá pracuje s inou funkciou predanou v parametri alebo vracia ako návratovú hodnotu inú funkciu ako funkciu vyššieho rádu. Tieto funkcie sú často využívané vo funkcionálnom programovaní, kde dávajú možnosť jednoduchému spôsobu so spracovaním výrazov. Väčšina jazykov ponúka takéto funkcie pre prácu s kolekciami ako je napríklad triedenie poľa.

V jazyku Swift sa tento princíp bohato využíva vďaka veľkej syntaxovej podpore pri používaní closures. Vo výpise 2.4 je uvedených niekoľko príkladov takýchto funkcií v jazyku Swift.

```
1 [1, 2, 3].map { $0 * $0 } // [1, 4, 9]
2
3 [1, 2, 3].filter { $0 > 2 } // [3]
4
5 [1, 2, 3].flatMap { [$0, $0 * 2] } // [1, 2, 2, 4, 3, 6]
```

Výpis 2.4: Príklad funkcií vyššieho rádu v jazyku Swift.

Používanie takýchto funkcií umožňuje abstraktnejší prístup k problému a umožňuje zaviesť základné abstraktné štruktúry používané vo funkcionálnom programovaní. Vo svojom článku Gio [19] tieto abstraktné štruktúry popísal spolu s príkladmi v jazyku Swift. Menovite sa jedná o:

- funktory,
- aplikatíva (anglicky applicatives),
- monády.

Tieto abstraktné štruktúry ponúkajú funkcie vyššieho rádu pre prácu s ich vnútornými hodnotami. V jazyku Swift vystupujú všetky kolekcie ako funktory aj monády vďaka funkciám `map` a `flatMap`.

2.2 Reaktívne programovanie v jazyku Swift

Reaktívne programovanie je založené na spracovávaní dát v prúdoch, ktoré pomáhajú lepšie dekomponovať logiku transformácie dát, a tým prispievať k znovupoužitelnosti kódu. Základným konceptom je chápať program ako systém s udalosťami, na ktoré sa vykonávajú patričné reakcie. Príkladom môže byť reakcia na stlačenie tlačidla alebo dokončenie asynchrónnej činnosti.

Pri takomto spracovávaní figurujú v systéme tri základné komponenty. Komponent zodpovedný za posielanie dát (publisher), komponent zodpovedný za prijímanie dát (subscriber) a treťou súčasťou sú komponenty zodpovedné za zmenu dát, ktoré je možné za seba spájať, a tým riadiť transformáciu dát (operator). Takéto spojenia často pripomínajú trúbky, a preto sa anglicky tento spôsob nazýva pipelining[2].

V roku 2019 vydala firma Apple novú knižnicu zameranú na takéto spracovávanie dát – *Combine*. Princíp tejto knižnice aj s jej možnosťami je popísaný v knihe od Josepha Hecka [21]. V knihe sa autor venuje aj princípu riadenia tohoto toku. Combine riadi celý tok dát zo strany subscribera, čo znamená, že hodnoty by mali byť počítané až vtedy, kedy je o nich záujem[3].

2.2.1 Subscriber

Subscriber bežne predstavuje koniec jedného toku dát. Ako bolo spomenuté, celý proces je riadený práve týmto komponentom, a teda je zodpovedný za posielanie požiadaviek svojmu publisherovi (začiatok toku dát). Požiadavka obsahuje, konkrétne aké množstvo dát očakáva a publisher by nemal zaslať viac, ako sa od neho očakáva.

2.2.2 Publisher

Na základe požiadaviek od subscribera sa poskytuje požadované množstvo dát. Pokiaľ subscriber očakáva nejaké dáta a sú v publisherovi prístupné, tak sa okamžite odosielajú. Príkladom môže byť generátor Fibonacciho postupnosti alebo asynchrónne spracovávanie dát.

2.2.3 Operator

Operátor predstavuje spojenie publishera so subscriberom za účelom transformácie dát alebo pre kontrolu toku dát.

2.2.4 Prísľub hodnoty v budúcnosti

V jazyku Swift sa pre asynchrónne spracovávanie bežne využíva princíp takzvaných callbackov. Jedná sa o určenie funkcie, ktorá sa má spätne zavolať po dokončení asynchrónnej činnosti – callback. Vadim Bulavin vo svojom článku [11] určuje dva veľké problémy takéhoto prístupu:

- asynchrónne programovanie s využívaním callbackov je ťažké udržiavať,
- porušuje inverziu kontroly.

[2] Aj napriek podobnosti sa v oficiálnej dokumentácii nikde pojem pipeline neobjavuje.
[3] Princíp je podobný lenivému vyhodnocovaniu výrazov (lazy evaluation).

Ďalej vo svojom článku popisuje jeden z princípov ako pristupovať k asynchrónnemu vykonávaniu kódu známy už dlhé roky – Future and Promises. Future reprezentuje kontext pre hodnotu, ktorá nemusí ešte existovať. Promise je prísľub hodnoty pre Future. Jedná sa o oddelenie samotného výpočtu od hodnoty. Keď sa stane hodnota prístupnou (skončí asynchrónny výpočet), môže byť použitá v ďalšom výpočte. Reťazenie viacej výpočtov s využitím tohoto princípu vedie k prúdovému spracovávaniu dát.

Vo výpise 2.5 je zobrazené asynchrónne spracovávanie bežným spôsobom a s využitím Future and Promises.

```
1  // Callbacks
2  URLSession.shared.dataTask(with: url) { data, response, error in
3      if let error = error {
4          deal(with: error)
5      } else {
6          URLSession.shared.dataTask(with: anotherURL) { data, response,
               error in
7              if let error = error {
8                  deal(with: error)
9              } else {
10                 URLSession.shared.dataTask(with: oneMoreURL) { data,
                       response, error in
11                     // do work
12                 }
13             }
14         }
15     }
16 }
17
18 // Combine Future
19 URLSession.shared.dataTaskPublisher(for: url)
20     .flatMap { data, response in
21         URLSession.shared.dataTaskPublisher(for: anotherURL)
22     }
23     .flatMap { data, response in
24         URLSession.shared.dataTaskPublisher(for: oneMoreURL)
25     }
26     .sink(receiveCompletion: { ... },
27           receiveValue: { ... })
```

Výpis 2.5: Porovnanie prístupu k asynchrónnej komunikácii so serverom. Prvý kód je bežný prístup a druhý s použitím Future and Promises. Kód je prebraný z článku od Vadima Bulavina [11].

2.3 Základy vývoja aplikácií určených pre operačné systémy iOS, iPadOS a MacOS

Aplikácie pre zariadenia od firmy Apple zdieľajú základné princípy, ako je životný cyklus, ich architektúra alebo prístup k notifikáciám. Táto sekcia sa venuje práve týmto základom, nakoľko od ich pochopenia sa odvíja výsledná implementácia riešenia.

2.3.1 Životný cyklus aplikácií

Každá aplikácia sa môže nachádzať v určitých stavoch voči systému. V oficiálnej dokumentácii [3] sú definované presné činnosti, ktoré môže aplikácia vykonávať na základe tohoto stavu. Ak je aplikácia viditeľná, môže reagovať na používateľa. Pokiaľ viditeľná nie je (je v pozadí), tak by aplikácia mala vykonávať čo najmenej výpočtov. Stav určuje aj prioritu k zdrojom ako je napríklad využitie procesoru. Aplikácia môže sledovať zmenu stavu na základe notifikácií prijatých objektami typu `UISceneDelegate` a `UIApplicationDelegate`.

Základné stavy

Aplikácia po zapnutí postupne prechádza do popredia a stáva sa aktívnou. Pokiaľ je aplikácia aktívna, tak je používateľovi zobrazené grafické rozhranie. V aplikáciach pre iOS 13 a iPadOS 13 sa jednotlivé okná so samostatne riadeným používateľským rozhraním nazývajú scény. Pokiaľ používateľ minimalizuje aplikáciu, tak sa presunie do pozadia. V pozadí ostáva, dokým sa opäť nedostane do popredia alebo sa operačný systém nerozhodne ukončiť jej beh.

Ak aplikácia podporuje scény, tak jednotlivé notifikácie o zmene stavu pre každú scénu sú doručované objektu typu `UISceneDelegate`. Životný cyklus scény je zobrazený na obrázku 2.1.

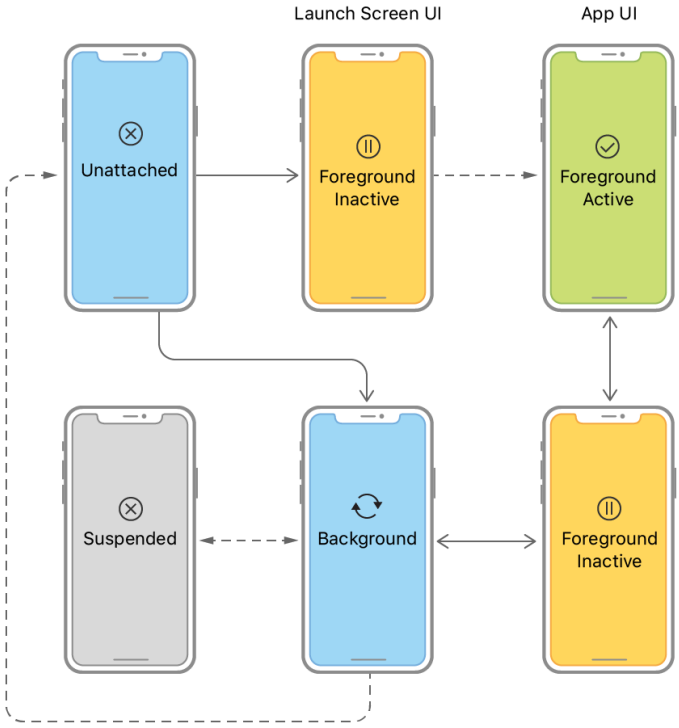
MacOS aplikácie a aplikácie so staršou verziou iOS dostávajú notifikácie iba pomocou `UIApplicationDelegate`. Životný cyklus je zobrazený na obrázku 2.2. Aplikácie podporujúce scény využívajú `UIApplicationDelegate` pre zachytenie notifikácií týkajúcich sa aplikácie ako celku (napríklad štart aplikácie alebo prijatie notifikácie).

2.3.2 Architektúra aplikácií a dátový model

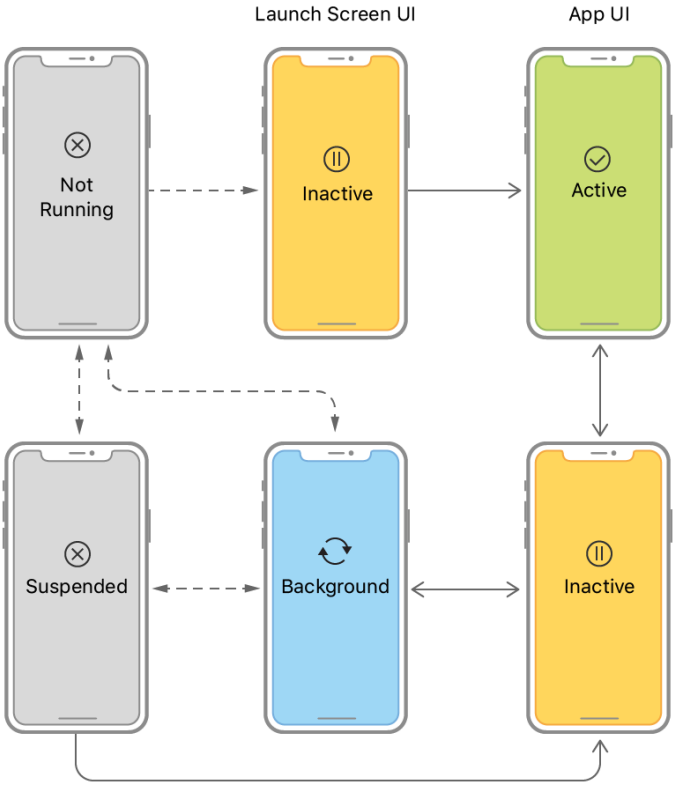
Pri programovaní aplikácií je potrebné vybrať vhodnú architektúru. Oficiálne knižnice od spoločnosti Apple priamo podporujú dve architektúry: Model-View-Controller (MVC) a Model-View-ViewModel (MVVM).

Model-View-Controller

Architektúra MVC sa v Apple zariadeniach vyskytuje už od začiatku. Podpora je založená na knižnici UIKit určenej pre iOS a Cocoa určenej pre MacOS. V týchto knižniciach sa nachádzajú triedy a protokoly určené pre správne rozloženie logiky modelu a používateľského rozhrania do samostatných modulov. Základnou súčasťou architektúry je kontrolér slúžiaci ako most medzi dátovým modelom a používateľským rozhraním aplikácie. Predimplementované kontroléry využívajú princíp delegátstva k určeniu objektu (Model) zodpovedného za dáta zobrazované pomocou používateľského rozhrania. Objekt zobrazujúci používateľské rozhranie (View) notifikuje kontrolér pri používateľskej akcii a prípadne pomocou určeného delegáta zobrazuje dáta z modelu.



Obr. 2.1: Životný cyklus jednotlivých scén. Obrázok prevzatý z oficiálnej dokumentácie [3].



Obr. 2.2: Životný cyklus aplikácie. Obrázok prevzatý z oficiálnej dokumentácie [3].

Model-View-ViewModel

MVVM do polovice roku 2019 nemalo priamu podporu z oficiálnych knižníc. S príchodom knižnice SwiftUI zameranej priamo na túto architektúru a deklaratívny princíp programovania sa použitie tejto architektúry stalo vo veľa prípadoch efektívnejšou metódou ako MVC. Narozdiel od MVC tu nefiguruje žiadny kontrolér. Užívateľské rozhranie (objekt typu View) využíva dvojcestné previazanie (anglicky binding) atribútov s objektom držiacim stav aplikácie (ViewModel) alebo prípadne priamo s dátovým objektom (Model). Pri zmene previazaného atribútu sa zmena okamžite prejavuje vo všetkých objektoch s previazanou hodnotou.

Vzťah architektúry s lokálnou a vzdialenou databázou

Obidve zmienené architektúry majú jasne vymedzený vzťah medzi dátovým modelom a zvyškom aplikácie. Práca s modelom pôsobí pre všetky časti aplikácie ako práca s lokálnymi dátami. Ďalší vzťah vzniká medzi modelom a vzdialenou databázou. Model sa stará o aktuálnosť dát a ich synchronizáciu s ďalšími zariadeniami alebo serverom a ich distribúciou pomocou delegátstva alebo previazanými atribútmi do zvyšku aplikácie.

2.3.3 Obsluha vzdialených notifikácií

Notifikácie slúžia k informovaniu aplikácie (alebo používateľa) o špeciálnych udalostiach. Aplikácia môže prijímať lokálne a vzdialené notifikácie. Lokálne sú len v rámci aplikácie a vzdialené sú doručované aplikácii s využitím serverovej služby. Pomocou vzdialených notifikácií je možné informovať aplikáciu o zmenách v zdieľanej databáze, o novej správe, o potrebe načítať nejaké dáta a podobne.

Priorita notifikácie

Vzdialené notifikácie sa rozdeľujú na prioritné a menej prioritné. Kľúčovým faktorom je účel obsahu danej notifikácie. Pokiaľ notifikácia má zobraziť používateľovi hlásenie, je doručovaná skoro okamžite s vysokou prioritou. V inom prípade je označená ako *background* notifikácia. Podľa oficiálnej dokumentácie [5] nie je zaručené, že takáto notifikácia vôbec bude doručená a v prípade, že sa majú doručiť viaceré naraz, tak môže byť doručená iba posledná.

Odosielanie notifikácie

Za odosielanie notifikácií je zodpovedná serverová služba spoločnosti Apple (APNS[4]). Vývojári môžu tejto službe zadávať požiadavky s obsahom notifikácie, ktorú má server odoslať na konkrétne zariadenie.

Požiadavky je možné zadávať pomocou HTTP metódy POST s použitím autentifikácie. Telo požiadavky sa skladá z dát vo formáte JSON, z ktorých bude notifikácia vykladaná. Vo výpise 2.6 je zobrazené telo menej prioritnej požiadavky s vývojárom definovanými atribútmi `acme1` a `acme2`.

Stránka zameraná na odosielanie notifikácie z oficiálnej dokumentácie [6] určuje aj limitáciu maximálnej veľkosti tela požiadavky, ktorá nesmie prekročiť veľkosť 4096 bytov[5] a telo nesmie byť komprimované.

[4] Apple Push Notification Service

[5] Pokiaľ je typ notifikácie VOIP tak je maximálna veľkosť 5120 bytov.

```
{
        "aps" : {
               "content-available" : 1
        },
        "acme1" : "bar",
        "acme2" : 42
}
```

Výpis 2.6: Telo požiadavky na vytvorenie background notifikácie. Kód prevzatý z oficiálnej dokumentácie [5].

Prijatie notifikácie

Ako uvádza oficiálna dokumentácia [5], background notifikácie sú doručované iba ak je aplikácia zapnutá. Pokiaľ aplikácia nebeží, sú všetky notifikácie pre ňu automaticky zahodené. Pokiaľ je aplikácia neaktívna, tak môže systém pozdržať notifikácie a doručiť z nich iba poslednú, a to v čase, ktorý sám uzná za vhodný. Ak aplikácia prejde do popredia, tak je notifikácia okamžite doručená a aplikácia môže reagovať.

Notifikácia je obdržaná objektom typu `UIApplicationDelegate`. Pokiaľ bola aplikácia neaktívna tak sa presunie do pozadia a má tridsať sekúnd na dokončenie všetkých akcií, aké potrebuje. Okrem notifikácie je argumentom predaný blok kódu, ktorého zavolanie indikuje koniec obsluhy notifikácie. Oficiálna dokumentácia [1] uvádza, že obsluha každej notifikácie je sledovaná operačným systémom. Systém sleduje čas obsluhy, energetickú náročnosť a cenu dát, ktoré sú stiahnuté pri obsluhe notifikácie. Na základe týchto faktorov systém rozhoduje, ako často bude aplikácii doručená notifikácia v budúcnosti.

Z uvedených informácií vyplýva, že firma Apple stále kladie vysoké nároky na kvalitu aplikácií a vývojár musí myslieť na efektivitu aj pri obsluhe notifikácií.

2.4 Databázové systémy v zariadeniach od spoločnosti Apple

Potreba uchovávať dáta v aplikácii aj po jej vypnutí viedla k tvorbe niekoľkých databázových systémov. Kým správny výber databázového systému môže prácu s aplikáciou urýchliť, nevhodný databázový systém alebo zlá práca so systémom môže prácu s aplikáciou naopak spomaliť. Na zariadeniach s operačnými systémami od Apple sa ako databázový systém používa hlavne SQLite, CoreData a Realm [24].

2.4.1 SQLite

Oficiálna dokumentácia [33] popisuje SQLite ako všeobecne najviac používaný databázový systém s otvoreným kódom. Je to malý, rýchly a vysoko spoľahlivý SQL engine priamo vstavaný vo všetkých mobilných a vo väčšine počítačových platformách. Práca s SQLite je na zariadeniach od spoločnosti Apple pomocou rozhrania v jazyku C portnutého do jazyka Swift [12]. SQLite je možné jednoducho používať vďaka nulovej potrebe predchádzajúcej konfigurácie. Umožňuje bezpečnú prácu s dátami z viacerých procesorov a vlákien a upravovať a mazať dáta bez potreby načítať ich do pamäte [24].

2.4.2 CoreData

Narozdiel od SQLite sa nejedná o databázu ale o perzistentný databázový kontajner a framework. Oficiálna dokumentácia [9] uvádza, že umožňuje ušetriť 50%-70% kódu nutného pre prácu s modelovou vrstvou. Má automatickú podporu pre uchovávanie dát v externých repozitároch, sledovanie zmien s možnosťou vrátenia zmeny (undo), znovu vykonaním zmeny (redo) a nástroje na zjednodušenie migrácie medzi rôznymi schémami databáz. Podporuje copy-on-write sémantiku pre zníženie záťaže pri zdieľaní dát a lazy loading objektov. Namiesto SQL príkazov umožňuje vytvoriť komplikované databázové požiadavky (anglicky: queries) s využitím predikátov.

Z predchádzajúcich vlastností vychádzajú aj najväčšie rozdiely medzi SQLite a CoreData. Kým SQLite podporuje obmedzenia na databázovej vrstve (anglicky constraints), CoreData musí obmedzenia riešiť na vyššej úrovni. SQLite umožňuje prácu s dátami bez potreby načítania ich do pamäte. CoreData naopak musí načítať všetky dáta do pamäte predtým, ako sa upravia alebo zmažú. CoreData ponúka rýchlejšiu prácu s dátami, ale zaberá viac miesta na disku a, pochopiteľne, aj v pamäti [36, 24]. CoreData umožňuje ukladať dáta v rôznych formátoch. Skoro vždy sú dáta ukladané v SQLite [23], ale je možnosť ukladať dáta v binárnej forme a dokonca aj v jazyku XML [8].

Vďaka skvelej integrácii CoreData do operačných systémov od firmy Apple je podľa Hudsona [23] systém CoreData často používaný vývojármi, o čom svedčí aj približný počet 500 000 aplikácii v App Store využívajúcich CoreData.

Databázový kontajner

Pred prácou s databázou pomocou CoreData je potrebné vytvoriť databázový kontajner. Celý postup sa dá zhrnúť do niekoľkých bodov. Ako prvé je potrebné načítať dátový model z aplikácie a vytvoriť objekt typu `NSManagedObjectModel`, ktorý podľa dokumentácie [4] reprezentuje schému všetkých objektov v databáze. V ďalšom kroku treba vytvoriť objekt typu `NSPersistentStoreCoordinator` zodpovedný za zápis a čítanie z disku. Po jeho načítaní sa určí URL s umiestnením databázy, kde sú umiestnené všetky dáta. Pokiaľ databáza na danej URL neexistuje, tak sa vytvorí. V poslednom kroku sa vytvorí objekt typu `NSManagedObjectContext` odkazujúci na coordinator vytvorený v predchádzajúcich krokoch. Vzťahy medzi jednotlivými objektmi sú zobrazené na obrázku 2.3. Po vykonaní týchto krokov je možné pracovať s databázou. Celý tento proces, ktorý môže zaberať až dvadsať riadkov kódu, je možné zredukovať na šesť riadkov za použitia objektu typu `NSPersistentContainer`. Na ukážke kódu vo výpise 2.7 je zobrazený skrátený zápis inicializácie databázového kontajnera.

```
1 let container = NSPersistentContainer(name: "dbname")
2 container.loadPersistentStores { storeDescription, error in
3     if let error = error {
4         print("Unresolved error \(error)")
5     }
6 }
```

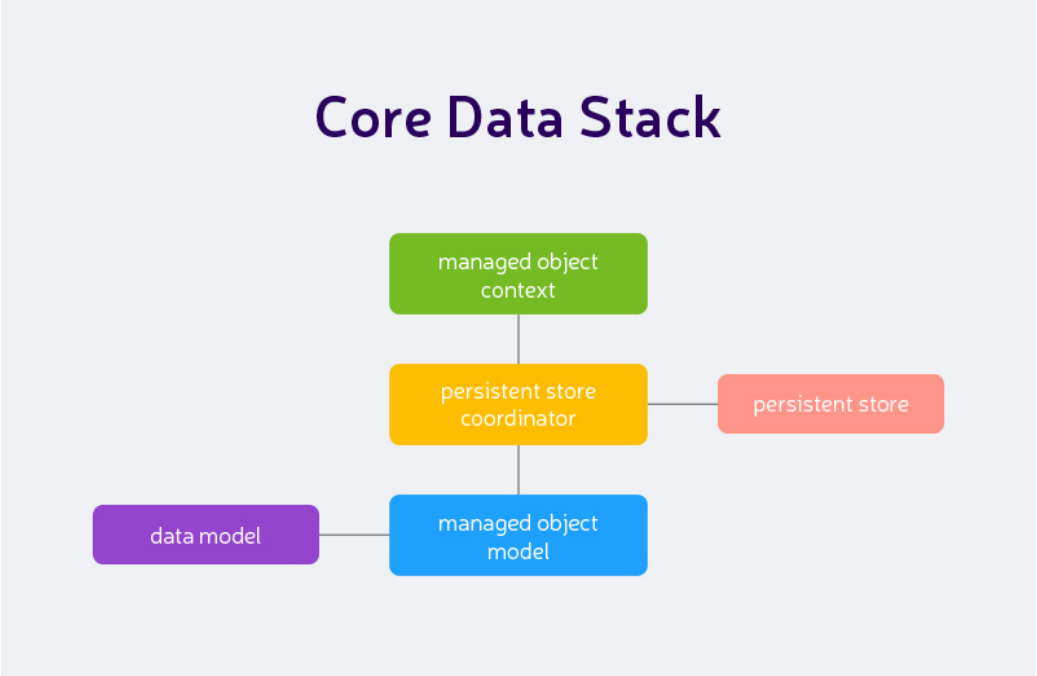
Výpis 2.7: Inicializácia databázového kontajnera nad databázou s názvom *dbname*.

Po vytvorení kontajnera je dôležité pridať funkciu pre ukladanie zmien objektov v pamäti späť na disk. Pre toto poskytuje `NSPersistentContainer` property `viewContext` (nakoľko anglické slovo property nemá vhodný preklad do slovenčiny so zachovaním správneho

významu, bude sa používať v následujúcom texte nepreložené) odkazujúcu na `NSManagedObjectContext` reprezentujúci databázu načítanú v pamäti. Pred zavolaním metódy `save` ukladajúcej zmeny na disk je vhodné skontrolovať či vôbec nejaké zmeny nastali. Ukážkový kód je možné vidieť vo výpise 2.8.

```
1 func saveContext() {
2     if container.viewContext.hasChanges {
3         do {
4             try container.viewContext.save()
5         } catch {
6             print("An error occurred while saving: \(error)")
7         }
8     }
9 }
```

Výpis 2.8: Funkcia zapisujúca zmeny objektov v pamäti na disk. Pred samotným zápisom na riadku štyri sa na riadku dva kontroluje, či vôbec nastali nejaké zmeny.



Obr. 2.3: Vzťahy medzi objektmi v CoreData. Obrázok prevzatý z článku od Helen Vakhnenko a Michaela Liptuga [35].

2.4.3 Realm

Databázový systém Realm je framework rozšírený prevažne medzi mobilnými aplikáciami. Vďaka jeho vnútornej implementácií v jazyku C++ ho bolo možné sprístupniť pre viaceré jazyky. Momentálne je možné bezproblémovo využívať Realm v jazykoch Java, Javascript, Swift, Objective-C a na platforme Xamarin. Podporuje bezpečnú prácu na viacerých vláknach, šifrovanie, notifikácie o zmenách dát v lokálnej databáze a má v sebe zabudovanú aj synchronizáciu objektov medzi lokálnou a serverovou databázou. Pre použitie synchronizácie musí byť použité cloudové rozhranie a nie je možné použiť vlastný typ databázy.

Hlavným zdrojom informácií použitých v následujúcom texte tejto podsekcie je oficiálna dokumentácia frameworku Realm [31].

Model databázy

Pre definovanie modelu sa vo frameworku Realm používajú bežné triedy jazyku Swift. Každá takáto trieda musí dediť z preddefinovanej triedy `Object`. Narozdiel od bežnej triedy musí byť väčšina atribútov v týchto triedach označených ako `@objc dynamic`. Jedinú výnimku majú atribúty s typom `List`, `LinkingObjects` a atribúty označené iba pre čítanie. Podporované sú všetky základné typy (`Bool, Int, String, Data, ...`) a typ `List` pre podporu kolekcie typu `Array`.

Realm podporuje základné vzťahy medzi objektami a to many-to-one a many-to-many. Pre lepšiu prácu s obojsmernými väzbami je určený typ `LinkingObjects`, ktorý zaručuje správnu inverznú väzbu.

Vo výpise 2.9 je model dvoch tried. Inštancia triedy reprezentujúcej osobu (`Person`) môže vďaka atribútu `dogs` vlastniť niekoľko psov (`Dog`) . Pre získanie osoby, ktorej daný pes patrí, sa využíva automatická inverzná väzba.

```
1 class Dog: Object {
2     @objc dynamic var name = ""
3     @objc dynamic var age = 0
4     let owners = LinkingObjects(fromType: Person.self, property: "dogs")
5 }
6
7 class Person: Object {
8     let dogs = List<Dog>()
9 }
```

Výpis 2.9: Ukážka modelu s využitím Realm.

Vytvorenie, úprava a zmazanie objektov

Každá zmena objektu v databáze vrátane jeho pridania musí byť vykonaná v databázovej transakcii. Pre tento účel slúži funkcia `write`, prípadne `beginTransaction`. Vďaka tomuto obmedzeniu je každý objekt spravovaný systémom Realm automaticky aktualizovaný po každej zmene v databáze.

Objekty je možné inštanciovať obdobne ako bežné objekty v jazyku Swift. Každý z týchto objektov musí byť možné inštanciovať bez argumentov (vďaka dedeniu z triedy `Object`) alebo za použitia jediného atribútu s typom pole alebo asociatívne pole. Okrem týchto predvolených konštruktorov je možné zadefinovať aj vlastné. Inštanciovaný objekt (`newObject`) je možné pridať do databázy a správy systému Realm pomocou volania funkcie `add(newObject)`.

Zmena atribútov sa vykonáva rovnako ako v bežných objektoch, pridelením novej hodnoty do atribútu objektu. Každý objekt ponúka možnosť pristupovať k nemu ako k asociatívnemu poľu a nastavovať atribúty dynamicky.

Zmazať objekt je možné iba ak je spravovaný Realmom. Volaním funkcie `delete(objectToDelete)` sa objekt `objectToDelete` odstráni z databázy a jeho atribút `isInvalidated` sa nastaví na hodnotu `true`. Atribút slúži k overeniu, či objekt je platný alebo už nie.

Výpis kódu 2.10 ukazuje popísané funkcie.

```
let realm = try Realm()

let person = Person()
let dog = Dog()

dog.name = "Barky"
dog["age"] = 3  // access atributes with subscript

try realm.write { realm in
    realm.add(person)  // save person
    realm.add(dog)    // save dog

    person.dogs.append(dog) // update person

    realm.delete(dog) // delete dog (also update 'dogs' attribute in
        person)
}
```

Výpis 2.10: Manipulácia s objektmi v databáze Realm. Využívajú sa triedy z výpisu 2.9.

Tvorba dotazov

Dotazy v databáze vracajú dátový typ `Results`, ktorý reprezentuje kolekciu objektov v databáze. Tento typ ponúka funkciu pre filtrovanie výsledkov alebo ich zoradenie. Pri filtrácii sa využívajú predikáty typu `NSPredicate`, ktoré sa používajú aj v systéme CoreData. Realm umožňuje reťazenie podmienok použitých pri filtrácii ako je zobrazené vo výpise 2.11. V tomto výpise sú zobrazené aj spôsoby ako je možné výsledky filtrovania zotriediť.

```
let dogs = realm.objects(Dog.self) // all dogs

let puppies = dogs.filter("age < %@", 1) // all dogs younger a year
let tobys = puppies.filter("name = 'Toby'") // all puppies with the name
     Toby

let sorted = tobys.sorted(byKeyPath: "name")
```

Výpis 2.11: Ukážka dotazov v Realme. Využívajú sa triedy z výpisu 2.9.

Databázové notifikácie

Realm ponúka tri možnosti sledovania zmien v lokálnej databáze. Tieto zmeny sú publikované vo forme notifikácií. Každý spôsob sleduje zmeny v databáze na inej úrovni.

- Najvšeobecnejší spôsob sleduje celú databázu a je možné pomocou tejto notifikácie zistiť iba či sa databáza zmenila, ale neumožňuje zistiť konkrétne zmeny v nej.
- Druhý spôsob sleduje zmeny v rámci kolekcie objektov rovnakého typu. Táto kolekcia môže byť získaná aj použitím funkcie `filter` alebo `sorted`. Každá zmena tejto kolekcie vyvolá notifikáciu pomocou, ktorej je možné zistiť zmeny. Pri zmene sa v notifikácii nachádzajú tri polia indexov spolu s touto kolekciou. Na základe indexov je

možné určiť, ktoré objekty boli pozmenené a ktoré vytvorené. Keďže kolekcia je už aktualizovaná, tak je možné určiť iba indexy objektov, ktoré boli zmazané. Pomocou tejto notifikácie nie je možné určiť, ktoré atribúty boli zmenené a ani ich staré hodnoty.

- Posledný spôsob sleduje zmeny nad jedným konkrétnym objektom. V týchto notifikáciách sa nachádzajú konkrétne zmeny nad objektom, a teda je možné určiť, aké atribúty sa zmenili a aj ich pôvodné a nové hodnoty.

Pre sledovanie zmien v databáze v tejto práci je najvhodnejší druhý spôsob (notifikácie pre zmenu objektov v kolekcii) zobrazený vo výpise 2.12, nakoľko sledovanie zmien v celkovej databáze nedáva informácie o objektoch a sledovať zmeny pre každý objekt zvlášť by mohlo byť neefektívne. Týmto spôsobom ale nie je možné určiť, ktoré objekty sú zmazané a ani aké atribúty boli zmenené. Riešenie týchto problémov je uvedené v sekcii 3.3.

Sledovanie zmien je možné spustiť pomocou funkcie `observe` nad požadovaným dátovým typom a uschovať si notifikačný token z návratovej hodnoty tejto funkcie. Pre ukončenie sledovania je potrebné zavolať funkciu `invalidate` nad vráteným notifikačným tokenom.

```
1  notificationToken = puppies.observe {
2          [weak self] (changes: RealmCollectionChange) in
3
4      switch changes {
5                  case .initial:
6                  // Results are now populated and can be accessed without
                       blocking the UI
7                  case let .update(collection, deletions, insertions,
                       modifications):
8                  // Query results have changed, so apply them
9                  // Always apply updates in the following order: deletions,
                        insertions, then  modifications.
10                 case .error(let error):
11                 // An error occurred while opening the Realm file on the
                       background worker thread
12         }
13 }
14 notificationToken.invalidate() // stop observing
```

Výpis 2.12: Sledovanie zmien v kolekcii psov s vekom menším ako jeden rok.

2.4.4 iCloud

Apple ponúka službu iCloud s možnosťou uloženia, upravenia, zmazania a zdieľania dát. Používanie tejto služby je striktne viazané na platformu Apple a jediný spôsob prístupu k dátam z inej platformy je možný jedine cez webové rozhranie[6] alebo s použitím Javascriptovej knižnice.

Pre prácu s touto službou slúži knižnica CloudKit, ktorú je možné využiť so systémom CoreData a aj so systémom Realm[7] pre synchronizáciu dát medzi viacerými klientskými

[6] https://icloud.com

[7] Prepojenie frameworku Realm so službou iCloud je možné s využitím knižnice IceCream dostupnej na https://github.com/caiyue1993/IceCream

zariadeniami. CoreData je navyše možné prepojiť s privátnym úložiskom v iCloude pomocou objektu typu `NSPersistentCloudKitContainer`. Týmto spôsobom sa výrazne zjednoduší napojenie CoreData na CloudKit a zaistenie synchronizácie. Nakoľko je `NSPersistentCloudKitContainer` pomerne nový, obsahuje niekoľko chýb, na ktoré upozornil aj Andrew Bancroft vo svojom článku [10]. Jednou zo zásadných chýb, ktoré sa v článku spomínajú, je nespoľahlivosť synchronizácie, ktorá sa môže pozastaviť a pre jej znovuspustenie je potrebné zavrieť a znova otvoriť aplikáciu.

Firma Apple ponúka pre aplikácie určitú veľkosť databázy zdarma. Táto veľkosť sa odvíja od počtu používateľov aplikácie a je možné ju za určitú finančnú sumu rozšíriť podľa potreby.

2.4.5 Porovnanie riešení synchronizácie CoreData a Realm

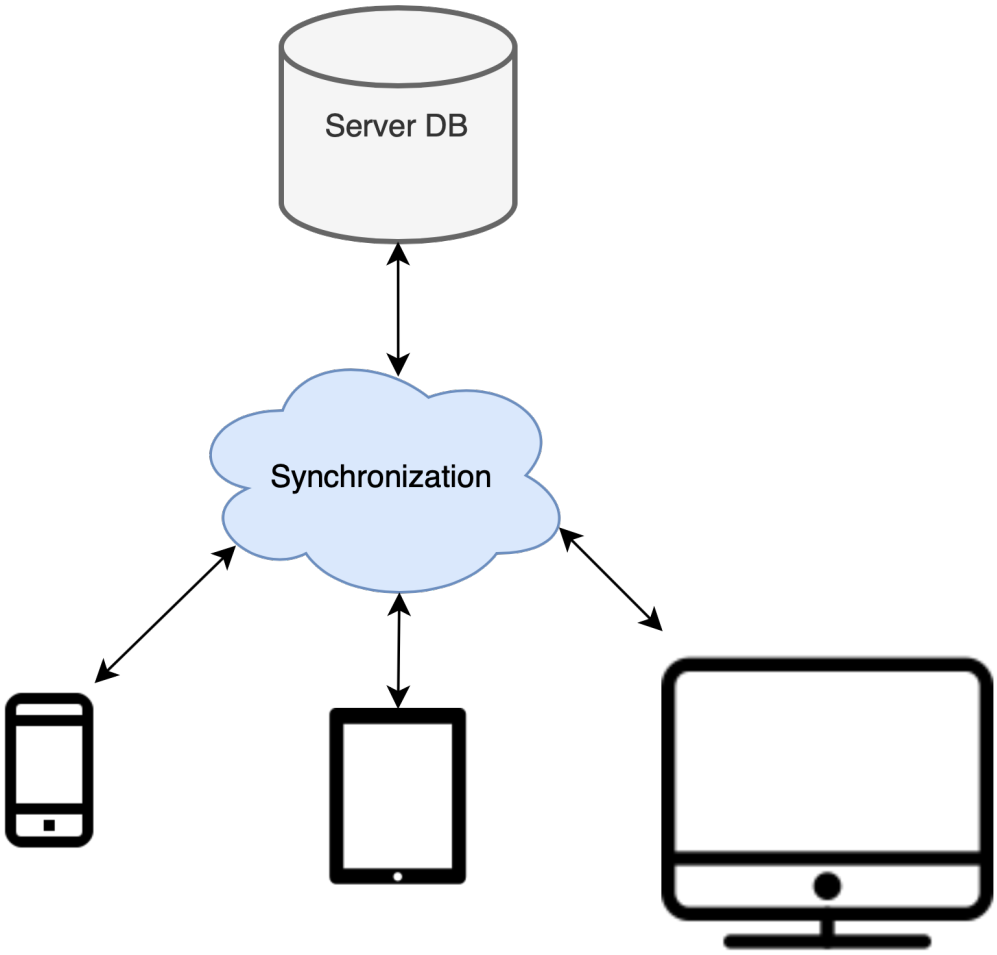
Lokálne dáta uložené systémom CoreData aj Realmom je možné synchronizovať medzi viacerými zariadeniami. Pri CoreData je možnosť synchronizovať dáta len pomocou CloudKit alebo pomocou vlastného riešenia. CloudKit využíva iCloud, ktorý nemá žiadne verejne prístupné rozhranie, a teda celá databáza je izolovaná v prostredí Apple. K takejto databáze nie je možné priamo pripojiť služby mimo platformu Apple, ako sú napríklad aplikácie na zariadeniach s operačným systémom Android. Tento spôsob synchronizácie dát je vhodný, iba ak je cieľová skupina primárne koncentrovaná okolo platformy Apple. Realm ponúka cloudovú službu, ku ktorej je možné pristupovať aj z iných platforiem. Rovnako ako pri iCloude nemá vývojár priamy prístup k databáze, a teda ani kontrolu nad typom a umiestnením databázy. Narozdiel od systému CoreData neumožňuje ukladanie väčších objektov. Realm ďalej neponúka jednoduchý spôsob zdieľania dát medzi viacerými používateľmi.

Na základe zhodnotenia porovnania uvedeného vyššie, sa riešenie tejto práce zaoberá synchronizáciou databázy spravovanej databázovým systémom Realm s možnosťou rozšírenia aj pre systém CoreData. Narozdiel od synchronizácie v systéme Realm navrhnuté riešenie podporuje zdieľanie objektov medzi viacerými používateľmi a umožňuje mať pod plnou kontrolou serverovú databázu aj správu používateľov.

2.5 Synchronizačné metódy

Stáva sa bežnou praxou vytvárať aplikácie, ktoré môže používateľ používať na viacerých zariadeniach. Či už sa jedná o mobilné, ako je napríklad telefón alebo tablet, tak aj o webové aplikácie a aplikácie určené pre desktopové počítače. Používateľ intuitívne očakáva, že v práci, ktorú vykonal v jednej aplikácii na jednom zariadení môže pokračovať v rovnakej aplikácii na inom zariadení a že dáta v obidvoch aplikáciach budú rovnaké. Ďalšie intuitívne očakávanie používateľa je, že v prípade, ak sa nejedná o webovú aplikáciu, ale napríklad o mobilnú, budú jeho dáta prístupné aj bez prístupu na internet. Okrem toho, že si bude môcť svoje dáta prezerať bez prístupu na internet, očakáva aj to, že bude môcť pokračovať v práci bez obmedzení. Táto potreba viedla k tvorbe viacerých spôsobov pre udržiavanie dát vo viacerých zariadeniach aktuálne a poskytnúť tak používateľovi čo najlepší zážitok z používania aplikácie (user experience).

Nakoľko sa jedná o komplexný problém, neexistuje žiadne štandardné riešenie použiteľné na všetky prípady [32]. Synchronizačné metódy sa rozdeľujú podľa toho, či sú založené na optimistickej alebo pesimistickej schéme. Synchronizácia s použitím pesimistickej schémy vyžaduje uzamknutie zápisu a prípadne aj čítania dát pokiaľ na tieto dáta niekto drží zápisový zámok. Týmto spôsobom sa zamedzuje vzniku konfliktných situácií, nakoľko iba



Obr. 2.4: Všeobecné zobrazenie viacerých zariadení synchronizujúcich sa medzi sebou.

jedna aplikácia môže v jednu dobu zapisovať. Synchronizácia založená na optimistickej schéme očakáva klientske kópie databázy, na ktorých môžu aplikácie vykonávať zmeny. Tieto metódy priam očakávajú vznik konfliktných situácii a spoliehajú sa na mechanizmus riešenia konfliktov [32]. V následujúcich sekciách sú popísané vybrané metódy založené na optimistickej schéme synchronizácie dát, nakoľko pesimistické nie sú vhodné pre aplikácie, ktoré potrebujú pracovať s dátami v režime off-line (bez prístupu na internet).

2.5.1 Komplexná synchronizácia

Pri synchronizácii odošle jedno zariadenie celý obsah databázy druhému. Po obdržaní dát začne druhé zariadenie porovnávať obdržané dáta s lokálnou databázou. Lokálny záznam sa aktualizuje v prípade, kedy záznam z obdržaných dát nie je identický s lokálnym záznamom. Výhodou tejto metódy je jednoduchá implementovatelnosť a integrovatelnosť do aplikácií, nakoľko nevyžaduje žiadnu úpravu databázovej schémy. Nevýhodou je potreba posielať všetky dáta aj v prípade, keď sa zmení iba malá časť a proces porovnávania môže byť výrazne časovo náročný pri aktualizovaní veľkého počtu záznamov [17, 13].

2.5.2 Transaction Level Result-Set Propagation

Aplikácia využívajúca metódu Transaction Level Result-Set Propagation (TLRSP) môže vykonávať lokálne zmeny v databáze aj v režime off-line. Po pripojení na server musia byť všetky klientske lokálne zmeny skontrolované serverom na prípadné konflikty pred tým, ako budú odoslané zvyšným aplikáciám. Klientska aplikácia môže priamo komunikovať iba so serverom. Aplikácia sa môže nachádzať v troch stavoch:

1. konzistentný stav – všetky konflikty sú vyriešené a aplikácia dokončila synchronizáciu,
2. zhromažďovací stav – lokálna databáza obsahuje nesynchronizované zmeny, udržiavajú sa tri množiny:
   (a) *read*: obsahuje všetky objekty, z ktorých sa čítalo,
   (b) *write*: obsahuje všetky pozmenené objekty,
   (c) *result*: obsahuje všetky objekty z množiny *write* s časovou známkou,
3. stav riešenia – zmeny v lokálnej databáze sa synchronizujú so serverom.

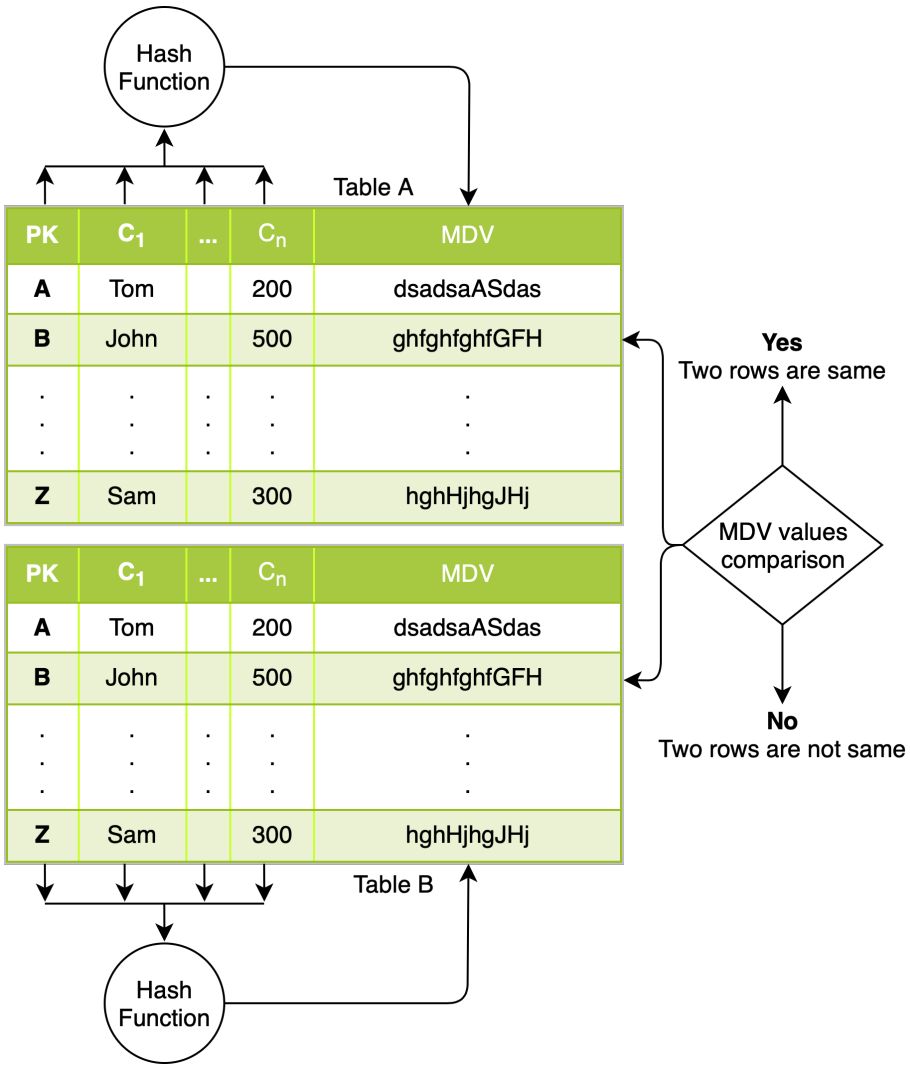
Počas stavu riešenia sa synchronizácia vykonáva v dvoch krokoch. Najprv sa na server odošlú všetky lokálne zmeny a vyriešia sa konflikty. V druhom kroku server vyberie všetky zmeny od poslednej synchronizácie a zašle ich späť klientskej aplikácii. Problémom tejto metódy je riešenie prípadných konfliktných záznamov vzniknutých lokálnymi zmenami klientov v režime off-line [16, 32].

2.5.3 Synchronization Algorithm based on Message Diggest (SAMD)

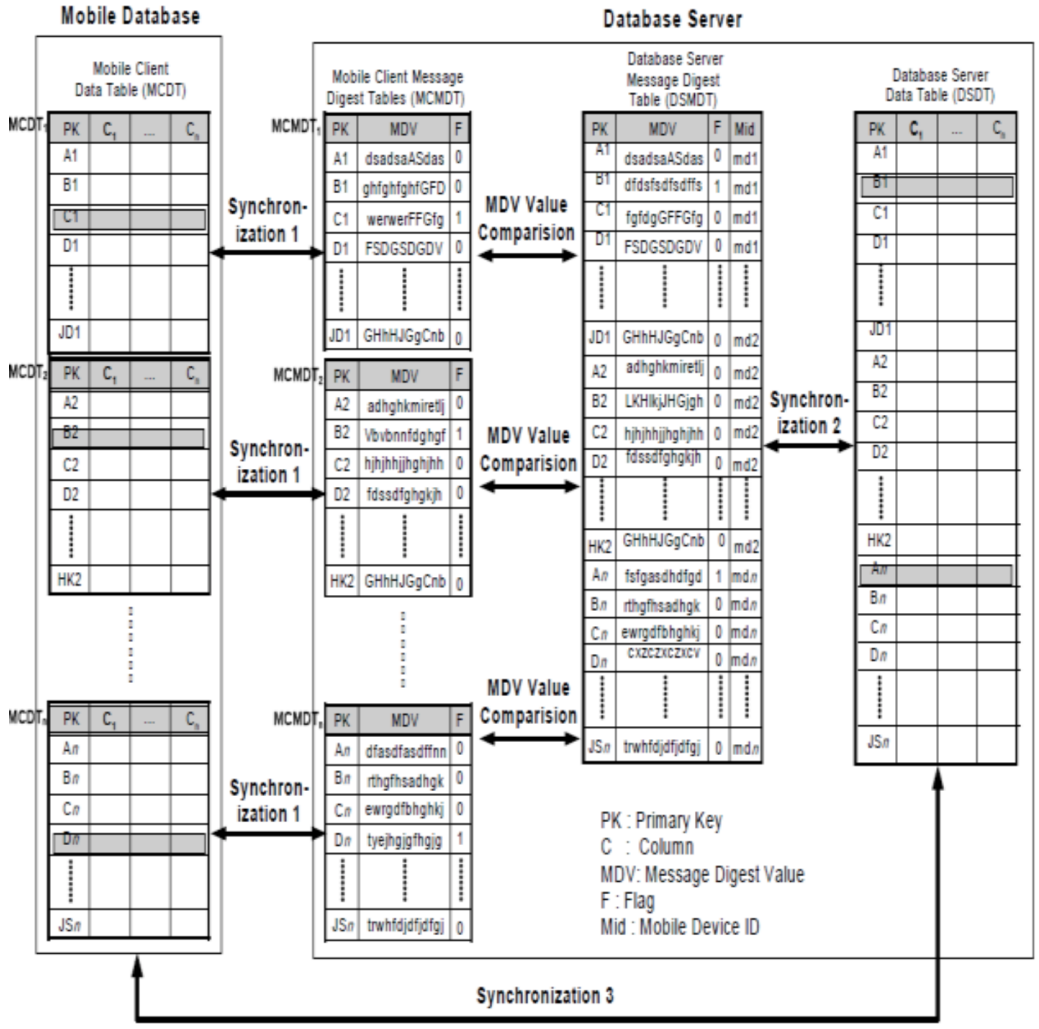
SAMD rieši synchronizačné problémy pomocou jednoduchých SQL funkcií špecifikovaných štandardom ISO, čím je možné ho efektívne použiť bez ohľadu na typ relačnej databázy na strane servera alebo strane klienta. Algoritmus drží kópiu celej databázy na všetkých zariadeniach. Každý záznam obsahuje aj hodnotu Message Digest (MDV) vypočítanú ako hash záznamu. Zmena záznamu spôsobí aj zmenu hashu a aktualizovaná hodnota Message Digest sa uloží. Pomocou porovnania hodnoty Message Digest dvoch databáz sa určia záznamy, ktoré majú byť synchronizované [17, 30].

V klientských databázach môžeme hodnotu Message Digest držať ako samostatný atribút v každej tabuľke (obrázok 2.5) alebo ju dynamicky počítať z atribútov pre každý riadok tabuľky pri synchronizácii. Na strane servera vzniká nutnosť pridať viaceré tabuľky, a to jednu pre každé zariadenie (Mobile Client Message Digest Table – MCMDT) a jednu tabuľku s hodnotami Message Digest servera (Database Server Message Digest Table – DSMDT). Každá tabuľka MCMDT obsahuje tri atribúty: primárny kľúč záznamu, hodnotu Message Digest a príznak zmeny. Jednotlivé riadky určujú klientskú hodnotu Message Digest pre každý záznam s príznakom, či daný záznam vyžaduje synchronizáciu. Tabuľka DSMDT obsahuje primárny kľúč záznamu, serverovú hodnotu MessageDigest, príznak zmeny a identifikátor zariadenia. Názorná ukážka je znázornená na obrázku 2.6. Počas synchronizácie klientského zariadenia so serverom sa do príslušnej tabuľky MCMDT nahrajú zmeny od klienta a nastaví sa príznak zmeny. Následne sa porovnajú hodnoty Message Digest s hodnotami v DSMDT a v prípade inkonzistencie sa aktualizuje databáza s klientskymi dátami.

Nevýhodou tohoto algoritmu je potreba posielať celý záznam aj v prípade, keď sa zmení iba jeden atribút, čo môže byť problematické v prípade nekvalitného internetového spojenia.



Obr. 2.5: Dve tabuľky spolu s hodnotou Message Digest, obrázok je vytvorený na základe predlohy z článku od Ramya, Koduri a Seetha[30].



Obr. 2.6: Tabuľky potrebné pre fungovanie SAMD synchronizáciu popísanú v sekcii 2.5.3. Obrázok je prevzatý z článku od Ramya, Koduri a Seetha[30].

Ďalšia veľká nevýhoda je potreba pridať veľké množstvo ďalších tabuliek na strane servera určených na uschovanie hodnoty Message Digest pre všetky záznamy a zariadenia [32].

2.5.4 Synchronizácia s príznakom status

Pri použití tejto metódy sa pre každý záznam udržiava príznak označujúci zmenu záznamu. Ak je príznak nastavený, znamená to, že záznam bol aktualizovaný a je potrebné ho synchronizovať. Pri synchronizácii sa posielajú iba záznamy s príznakom. Aj keď sa táto metóda zdá byť jednoduchá, obsahuje určité komplikácie. Jedna z komplikácií je nutnosť udržiavať informáciu o zariadeniach, ktoré zmenený záznam obdržali [17]. Výhodami sú nezávislosť synchronizovaných záznamov (chyba pri aktualizovaní niektorých záznamov neovplyvní aktualizáciu zvyšných) a udržiavanie bezstavovej integrácie [13].

2.5.5 Synchronizácia s použitím časových známok

Narozdiel od predchádzajúcej metódy sa namiesto príznaku využíva časová známka poslednej zmeny dát. Aplikácia si drží túto informáciu pre každý záznam v databáze. Okrem nej si drží aj informáciu o čase poslednej synchronizácie so serverom. Pri synchronizácii sa na server pošlú všetky záznamy s novšou časovou známkou ako je časová známka poslednej synchronizácie spolu s touto známkou. Server následne pošle všetky záznamy aktualizované od poslednej synchronizácie. Veľa aplikácií automaticky drží informáciu o poslednej aktualizácii záznamu, čo je v tomto prípade veľká výhoda. Nevýhodou oproti predchádzajúcej metóde je, že ak zlyhá synchronizácia jedného záznamu, automaticky zlyhajú všetky. Ďalšou nevýhodou je, že ak aplikácia vykonáva malé zmeny vo veľkom množstve záznamov, je nutné synchronizovať celé záznamy [13]. Pokiaľ je povolená synchronizácia priamo medzi klientmi, vzniká neefektívne synchronizovanie, ak dve aplikácie, ktoré majú rovnaké lokálne dáta, sa synchronizujú prvýkrát medzi sebou. Vďaka rozličným časovým známkam si vymenia všetky dáta [17].

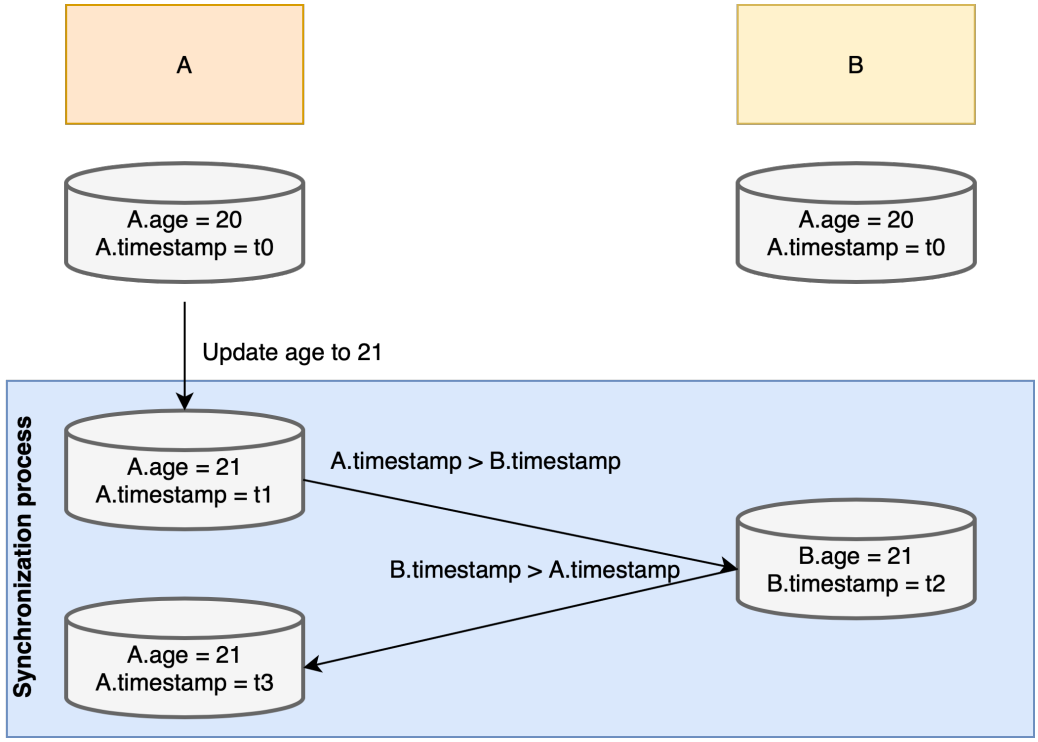
Pri použití tejto metódy bez servera hrozí podľa článku od spoločnosti Dell Boomi [14] vznik kruhových aktualizácií. Situácia nastane keď:

1. aplikácia A zmení záznam r_i a aktualizuje jeho časovú známku na t_1,
2. pri synchronizácii s aplikáciou B sa záznam r_i prekopíruje do aplikácie B a nastaví sa jeho časová známka na t_2,
3. synchronizačný proces zistí, že záznam r_i bol v aplikácii B aktualizovaný (má novšiu časovú známku), prekopíruje ho do aplikácie A a nastaví jeho časovú známku v aplikácii A na t_3,
4. keď sa nabudúce spustí synchronizačný proces, odhalí sa, že záznam r_i s časovou známkou t_3, je novší ako záznam v aplikácii B s časovou známkou t_2, a tak spustí aktualizáciu znova.

Kruhové aktualizácie spôsobia, že sa všetky záznamy budú aktualizovať pri každom synchronizačnom procese. Tento problém je graficky znázornený na obrázku 2.7. Riešením môže byť držať okrem časovej známky aj údaj o tom, kto naposledy zmenil daný záznam. Výhodou tohoto riešenia je jednoduchá možnosť implementácie, ale vznikne potreba jednoznačne identifikovať jednotlivých používateľov.

2.5.6 Synchronizácia pomocou logov

Tento druh synchronizácie sa často používa priamo v databázach alebo vo verzovacích systémoch a je využitý aj v systéme CoreData alebo v databázach iCloudu. Princíp je založený na tom, že každá zmena v databáze je vykonaná ako jedna transakcia. Jednotlivé transakcie sa potom ukladajú v poradí, v ktorom boli vykonávané ako logy. Pri synchronizácii sú jednotlivé logy prekopírované na cieľové zariadenie a postupne interpretované nad databázou. Hlavnou nevýhodou je rýchly rast veľkosti logov na serveri, nakoľko každá zmena sa ukladá samostatne [17]. Táto nevýhoda môže byť čiastočne odľahčená obmedzením na maximálny čas, koľko môže byť log v databáze. Príkladom môže byť, že všetky logy staršie ako 3 mesiace sa zredukujú na čo najmenší počet logov. Jedna z redukcií môže byť nahradenie viacerých zmien toho istého atribútu za jedinú. Z tejto vlastnosti vyplýva aj výhoda histórie a možnosti vykonať návrat do určitého bodu v databáze. Ďalej ku každému logu



Obr. 2.7: Problém kruhovej aktualizácie. Po vykonaní synchronizačného procesu sú synchronizované dáta označené novšou známkou ako v iných zariadeniach, čo spôsobí ich znovusynchronizáciu.

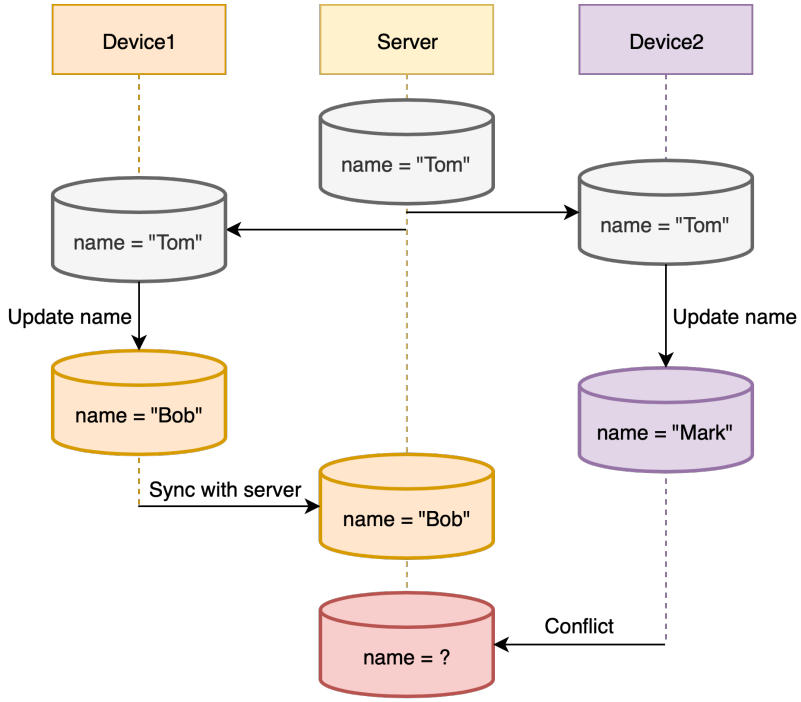
môžeme vytvoriť log inverzný, ktorý vráti späť zmeny spôsobené daným logom, čím získavame možnosť undo. Ďalšia výhoda oproti spomenutým metódam je možnosť aktualizovať iba jeden atribút a nie celý záznam, čo zmenšuje veľkosť prenášaných dát.

2.5.7 Konflikty a spôsoby ich riešenia

Ak viacej zariadení bez prístupu na internet upraví rovnaký atribút rovnakého záznamu na iné hodnoty a následne sa pokúsia synchronizovať, vzniká konflikt, nakoľko nie je jasné, ktorá verzia atribútu má ostať a ktoré verzie sa majú vymazať (obrázok 2.8) [17]. Môžeme rozoznávať medzi tromi druhmi konfliktov. Aktualizácia aktualizovanej hodnoty (Update-Update) nastáva v spomenutej situácii. Aktualizácia zmazanej hodnoty (Update-Delete) nastane, keď zdrojová aplikácia aktualizovala hodnotu, ktorá bola v cieľovej zmazaná a zmazanie aktualizovanej hodnoty (Delete-Update) nastane v opačnej situácii ako Update-Delete, teda keď sa zdrojová aplikácia snaží presadiť zmazanie hodnoty, ktorá bola v cieľovej aplikácii pred synchronizáciou aktualizovaná [28].

Vzniknutý konflikt je možné vyriešiť jednou z nasledujúcich stratégií navrhnutých v článku od Young Seok Lee, Youn Soo Kim a Hoon Choi [27]:

- pôvodca vyhráva – konfliktné dáta sa prepíšu na dáta doručené,
- prijímateľ vyhráva – konfliktné dáta sa neprepisujú,
- klient vyhráva – konfliktné dáta sa prepíšu podľa dát klienta,



Obr. 2.8: Vznik konfliktu

- server vyhráva – konfliktné dáta sa prepíšu podľa dát servera,
- novší vyhráva – konfliktné dáta sa prepíšu na novšiu verziu dát na základe časovej známky,
- duplikácia – požadované zmeny sa vykonajú na duplikovaných dátach.

Adolfo Marinucci vo svojom článku [28] popísal taktiež niektoré spomenuté stratégie a uviedol aj stratégiu zameranú na manuálne vyriešenie konfliktov, v ktorej výslednú hodnotu konfliktných dát určuje programátor alebo používateľ aplikácie.

Synchronizačné aplikácie môžu podporovať všetky alebo len niektoré zo spomenutých stratégií, ale systémový administrátor, alebo majiteľ zariadenia, musí použiť iba jednu, aby sa zachovala integrita dát [27, 17].

Názorný príklad riešenia konfliktných situácií pri použití stratégie novší vyhráva je zobrazený v tabuľke 2.1. Z tabuľky je možné vidieť, že môže nastať chyba v jednom až troch prípadoch a stačí vytvoriť päť pravidiel pre implementáciu tejto stratégie.

2.5.8 Spôsoby implementácie

Každá synchronizačná metóda potrebuje prepravovať dáta medzi viacerými zariadeniami. Spôsob tejto prepravy je často riešený pomocou webových služieb, či už použitím čistého HTTP(S) alebo použitím viac formálnych metód, ako je napríklad REST (Representational State Transfer Protocol) alebo SOAP (Simple Object Access Protocol) [15].

SOAP

SOAP je protokol navrhnutý Microsoftom v roku 1998 ako náhrada za vtedajší protokol COBRA. Správy protokolu sú založené primárne na XML, čím sa protokol stáva platformovo

Novšie dáta	Staršie dáta	Akcia	Pravidlo
pridanie	pridanie	chyba	P1
	zmazanie		
	aktualizovanie		
zmazanie	pridanie	zmazanie/chyba	P2
	zmazanie	zmazanie	P3
	aktualizovanie		
aktualizovanie	pridanie	aktualizácia/chyba	P4
	zmazanie	aktualizácia	P5
	aktualizovanie		

Tabuľka 2.1: Riešenie synchronizácie a konfliktov pri použití stratégie novší vyhráva. Tabuľka prevzatá z článku od YoungSeok Lee, YounSoo Kim a Hoon Choi [27]. Akcia sa volí na základe kombinácie novších a starších dát. Podľa akcie sa vykoná požadované pravidlo pre integrovanie zmeny.

nezávislý [29, 34]. Hlavným princípom je, že poskytovateľ služieb zverejní popis služieb alebo rozhranie registru služieb tak, aby klient mohol nájsť požadovanú službu a použiť ju. Veľkosť prenášaných správ môže pôsobiť nepriaznivo na výkon, pretože pri vytváraní správ protokol SOAP pridáva ďalšie časti hlavičky a tela do správy. Najčastejšie sú správy prenášané pomocou protokolu HTTP alebo SMTP [34].

REST

Narozdiel od SOAP je tento protokol novší a vytvorený pre používanie spolu s protokolom HTTP. Tvar správ nie je konkrétne špecifikovaný. Môžu byť vo formáte XML, JSON prípadne v inom formáte. Služby sú poskytnuté ako zdroje identifikované pomocou URI (Uniform Resource Identifier). Operácia s daným zdrojom je určená pomocou HTTP metód, ako sú napríklad GET, PUT, POST, DELETE a podobne [34].

Porovnanie REST a SOAP

REST sa v posledných rokoch stal dominantnou variantou pre tvorbu nových aplikačných rozhraní (API). Pál a Láner vo svojom článku [29] píšu, že v roku 2014 používalo REST až 69% nových API a iba 22% SOAP. Hlavné dôvody, prečo vývojári presadzujú REST, sú rýchlejšie spracovávanie správ (SOAP správy sú rádovo väčšie), menšia veľkosť správ (menej sa vyťažuje pamäť) a lepšia škálovatelnosť. REST sa najčastejšie používa pri integrácii mobilných aplikácií s informačnými systémami a pri jednoduchej výmene dát medzi klientom a serverom. SOAP, na druhú stranu, ponúka vyššiu bezpečnosť a väčšiu spoľahlivosť. Počet možných chýb je taktiež menší pri protokole SOAP. Jeho využitie je primárne v biznisových a bankových informačných systémoch [34].

2.5.9 Problémy v aplikáciách pri používaní synchronizácie

Okrem konfliktov môžu počas synchronizácie vzniknúť problémy spôsobené životným cyklom aplikácie. Jedným z problémov je, ako sa má aplikácia zachovať, pokiaľ používateľ pracuje na jednom zariadení s dátami, ktoré sa na inom zariadení počas práce vymažú. Ďalším problémom je, keď synchronizačný proces nastane v bode, keď používateľovi sú prezentované dáta, ktoré sú počas procesu synchronizácie pozmenené a aplikácia nie je navrhnutá

na zobrazovanie zmeny. Napríklad pri zobrazovaní používateľského profilu je potrebné sledovať prípadné zmeny a nepoužívať atribúty používateľa ako napríklad meno a email ako konštanty.

Tieto problémy kladú dôraz na zadefinovanie spôsobu ako vytvárať aplikácie so synchronizovanými dátami tak, aby vznik problémov bol ľahko riešiteľný alebo sa vzniku problémom úplne zabránilo.

2.5.10 Zhrnutie metód synchronizácie

V predchádzajúcich podsekciách bolo spomenutých niekoľko synchronizačných metód. Metódy SAMD a TLRSP nie je možné používať pre synchronizáciu priamo medzi klientmi, narozdiel od zvyšných. SAMD navyše vyžaduje veľké množstvo tabuliek na serveri. Komplexná synchronizácia je pri potrebe často synchronizovať malé zmeny vo veľkých databázach prakticky nepoužiteľná, ale je veľmi jednoducho implementovateľná a nie je potrebné nijako meniť serverové rozhranie. Jednoducho implementovateľná je aj synchronizácia pomocou príznaku status alebo s použitím časových známok, pričom synchronizácia pomocou príznaku status vie pokračovať v synchronizácii aj pri zlyhaní synchronizácie jedného záznamu bez potreby znovu spustiť synchronizačný proces. Jediná spomenutá metóda umožňujúca efektívne synchronizovať zmeny jednotlivých atribútov je synchronizácia pomocou logov. Táto metóda umožňuje efektívne udržiavať históriu dát za cenu rýchleho rastu logov uložených na serveri.

V reálnych aplikáciách sa používajú kombinácie metód, čím sa znižujú nevýhody a maximalizujú výhody jednotlivých synchronizačných metód. Príkladom môže byť využitie príznaku status a časových známok súčasne čím sa zefektívni synchronizácia [17].

Pri návrhu protokolu použitého v tejto práci bol braný ohľad na niekoľko rôznych faktorov, ako je bezstavovosť, bezpečnosť, jednoduchosť, robustnosť a veľkosť správ. Aby boli správy protokolu čo najmenšie, je základom protokolu synchronizácia pomocou logov doplnená časovými známkami, čím sa zlepšuje schopnosť riešenia konfliktov pri synchronizácii. Pre zabránenie vzniku kruhových aktualizácií je zavedený centrálny prvok, voči ktorému sa všetci synchronizujú. Protokol je inšpirovaný SAMD, pričom sa namiesto hashov pre každý záznam vyvára identifikátor pre kolekciu objektov. Identifikátor slúži k určovaniu, či boli objekty v kolekcii pozmenené alebo nie, obdobne ako v SAMD. Vďaka tomu, že sú zmeny reprezentované logmi, nie je potrebné vytvárať tabuľku pre každé zariadenie. Postup výmeny dát je riadený podobne ako v TLRSP, v ktorom sa najprv odošlú všetky zmeny na server, kde sa vyriešia všetky konflikty. Následne server odošle klientovi zmeny, ktoré nemá zosynchronizované.

2.6 Prepojenie používateľa, databázy a synchronizácie v aplikácii

V bežných aplikáciách je napojenie používateľského rozhrania na lokálnu databázu často jednostranné a riadené používateľom. Keď používateľ vykoná zmenu v nejakom zázname, tak vyvolá aj aktualizáciu používateľského rozhrania tak, aby zobrazovalo aktuálne dáta. Pri použití synchronizácie vznikajú zmeny v databáze, ktoré sa musia spropagovať do používateľského rozhrania bez používateľovej akcie. V tejto sekcii je popísaný spôsob, vďaka ktorému je možné udržiavať používateľské rozhranie aktuálne voči lokálnej databáze. Keďže zvolený databázový systém v tejto práci je Realm, tak je popísané iba prepojenie pre neho.

2.6.1 Prístup k databáze

Pre prístup k dátam uloženým v databáze sa používa objekt typu `Realm`, ktorý je viazaný na vlákno, v ktorom bol vytvorený. Zmena v databáze sa vykonáva pomocou funkcie `write` zavolanej nad týmto objektom, ako je zobrazené vo výpise 2.10. Používateľ aj synchronizačný proces môžu pomocou tohoto objektu vykonávať zmeny nezávislo a databázový systém sa postará, aby pracovali s rovnakými objektami.

2.6.2 Sledovanie zmien v databáze

Používateľ aj synchronizačný proces môžu sledovať jednotlivé zmeny nad objektami pomocou Realm notifikácií, ktoré sú automaticky volané po každom zapísaní do databázy. Synchronizačný proces na základe notifikácie generuje logy, ktoré sa odošlú na server. Aplikácia môže využívať notifikácie k aktualizácii používateľského rozhrania.

2.6.3 Propagovanie zmien do používateľského rozhrania

V aplikáciách pre zariadenia od firmy Apple je najpoužívanejší prvok používateľského rozhrania tabuľka. Väčšinou sa jedná o jedno-stĺpcovú tabuľku, kde každý riadok reprezentuje jeden konkrétny objekt uložený v databáze. Tabuľky je možné spravovať pomocou notifikácií. Prepojenie databázy s používateľským rozhraním využívajúceho storyboardy[8] je zobrazené vo výpise 2.13. Na základe zmien obdržaných v notifikácii sa aktualizujú jednotlivé riadky tabuľky. Rozhranie vytvorené pomocou knižnice SwiftUI je možné aktualizovať pomocou publishera, ktorý sleduje tieto notifikácie a pri zmene objektov vyvolá aktualizáciu.

```
1  class ViewController: UITableViewController {
2      var notificationToken: NotificationToken? = nil
3
4      override func viewDidLoad() {
5          super.viewDidLoad()
6          let realm = try! Realm()
7          let results = realm.objects(Person.self).filter("age > 5")
8
9          notificationToken = results.observe { [weak self] changes in
10             guard let tableView = self?.tableView else { return }
11             switch changes {
12             case .initial:
13                 tableView.reloadData()
14             case let .update(_, deletions, inserts, modifs):
15                 tableView.beginUpdates()
16                 let ipd = deletions.map { IndexPath(row: $0, section: 0)}
17                 tableView.deleteRows(at: ips,with: .automatic)
18                 let ipi = inserts.map { IndexPath(row: $0, section: 0) }
19                 tableView.insertRows(at: ipi, with: .automatic)
20                 let ipm = modifs.map { IndexPath(row: $0, section: 0) }
21                 tableView.reloadRows(at: ipm, with: .automatic)
22                 tableView.endUpdates()
```

[8] Storyboard je vizuálna reprezentácia používateľského rozhrania.

```
23              case .error(let error):
24                  fatalError("\(error)")
25              }
26          }
27      }
28
29      deinit {
30          notificationToken?.invalidate()
31      }
32  }
```

Výpis 2.13: Riadenie zmien používateľského rozhrania v tabuľke pomocou Realm notifikácií. Príklad prevzatý z oficiálnej dokumentácie [31].

2.6.4 Zhrnutie prepojenia

Synchronizačný proces je možné prepojiť s lokálnou databázou bez nutnosti zásahu do používateľského rozhrania. Používateľke rozhranie aj synchronizačný proces komunikujú s lokálnou databázou pomocou vlastnej inštancie objektu typu `Realm`, ktorý sa stará o odosielanie notifikácií.

Kapitola 3

Koncept práce

Riešený problém v tejto práci očakáva existenciu nejakej klientskej aplikácie na zariadení A, ktorá v rámci svojho životného cyklu vytvára objekty viazané k jej používateľovi. Tieto objekty majú perzistentný charakter, a teda je potrebnosť ich ukladať do databázy. Ďalej sa predpokladá využívanie tejto aplikácie aj na mobilných zariadeniach (mobilný telefón, tablet), a teda aplikácia je určená pre prácu aj bez internetového pripojenia. Ďalší predpoklad je, že jeden používateľ môže mať rovnakú aplikáciu na viacerých zariadeniach $(B, C, D, ...)$, pričom intuitívne očakáva, že dáta patriace tejto aplikácii budú zrkadlené na všetkých jeho zariadeniach. To znamená, že pri zmene objektu v aplikácii na zariadení A sa automaticky spustí synchronizačný proces, ktorý vykonané zmeny distribuuje do aplikácií na zariadeniach B, C, D a podobne.

Špecifikom problému je teda využívanie klientskej aplikácie aj bez garancie neustáleho internetového pripojenia, a teda táto práca predpokladá prácu s aplikáciou aj v režime offline a potrebu následnej distribúcie dát medzi ďalšie klientske zariadenia. S touto problematikou súvisí aj riešenie analýzy chýb konzistencie v databázach (konfliktov), ktoré nastatú, keď sa rovnaký objekt zmení na viacerých zariadeniach a je potrebné určiť, ktorá zmena sa má ponechať a ktoré zvyšné zmeny zahodiť. Ďalším špecifikom problému je zdieľanie dát medzi viacerými používateľmi a tvorba skupín, v ktorej majú viacerí používatelia prístup k rovnakým objektom. Táto diplomová práca predkladá riešenie zrkadlenia dát medzi viacerými zariadeniami jednej klientskej aplikácie s podporou riešenia konfliktov a používateľskými skupinami vo forme knižníc určených pre klientské a serverové aplikácie – MeerkatSync.

Detailný návrh jednotlivých procesov je uvedený v následujúcich sekciách. Najprv je popísaný návrh základného princípu predkladaného synchronizačného riešenia v sekcii 3.1. V dvoch nasledujúcich sekciách je popísaný návrh procesov centrálneho prvku (sekcia 3.2) a procesov v klientskych aplikáciách (sekcia 3.3) zaisťujúcich synchronizačný proces. V poslednej sekcii tejto kapitoly (sekcia 3.4) je popísaný návrh synchronizačného protokolu pre výmenu synchronizačných správ využívaný klientskou aplikáciou a aj centrálnym prvkom.

3.1 Princíp navrhovaného synchronizačného procesu

Predkladané riešenie je založené na komunikácii typu klient-server, to znamená, že pre distribúciu dát sa využíva centrálny prvok – server. Centrálny prvok v celom systéme figuruje ako účastník zodpovedný za rozhodovanie o podobe databázy v okamihoch, kedy je detekovaný konflikt. Okrem toho si centrálny prvok drží aktuálnu kópiu databázy aplikácie

so statusom master-kópie, čo znamená, že databáza centrálneho prvku slúži ako vzor pre databázu všetkých klientov.

V nasledujúcom texte sa bude slovom aplikácia označovať klientska aplikácia, ktorej dáta sú automaticky synchronizované.

3.1.1 Schéma aplikačnej databázy

Schéma je založená na používateľskej špecifikácii relácií medzi entitami. Pre špecifikované schéma musí platiť, že každý synchronizovateľný atribút v entite musí byť jeden z nasledujúcich typov[1]:

- reťazec (`String`),
- celé číslo (`Int`),
- pole bytov (`Data`),
- boolean (`Bool`),
- kladné celé číslo (`UInt`),
- desatinné číslo (`Double`),
- dátum (`Date`),
- optional[2] varianty vyššie spomenutých typov (`Optional<T>`),
- optional odkaz na inú entitu (`Optional<Object>`)
- a pole entít (`[Object]`).

Typové obmedzenie vychádza z nutnosti transformovať daný typ do bytovej reprezentácie a späť. Výhoda tohoto obmedzenia je, že centrálny prvok má dostatok informácií pre vykonávanie kontroly, napríklad pre zabránenie, aby sa do atribútu typu reťazec dostal iný typ a podobne.

Jednotná reprezentácia hodnôt zaručuje jednoduché ukladanie týchto hodnôt v databáze centrálneho prvku. V ďalšom texte sa bude bytová reprezentácia označovať pojmom `Data` podľa označenia z jazyku Swift.

3.1.2 Základné procesy

Samotná synchronizácia dát sa zaisťuje niekoľkými samostatnými procesmi. Na strane aplikácie to sú procesy určené k detekcii zmien v lokálnej databáze a ich uloženie vo forme logov pre neskoršiu synchronizáciu a proces určený k distribúcii zmien centrálnemu prvku. Distribúcia zmien centrálnemu prvku je spúšťaná automaticky pri každej synchronizovateľnej zmene v aplikačnej databáze za prístupnosti internetového pripojenia. Algoritmus tejto distribúcie je zobrazený na algoritme 1 a skladá sa z troch dôležitých krokov:

- ustanovenie spojenia medzi aplikáciou a centrálnym prvkom,

[1] V zátvorke sú uvedené odpovedajúce typu v jazyku Swift.

[2] Typ Optional je generický typ, ktorý umožňuje reprezentovať nenastavenú hodnotu – `nil`.

- odoslanie lokálnych zmien centrálnemu prvku, ktorý rozhodne o ich integrácii do master-kópie databázy,
- prijatie zmien od centrálneho prvku potrebných k dosiahnutiu konzistencie medzi lokálnou kópiou a master-kópiou. Tieto zmeny vyplývajú z rozdielu medzi aplikačnou databázou a master-kópiou.

Algorithm 1 Algoritmus distribúcie zmien na server.

Input: New local updates
Output: Difference between local and master copy of database

```
1: server.connect()
2: server.send(newUpdates)
3: diff ← server.getDifference()
4: return diff
```

Na strane centrálneho prvku sa jedná o procesy obstarávajúce jednotlivé požiadavky klientskych zariadení, procesy zostavujúce protokol o zmenách pre jednotlivé zariadenia, procesy určené k distribúcii informácií o zmenách v master-kópii databázy pomocou notifikácií. Navrhnutá knižnica MeerkatSync predpokladá notifikačný systém iba ako doplnkový spôsob zisťovania, či nastala zmena v master-kópii, nakoľko notifikácie môžu byť nespoľahlivé a nie je zaistené ich doručenie na všetky zariadenia[3].

3.1.3 Verzovanie dát

MeerkatSync využíva spôsob synchronizácie, pri ktorom je obsah dát čo najmenší, a teda pri distribúcii dát sa prenáša iba protokol o zmenách vykonaných od poslednej synchronizácie. Tento princíp vyžaduje určenie spôsobu, akým budú označované jednotlivé verzie objektov. Táto práca využíva dva spôsoby verzovania pre zaručenie správneho prístupu k dátam a riešeniu konfliktov. Jeden spôsob vychádza zo synchronizácie s využitím časových zámok, pričom druhý doplňujúci prístup určuje pre každú množinu súvisiacich objektov číselnú verziu.

Časová známka

Každá vykonaná zmena atribútu v objekte alebo objektu samotného je označená časovou známkou na základe aktuálneho systémového času zariadenia. Tento princíp slúži k určeniu verzie objektu na strane klientskej aplikácie. Vďaka časovej známke vzniknutej pri každej zmene je možné pri riešení konfliktov využívať stratégiu novší vyhráva, ako je uvedené v tabuľke 2.1. Časová známka teda reprezentuje verziu daných objektov zo strany klienta.

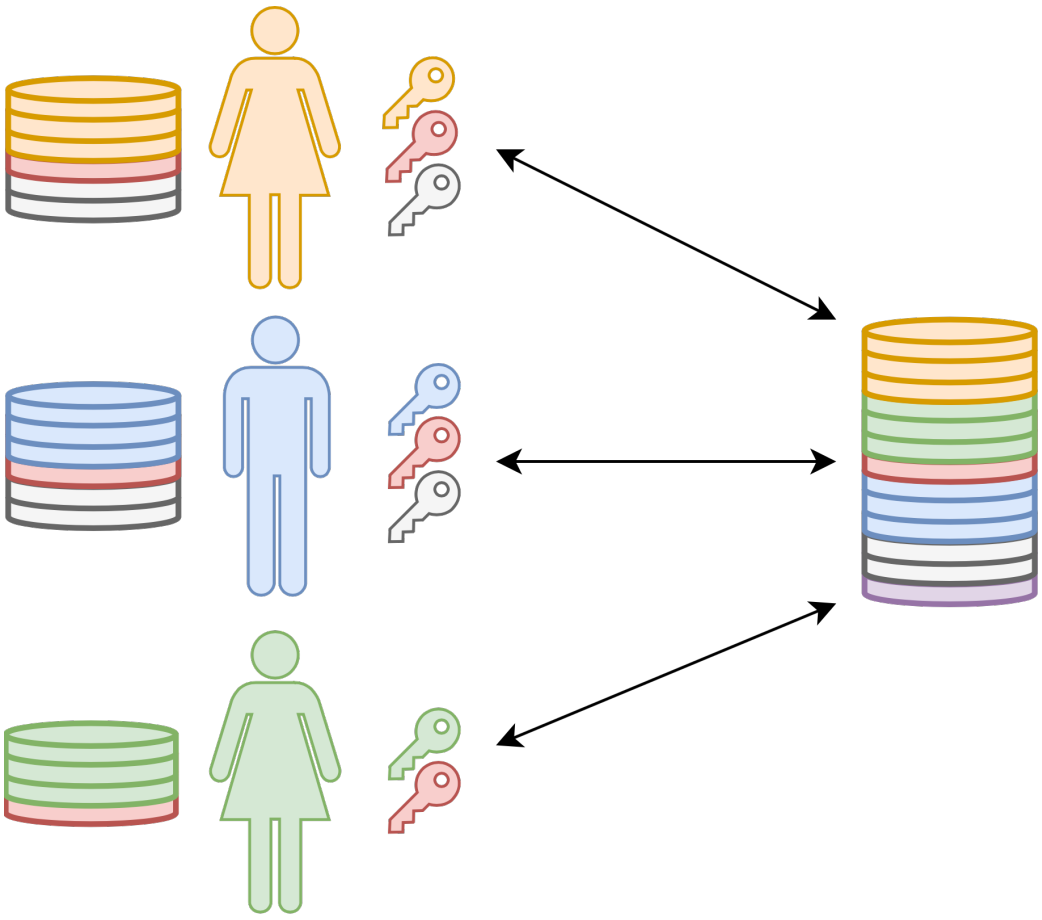
Číselná verzia skupiny objektov

Časová známka vzniká na klientskych zariadeniach, ktoré môžu mať rozličné systémové časy. Ďalšia čiastočná nevýhoda je, že sa viaže ku konkrétnemu atribútu a neoznačuje skupinu zmien ako celok, a preto je potrebné využívať ďalší spôsob verzovania, ktorý vyrieši tieto problémy. Na strane centrálneho prvku sa teda pri vykonaní zmien nad nejakými objektmi vytvorí nový číselný identifikátor reprezentujúci aktuálnu verziu skupiny týchto objektov.

[3] Napríklad pokiaľ je aplikácia v režime off-line.

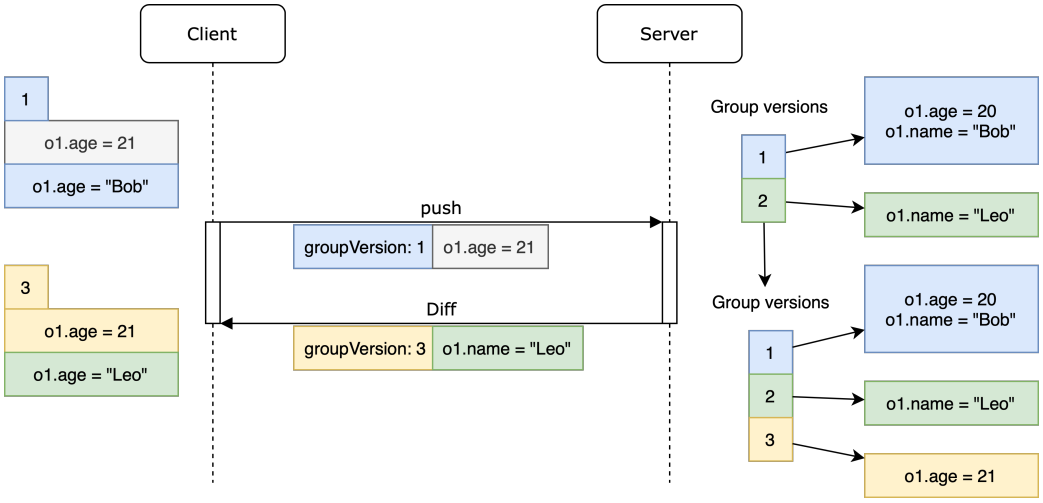
V predkladanom riešení sú teda objekty rozdelené do samostatných skupín s prázdnym prienikom voči ktorým sa tento identifikátor viaže. Číselný identifikátor teda označuje verziu skupiny objektov.

Vďaka tomuto princípu je možné jednoducho zdielať objekty medzi viacerými používateľmi. Na strane centrálneho prvku je potrebné udržiavať informáciu o príslušnosti používateľov k jednotlivým skupinám a ich oprávnenia v danej skupine. Na obrázku 3.1 je uvedený princíp zdieľania objektov v databáze pomocou týchto skupín. Skupina, ktorá je výhradne vyčlenená iba jednému používateľovi sa bude v nasledujúcom texte označovať ako primárna skupina používateľa.



Obr. 3.1: Zdieľanie skupiny objektov. Každá farba označuje inú skupinu objektov. Skupina objektov, ktorá má rovnakú farbu ako daný používateľ označuje používateľovu primárnu skupinu.

Vďaka tomu, že je daná verzia generovaná na strane centrálneho prvku a je nezávislá na časových známkach, je využívaná klientskymi aplikáciami, ktoré si túto verziu ukladajú. Aplikácie sa pri komunikácii s centrálnym prvkom preukážu lokálne uloženými verziami a centrálny prvok na základe tejto informácie vie určiť, aké zmeny im treba dodatočne zaslať, aby bola klientska kópia databázy rovnaká ako master-kópia. Príklad takejto výmeny je zobrazený na obrázku 3.2.



Obr. 3.2: Využívanie skupinovej verzie pri získavaní nových zmien. Jednotlivé farby reprezentujú zmeny podľa verzie skupiny. Na obrázku zmenil klient hodnotu atribútu `o1.age`. Pri synchronizácii sa okrem zmeny poslala aj posledná známa verzia skupiny, v ktorej je objekt (modrá), vďaka čomu vedel centrálny prvok určiť, aké zmeny musí klient vykonať, aby sa ich databáza zhodovala (zelená).

3.1.4 Reprezentácia zmien

Pri vykonaní každej zmeny v klientskej kópii databázy je potrebné túto zmenu zachytiť a upraviť do správneho tvaru tak, aby obsahovala iba požadované informácie. V tejto práci sa jednotlivé zmeny zoskupujú do troch samostatných skupín:

- zmazané objekty,
- vytvorené objekty,
- upravené objekty.

Takéto zoskupenie sa v tejto práci bude označovať ako transakcia.

Zmeny v týchto skupinách sú označované ako pod-transakcie a každá obsahuje časovú známku a prípadne aj atribúty, ktoré sa zmenili spolu s ich novou hodnotou.

Redukovanie veľkosti transakcie

Veľkosť transakcie je možné redukovať niekoľkými spôsobmi. Jedným zo spôsobov je, že v prípade, ak sa v jednej transakcii objavuje zmena toho istého atribútu viackrát, tak sa ponechá iba najnovšia zmena (pri každej zmene je časová známka jej vzniku). Druhý spôsob pre redukciu zbytočných zmien je, že v prípade, kedy sa objaví zmena určujúca zmazanie nejakého objektu Obj_1, sa zmažú všetky staršie zmeny tohoto objektu.

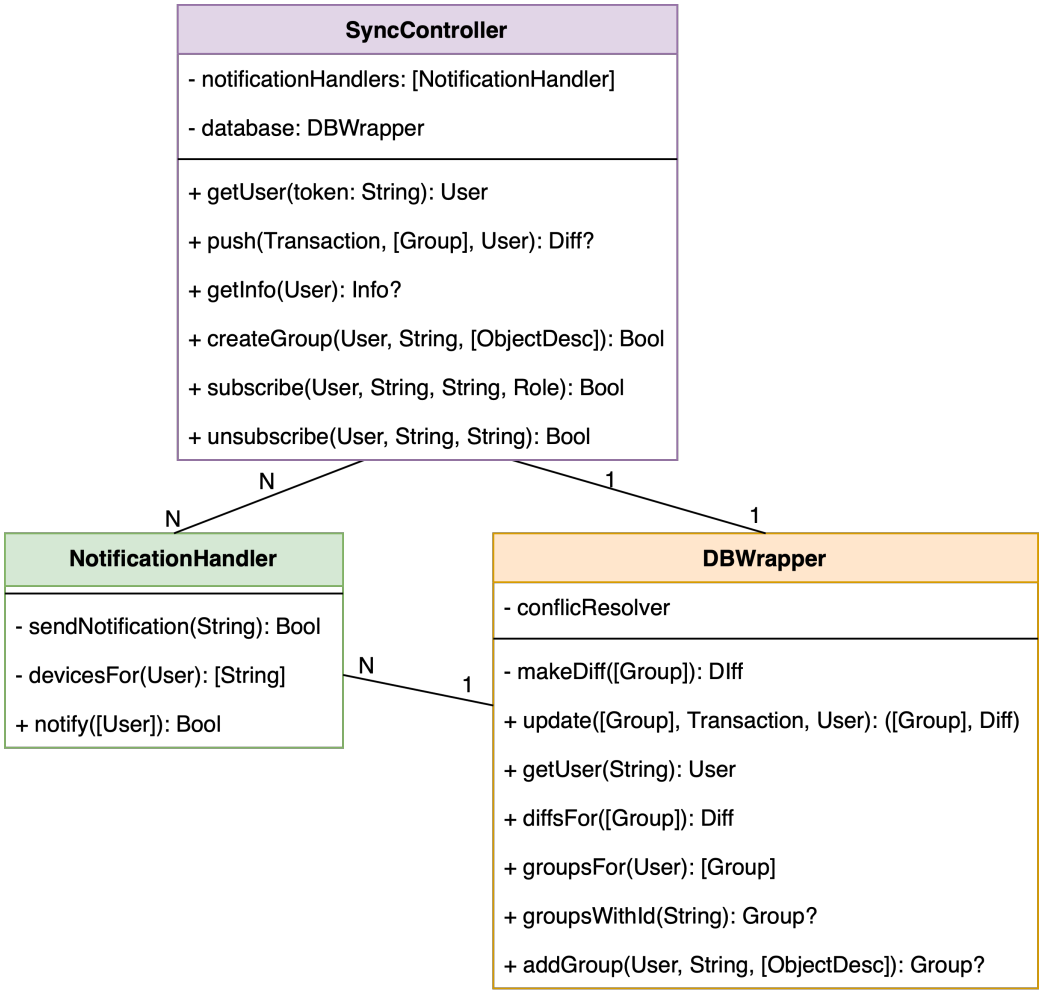
3.2 Návrh serverovej časti

Serverová časť reprezentuje centrálny prvok navrhovaného systému. Ako bolo spomenuté, tento prvok si drží kópiu databázy so statusom master-kópie a rozhoduje o tom aké zmeny

budú do tejto databázy zahrnuté, a ktoré budú odmietnuté. Centrálny prvok rozhoduje aj o tom, kedy a komu budú odoslané notifikácie.

Aby bolo navrhované riešenie čo najviac generické, bolo treba rozdeliť jednotlivú logiku na niekoľko častí a navrhnúť ich rozhranie, ktoré je zobrazené na obrázku 3.3.

V tejto sekcii sú popísané dôležité procesy zaobstarávajúce spomenuté činnosti a komunikácia medzi nimi.



Obr. 3.3: Diagram tried serverovej časti.

3.2.1 Riadenie synchronizácie – synchronizačný kontrolér

V centrálnom prvku je celé riadenie synchronizačného procesu vykonávané pomocou synchronizačného kontroléra, ktorý v systéme figuruje ako jedináčik pre každú klientskú aplikáciu. Jeho metódy sú volané na základe jednotlivých požiadaviek klientskych aplikácií.

Konkurencia požiadaviek

Aby bolo možné využívať predkladané riešenie aj v systémoch, ktoré sú náročné na počet klientskych požiadaviek, bolo potrebné určiť spôsob prístupu ku kontroléru tak, aby žiadna

požiadavka nebola zbytočne odkladaná a bolo možné obsluhovať čo najviac požiadaviek súčasne.

Nakoľko každá požiadavka sa viaže ku konkrétnym skupinám objektov, tak kontrolér považuje tieto skupiny ako zdroje s obmedzeným prístupom. Na základe typu požiadavky sa určí, či sa počas jej odbavovania môže zmeniť nejaký objekt v nejakej skupine. Pokiaľ požiadavka plánuje iba čítať, tak sa skupiny, z ktorých plánuje čítať, uzamknú pre čítanie a každá požiadavka s možnosťou zápisu do týchto skupín bude pozastavená, dokým všetky rozpracované požiadavky nedokončia čítanie. Pri potrebe zapisovať do nejakej zo skupín sa tieto skupiny uzamknú pre zápis, vďaka čomu sú zdroje priradené výlučne iba jednej požiadavke až dokým neskončí zápis a tieto zdroje neuvoľní.

Vďaka tomuto je možné bezpečne vykonávať viaceré požiadavky, ktoré budú z rovnakej skupiny a požiadavky smerované na skupiny, s ktorými sa práve nemanipuluje, môžu byť vykonávané súčasne.

Integrovanie nových zmien

Pri potrebe nahrať nové zmeny do master-kópie musí požiadavka obsahovať transakciu s novými zmenami a verzie skupín od poslednej synchronizácie. Na začiatku vybavovania tejto požiadavky sa pre daného používateľa získajú všetky skupiny s verziami a porovnajú sa s verziami v požiadavke. Na základe tohoto porovnania sa určia skupiny, ktoré používateľovi nepatria, ktoré sú nové (klientska aplikácia vykonávajúca požiadavku o nich nevie), ktoré skupiny boli aktualizované od poslednej synchronizácie a ktoré ostali nezmenené. Následne sa transakcia predá komponentu, ktorý zaobaľuje serverovú databázu, ktorá ako jediná má priamy prístup k master-kópii. Po úspešnej integrácii zmien do master-kópii sa pre každú zmenenú skupinu získajú jej používatelia a odošle sa im informácia o zmene v master-kópii. Funkcia je zobrazená na algoritme 2.

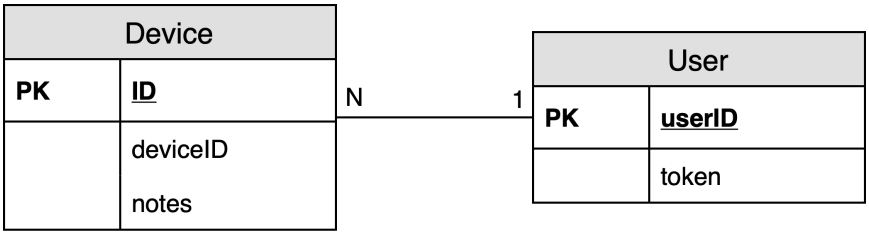
Algorithm 2 Algoritmus integrovania nových zmien v kontroléri.

Input: New updates as transaction t, set of client's groups G_c, user $user$
Output: Difference between client's copy of database and master-copy

1: $G_s \leftarrow$ dbWrapper.groupsFor($user$)
2: $G_{new} \leftarrow G_s \setminus G_c$
3: $G_{valid} \leftarrow G_s \cap G_c$
4: $diff_{updates} \leftarrow$dbWrapper.update($G_{valid}$, t)
5: $diff_{new} \leftarrow$dbWrapper.pull(G_{new})
6: $diff.groups \leftarrow diff_{updates}.groups \cup diff_{new}.groups$
7: $diff.transactions \leftarrow diff_{updates}.transactions \cup diff_{new}.transactions$
8: **return** $diff$

Distribuovanie informácie o zmene v master-kópii

Kontrolér môže mať niekoľko menších podsystémov zodpovedných za odosielanie informácii o zmene v master-kópii vo forme notifikácií (v nasledujúcom texte bude takýto podsystém označovaný ako notifikátor). Vďaka takémuto prístupu je možné mať oddelenú logiku pre posielanie notifikácií pre rôzne skupiny zariadení (napríklad iOS a android) alebo odosielanie viacerých typov notifikácií tomu istému zariadeniu (menej prioritné a viac prioritné notifikácie).



Obr. 3.4: Relačný diagram databázy využívanej notifikátorom. Entitná množina Device reprezentuje jednotlivé zariadenia používateľa. Pri každom zariadení sa udržuje informácia o jeho identifikátore a poznámka pre určenie, ktoré notifikátory môžu na dané zariadenie odoslať notifikáciu.

Každý notifikátor musí mať prístup k databáze obsahujúcej identifikátory zariadení prepojené s identifikátorom používateľa. Na základe tohoto prepojenia sa určuje, akým zariadeniam má byť zaslaná notifikácia. Aby bolo možné zdieľať túto databázu medzi viacerými notifikátormi, musí existovať doplňujúca informácia pre každé zariadenie, podľa ktorej vie notifikátor určiť, či má na dané zariadenie odoslať notifikáciu alebo nie. Relačný diagram databázy, ktorú využíva notifikátor je zobrazený na obrázku 3.4.

3.2.2 Zaobalenie serverovej databázy

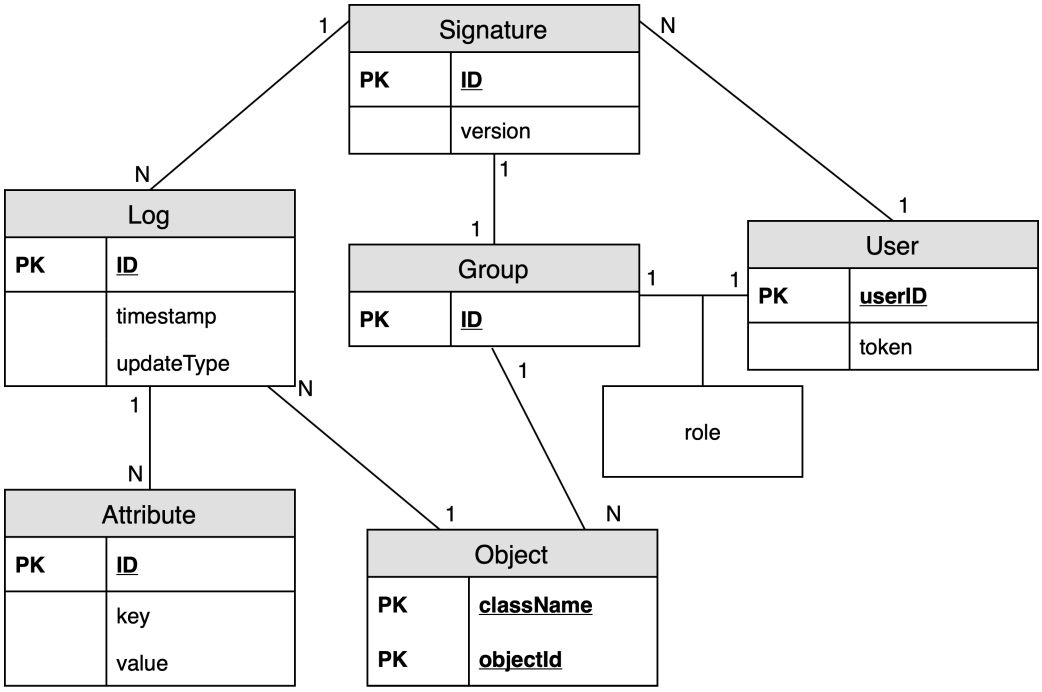
Master-kópia využívaná centrálnym prvkom musí byť perzistentná, a teda je potreba ju ukladať do databázy aj na strane servera. Nakoľko existuje veľké množstvo možností databázových systémov, ktoré môžu byť použité na strane servera, bolo potrebné zjednotiť rozhranie pre prácu s databázou a oddeliť logiku komunikácie s konkrétnym typom databázy. Navrhnuté riešenie ponúka abstraktné rozhranie, ktoré je nezávislé na konkrétnom type databázového systému, čím je možné navrhnuté riešenie prispôsobiť aj pre požadovaný databázový systém. Táto práca predpokladá využívanie databázového systému MySQL na strane servera, do ktorého sa bude ukladať master-kópia aplikačnej databázy.

Schéma serverovej databázy

Databáza na serverovej strane musí udržiavať informácie o skupinách objektov, ich členoch a jednotlivých transakciách. Relačný návrh databázy je zobrazený na obrázku 3.5. Každá transakcia spôsobí zmenu verzie skupiny objektov, ktoré mení, a preto je v relačnom návrhu entitná množina `Signature`. Jednotlivé zmeny objektov z transakcie sú reprezentované entitnou množinou `Log`. Typ zmeny určuje hodnotu `updateType`. Prípadné nové hodnoty atribútov zmenených objektov sú reprezentované entitnou množinou `Attribute`.

Navrhnuté riešenie počíta s tým, že jednotlivé informácie používateľských zariadení sú uložené v rovnakej databáze a teda výsledný relačný návrh je spojením obrázku 3.5 a 3.4.

Aby bolo možné ukladať do atribútu `value` entitnej množiny `Attribute` všetky požadované hodnoty rôznych typov, tak sa v tejto práci ukladajú všetky hodnoty atribútov ako `Data`. Číselné typy môžu byť ukladané v binárnom kódovaní, reťazec ako pole znakov a podobne.



Obr. 3.5: Relačný návrh serverovej databázy.

3.2.3 Uloženie nových zmien do master-kópie a získanie rozdielu medzi klientskou kópiou a master-kópiou databázy

Počas ukladania nových zmien do master-kópie je potrebné vykonať kontrolu na konflikty a prípadne odstrániť redundantné záznamy. Následne je potrebné získať z databázy nové zmeny a v prípade, kedy nejaká zmena nebola akceptovaná, treba vytvoriť opačnú zmenu, aby mohol klient vrátiť danú zmenu naspäť.

Riešenie konfliktov

Pri ukladaní nových zmien je potrebné získať z master-kópie všetky konfliktné zmeny. V predkladanom riešení sa využíva riešenie konfliktov na základe časovej známky danej zmeny, čo vedie k tomu, že v master-kópii budú najnovšie zmeny. Pri jednoduchých typoch ako je reťazec alebo celé číslo sa celá hodnota prepíše na novšiu. Tento princíp je pri poliach nevhodný, nakoľko sa intuitívne očakáva, že ak sú v konflikte dve rôzne pridania prvkov do jedného poľa, tak výsledkom bude pole s obidvomi prvkami a nie len s jedným.

Algoritmus pre riešenie konfliktov dvoch úprav jedného poľa navrhnutý v tejto práci potrebuje ako vstup pôvodné pole, nad ktorým sa vykonali zmeny, zmeny v poli so staršou časovou známkou (c_0) a zmeny s novšou časovou známkou(c_1). Základný princíp tohoto algoritmu je postupné prechádzanie zmien od najmenšieho indexu a následné upravovanie indexu ostatných zmien.

Prijatie novej zmeny

Prijaté zmeny sú tie, ktoré neboli konfliktné alebo vzniknutý konflikt bol vyriešený v ich prospech, a zároveň používateľ mal práva ich vykonať. Tieto zmeny sú uložené do master-

kópie databázy a ich uloženie spôsobí zmenu verzie skupín, v ktorých sú objekty ovplyvnené týmito zmenami.

Odmietnutie novej zmeny

Pre odmietnutie zmeny musí platiť, že konflikt nebol vyriešený v prospech danej zmeny alebo používateľ nie je oprávnený vykonať túto zmenu. O takejto situácii je potrebné informovať klienta, aby vykonal spätnú zmenu, a tým vrátil kópiu databázy v klientskej aplikácii do správneho stavu. Aby sa zaručilo, že klientska aplikácia vykoná správne zmeny, tak centrálny prvok tieto spätné zmeny vytvorí sám.

Získanie zmien, ktoré klient nemal synchronizované

Pri zápise nových zmien do master-kópie prebieha aj kontrola existencie zmien, o ktorých klient nevie. To je určené na základe verzií jednotlivých skupín objektov, ktoré zaslal centrálnemu prvku. Ak také zmeny existujú, tak mu sú odoslané v odpovedi. Pokiaľ bola nejaká klientska zmena zamietnutá, tak spätná zmena je odoslaná spolu so zmenami, o ktorých klient nevedel.

Redukcia logov v serverovej databáze

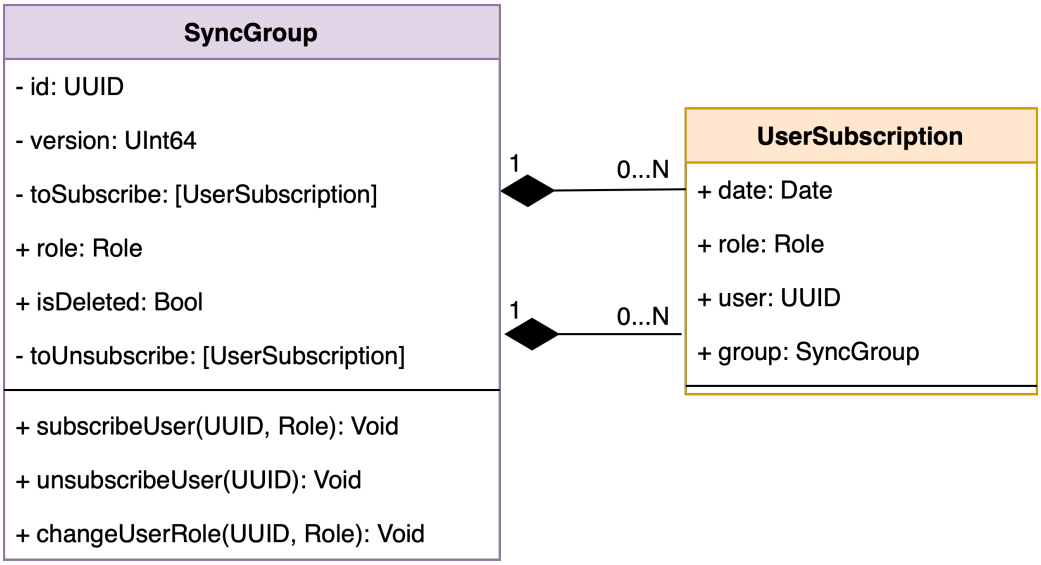
Tak ako je možné mazať redundantné zmeny v transakciách, je možné mazať tieto zmeny aj priamo v databáze, čím sa môže skrátiť čas potrebný pre získanie zmien a výrazne zmenšiť veľkosť serverovej databázy. Riešenie vytvorené v tejto práci ponúka možnosť využívať redukciu aj v databáze. Pri jej povolení sa pri vložení novej zmeny odstránia všetky redundantné staršie zmeny z databázy. V databáze ostávajú uložené zmeny určujúce vytvorenie a zmazanie objektu aby bolo zaistené, že klient bude informovaný o všetkých zmazaniach objektov.

3.3 Návrh klientskej časti

Klient predstavuje dynamickú a najviac problémovú časť synchronizačného procesu. Jednotliví klienti môžu pracovať s dátami bez internetového pripojenia, a preto je potrebné jednotlivé zmeny ukladať a synchronizovať ich až pri obnovení internetového pripojenia. V prípade slabého internetového pripojenia môže nastať prerušenie spojenia počas synchronizácie, s ktorým je potrebné sa patrične vyrovnať.

3.3.1 Zaobalenie klientskej databázy

Riešenie tejto práce predpokladá, že databáza, ktorá má byť synchronizovaná, využíva databázový systém Realm. Aby bolo možné rozšíriť riešenie práce aj pre ďalšie systémy (napríklad CoreData), bolo potrebné vytvoriť správne zaobalenie databázovej časti tak, aby bolo možné jednoducho napojiť klientske databázy na synchronizačný proces. Toto zaobalenie musí riešiť problém zisťovania zmien v klientskej databáze a byť schopné integrovať zmeny zo serverovej databázy.



Obr. 3.6: Triedny diagram `SyncGroup`.

Detekcia zmien v databáze

Realm ponúka možnosť odchytávať notifikácie o každej zmene v databáze. Notifikácie obsahujú informáciu o objektoch vytvorených, pozmenených a zmazaných. S využitím týchto notifikácií je možné vytvoriť jednotlivé pod-transakcie a ukladať ich do databázy.

Trieda SyncGroup

Keďže každý objekt musí byť v skupine objektov, je navrhnutá trieda `SyncGroup`, ktorá reprezentuje abstrakciu nad touto skupinou. Synchronizovateľný objekt musí mať atribút s týmto typom, aby bolo možné určiť, do akej skupiny daný objekt patrí.

Objekty tohoto typu majú funkcie pre pridávanie a mazanie členov v skupine a atribút s hodnotou určujúcou oprávnenia používateľa aplikácie v danej skupine. Spolu s týmito atribútmi je uschovávaný aj neverejný atribút `version` držiaci poslednú známu hodnotu verzie tejto skupiny. Triedny diagram `SyncGroup` je uvedený na obrázku 3.6. Na diagrame je zobrazená aj neverejná trieda `UserSubscription` reprezentujúca zmenu v skupine. Po dokončení synchronizácie danej zmeny je objekt automaticky zmazaný.

Objekty

Každý synchronizovateľný objekt musí mať identifikátor unikátny nie len v klientskej databáze, ale aj v rámci skupiny objektov, do ktorej patrí. Toto je možné dosiahnuť s využitím UUID definovaného v RFC 4112 [26], podľa ktorého sa jedná o globálne unikátny identifikátor. Pridaním prefixu tvoreného identifikátorom používateľa je možné zvýšiť unikátnosť identifikátora a pridať informáciu o vlastníkovi objektu.

Okrem identifikátora je potrebné pre každý objekt vedieť zistiť typ objektu (názov triedy). Tento názov je možné zistiť pomocou statickej metódy `className()` objektov databázy Realm.

Aby bolo možné zistiť identifikátor zmazaného objektu, je nutné používať princíp soft-delete. Jeho princíp je založený na pridaní atribútu vo význame príznaku označujúceho, či sa

jedná o zmazaný alebo nezmazaný objekt. Nastavením tohoto príznaku sa objekt nezmaže, ale iba pozmení, a táto zmena sa prejaví v notifikácii. Objekt bude z pamäte trvalo zmazaný až po vykonaní synchronizácie. Potreba tohoto príznaku vynucuje pri filtrovaní objektov z databázy vždy filtrovať aj s týmto príznakom, inak by používateľ navrhnutej knižnice mohol robiť s objektmi označenými ako zmazané.

Skupina objektu je udržovaná v atribúte typu `Optional<SyncGroup>`, čo dáva možnosť nenastaviť objektu skupinu. V takom prípade sa za skupinu objektu nastaví používateľova primárna skupina počas synchronizačného procesu.

Obmedzenia na objekt môžu byť implementované pomocou protokolu zobrazeného vo výpise 3.1.

```
1 protocol SynchronizableObject {
2     var id: UUID { get } // String can be used too
3     var isDeleted: Bool { get set }
4     var group: SyncGroup? { get set }
5 }
```

Výpis 3.1: Protokol určujúci obmedzenia na synchronizovateľný objekt.

Ukladanie pod-transakcií

Pri vykonaní zmien v databáze sa vytvorí z notifikácie pre každú zmenu niekoľko objektov reprezentujúcich pod-transakciu. Táto notifikácia obsahuje referenciu na pozmenené pole a tri množiny indexov:

- indexy zmazaných prvkov (tieto indexy sú neplatné voči pozmenému poľu),
- indexy nových prvkov a
- indexy modifikovaných prvkov.

Vďaka mazaniu objektov spôsobom soft-delete je možné určiť všetky atribúty pod-transakcie vyžadované pre zaručenie synchronizácie pri každej zmene databázy. Časová známka sa určí na aktuálny čas a jednotlivé zmeny v atribútoch popisovaného objektu môžu byť reprezentované ako kolekcia objektov s názvom atribútu a jeho novou hodnotou. Ku každej pod-transakcii sa uloží aj jej typ zmeny (zmazanie, vytvorenie, modifikácia). Takto vytvorené objekty sú následne uložené do databázy. Relačné znázornenie týchto objektov je na obrázku 3.7. Algoritmus transformácie zmien jedného objektu na pod-transakciu je znázornený algoritmom 3. Celkový algoritmus transformácie je znázornený na algoritme 4.

3.3.2 Životný cyklus transakcie

Každá transakcia vytvorená na strane klienta musí prejsť určitým životným cyklom. Definovanie životného cyklu vedie k jeho pochopeniu a odhaleniu slabých miest, ktoré môžu spôsobovať problémy.

Vytvorenie transakcie

Ako bolo popísané v predchádzajúcej podsekcii, každá zmena v databáze vytvára pod-transakcie, ktoré sa ukladajú do databázy. Počiatkom životného cyklu transakcie je práve uloženie prvej pod-transakcie do databázy.

Algorithm 3 Transformácia zmeny v jednom objekte na pod-transakciu.

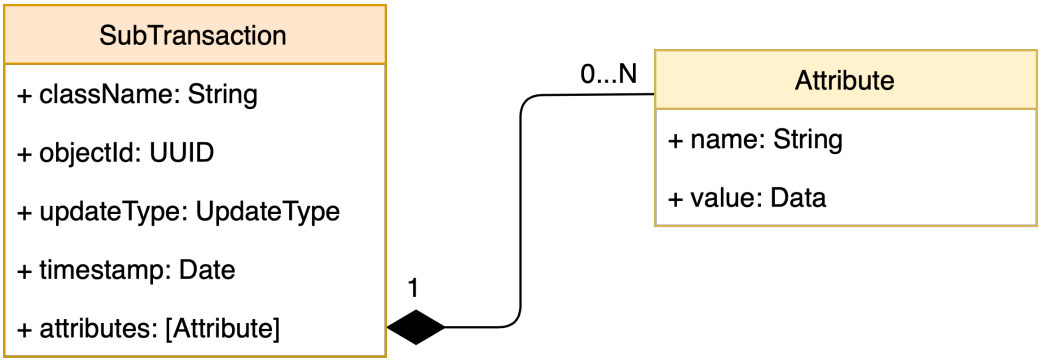
Input: Object o, update type t_u
Output: Subtransaction

$i \leftarrow 0$
$A_i \leftarrow \emptyset$
for all $label \in (scheme(o).propertiesNames \setminus \{isDeleted, id\})$ **do**
  if $t_u = modification$ **or** $label = group$ **then**
    $a.name = label$ // 'a' *is type of* `Attribute`
    $a.value =$valueToData(o, $label$) // *get value for attribute* '$label$' *in the object* 'o'
    $A_{i+1} = A_i \cup \{a\}$
    $i \leftarrow i + 1$
  end if
end for
$subtransaction.objectId = o.id$
$subtransaction.className = scheme(o).className$
$subtransaction.updateType = t_u$
$subtransaction.attributes = A_i$
return *subtransaction*

Algorithm 4 Transformácia zmien viacerých objektov do jednej transakcie. Algoritmus využíva funkciu `toSubtransaction`, ktorá je popísaná algoritmom 3.

Input: Created objects O_c and modified objects O_m
Output: Transaction

$i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$
$creations_i \leftarrow \emptyset$, $deletions_j \leftarrow \emptyset$, $modifications_k \leftarrow \emptyset$
for all $o \in O_c$ **do**
  $creations_{i+1} \leftarrow creations_i \cup \{\text{toSubtransaction}(o, creation)\}$
  $i \leftarrow i + 1$
end for
for all $o \in (O_c \cup O_m)$ **do**
  if o.isDeleted = true **then**
    $deletions_{j+1} \leftarrow deletions_j \cup \{\text{toSubtransaction}(o, deletion)\}$
    $j \leftarrow j + 1$
  else
    $modifications_{k+1} \leftarrow modifications_k \cup \{\text{toSubtransaction}(o, modification)\}$
    $k \leftarrow k + 1$
  end if
  $i \leftarrow i + 1$
end for
$transaction.deletions = deletions_j$
$transaction.creations = creations_i$
$transaction.modifications = modifications_k$
return *transaction*



Obr. 3.7: Diagram tried pre jednu pod-transakciu.

Dopĺňanie transakcie

Existencia transakcie automaticky spúšťa synchronizačný proces. V prípade, keď nie je možné vykonať synchronizáciu (napríklad nie je prístupné internetové pripojenie), synchronizácia neprebehne a je možné dopĺňať transakciu o ďalšie pod-transakcie ich ukladaním do databázy.

Ukončenie životného cyklu transakcie

Pri začatí synchronizácie sa zmeny načítajú do štruktúry typu `Log`, ktorá reprezentuje práve synchronizovanú transakciu a všetky zmeny sa označia ako práve synchronizované (algoritmus 5). Následne sa nastavia prípadne chýbajúce skupiny objektov týchto zmien a určí sa identifikátor identifikujúci štruktúru `Log`. V prípade, že odoslanie synchronizácie zlyhá (napríklad nekvalitné pripojenie na internet), musí sa transakcia označiť ako nesynchronizovaná a synchronizačný proces sa preruší, nakoľko nie je možné zaručene prehlásiť, či server obdržal celú transakciu. Pokiaľ by sa aj napriek takejto chybe transakcia ukončila a zmazala z databázy, stratili by sa údaje o všetkých zmenách, ktoré transakcia obsahovala. Z toho vyplýva, že životný cyklus transakcie je možné ukončiť až po úspešnom odoslaní transakcie a obdržaní odpovede od servera. Celý tento postup je vyjadrený sekvenčným diagramom na obrázku 3.8.

Algorithm 5 Získanie zmien určených pre synchronizáciu.

Input: Nothing
Output: Transaction as `Log`

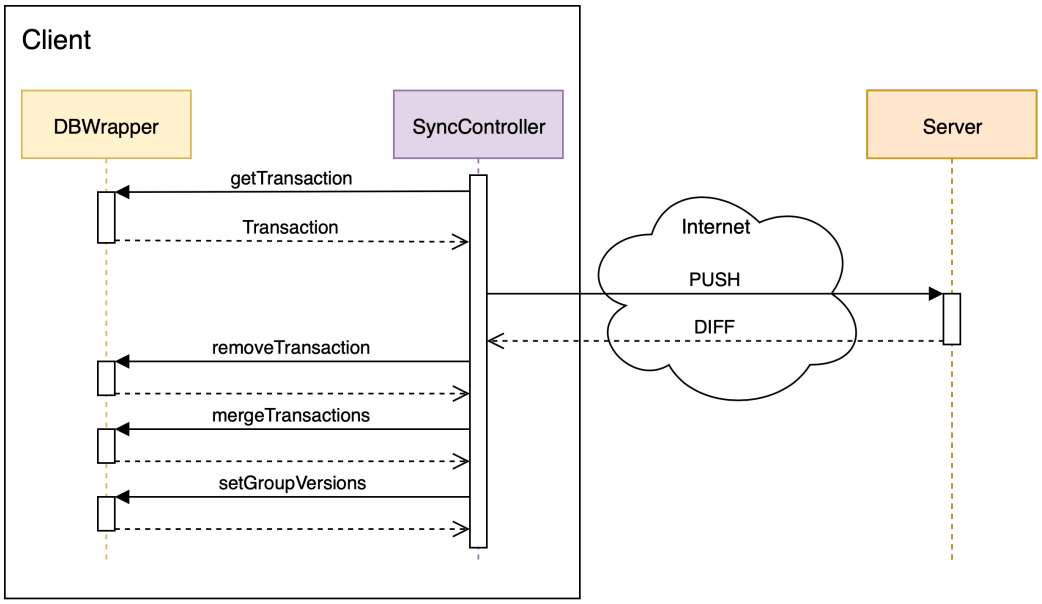
1: $subtransactions \leftarrow \{s \in \text{dbWrapper.loadSuntransactions}() \mid s.inSync = false\}$
2: **for all** $s \in subtransactions$ **do**
3: $\quad s.inSync \leftarrow true$
4: **end for**
5: $log.id \leftarrow$UUID.new()
6: $log.deletions \leftarrow \{s \in subtransactions \mid s.updateType = deletion\}$
7: $log.creations \leftarrow \{s \in subtransactions \mid s.updateType = creation\}$
8: $log.modifications \leftarrow \{s \in subtransactions \mid s.updateType = modification\}$
9: dbWrapper.update($subtransactions$)
10: dbWrapper.store(log)
11: **return** log

Transakcie obdržané od centrálneho prvku

Životný cyklus týchto transakcií začína na centrálnom prvku a je výrazne kratší, nakoľko nie je potreba transakcie ukladať na strane klienta. Po ich obdržaní sa okamžite integrujú do lokálnej databázy. Pokiaľ má používateľ nezosynchronizované zmeny, ktoré sú v konflikte s obdržanou transakciou, tak je konflikt vyriešený podľa rovnakých pravidiel ako je popísané v sekcii 3.2.3.

3.3.3 Synchronizačný kontrolér klienta

Synchronizačný kontrolér je zodpovedný za úlohy zaisťujúce synchronizáciu dát. Kontrolér vykonáva sieťovú komunikáciu s centrálnym prvkom, ktorému zasiela zmeny v lokálnej databáze, zmeny v skupinách a prijíma nové zmeny v master-kópii. Zaslanie nových zmien je zobrazené na sekvenčnom diagrame na obrázku 3.8. Kontrolér si najprv vyžiada zmeny vo forme transakcie a tie odošle na server. V odpovedi servera sa môžu nachádzať nové zmeny, ktoré sa musia zaintegrovať spolu s novými verziami skupín.



Obr. 3.8: Sekvenčný diagram zobrazujúci činnosti klientského synchronizačného kontroléra pri zasielaní transakcie na server.

Propagácia chyby

Počas celého synchronizačného procesu je veľké množstvo miest, kde môže vzniknúť chyba či už pri sieťovej komunikácii alebo pri práci s databázou. Kontrolér zbiera z celej časti klientskej knižnice MeerkatSync všetky chyby a dáva možnosť ich sledovať. Rôzne objekty sa môžu prihlásiť k sledovaniu chybových hlásení a na základe nich môžu užívateľovi zobrazovať chybové správy.

Nastavenie primárnej skupiny používateľa

Bez prihlásenia používateľa do systému centrálneho prvku nie je možné vedieť identifikátor jeho primárnej skupiny. Takéto obmedzenie by mohlo viesť k vyžadovaniu prihlásenia

používateľa pre používanie aplikácie. Aby bolo možné aplikáciu používať aj bez nutnosti prihlásiť sa do systému a získať identifikátor primárnej skupiny, využíva sa v synchronizačných procesoch iný (prednastavený) identifikátor. Tento identifikátor je automaticky nahradený v odosielanej správe za skutočný. V prijatých správach je zasa automaticky nahradený naspäť za prednastavený. Celý tento proces je zaznačený pomocou pseudokódu v algoritme 6.

Algorithm 6 Komunikácia so serverom s korekciou identifikátora primárnej skupiny.

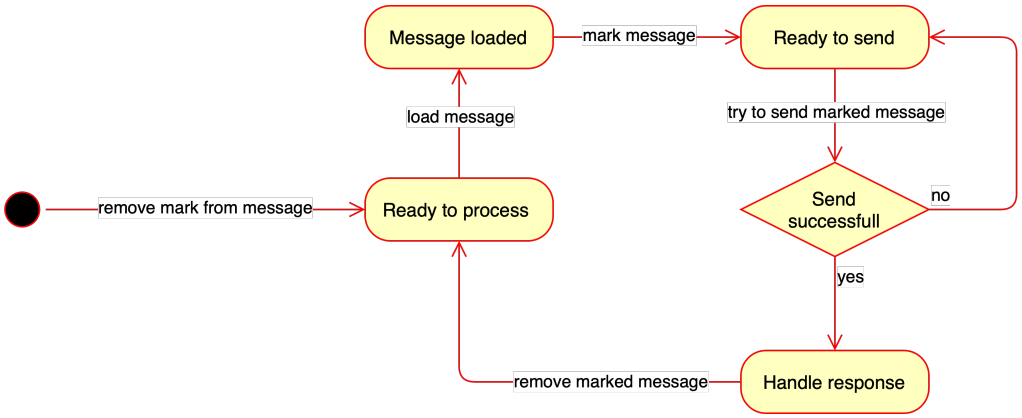
Input: Message m_d, real primary group identifier $PGID$
Output: Response from server

1: $m_p \leftarrow$ replace all occurence of $DefaultPrimaryGroupID$ in m_d with $PGID$
2: $r_p \leftarrow$ server.send(m_p)
3: $r_d \leftarrow$ replace all occurence of $PGID$ in r_p with $DefaultPrimaryGroupID$
4: **return** r_d

Životný cyklus kontroléra

Kontrolér tvorí kritickú časť aplikácie, a preto je potrebné presne definovať, v akých stavoch sa môže nachádzať. Na obrázku 3.9 je zobrazený stavový diagram kontroléra. Diagram zobrazuje, že môže byť spracovávaná v jednom čase iba jedna správa. Kontrolér načíta jednu správu, ktorú označí ako práve spracovávanú. Túto správu sa následne pokúša odoslať na server, ktorý ju spracuje. pri vzniku chyby počas odosielania (napríklad pri prerušení pripojenia na sieť) sa kontrolér pokúsi odoslať správu znova. Po úspešnom odoslaní sa spracuje odpoveď, označená správa sa zmaže z databázy a načíta sa nová.

Kontrolér počíta aj s prípadným pádom aplikácie počas spracovávania požiadavky. Kvôli takémuto prípadu vykonáva kontrolér na začiatku svojho životného cyklu prípadné zrušenie značky zo správy, aby bolo umožnené danú správu načítať znova.



Obr. 3.9: Stavový diagram popisujúci stavy klientskeho synchronizačného kontroléra

3.3.4 Obsluha notifikácií

MeerkatSync využíva notifikácie iba ako doplňujúci spôsob zisťovania, či prebehli nejaké zmeny v master-kópii, o ktorých aplikácie nevie. Pri obdržaní notifikácie zariadením musí

aplikácia v čo najkratšom čase vykonať jej obsluhu a oznámiť dokončenie obsluhy. V zariadeniach s operačnými systémami od firmy Apple sa oznámenie vykonáva zavolaním bloku kódu obdržaného spolu s notifikáciou. Keďže používateľ využívajúci navrhnutú knižnicu môže požadovať využiť notifikáciu aj pre vykonanie iných činností, je potrebné vyvolať dokončenie až po dokončení synchronizácie a všetkých ďalších činností. Z tohoto dôvodu sa pri vyvolaní synchronizácie vráti blok kódu, ktorý nahrádza (zaobaľuje) pôvodný blok indikujúci dokončenie obsluhy notifikácie. Tento nový blok kódu zaručuje, že dokončenie obsluhy notifikácie bude zavolané až po ukončení všetkých činností. Obsluha notifikácie môže vyzerať tak ako je popísané v algoritme 7.

Algorithm 7 Obsluha notifikácie.

Input: Notification n and callback c
Output: Nothing

1: $newCallback \leftarrow$ pullForNotification(n, c) *// asynchronously download new updates*
2: do something else
3: $newCallback()$ *// callback c will be called when download is completed*

3.4 Návrh protokolu

Navrhnutý protokol je určený pre synchronizáciu dát jedného používateľa alebo skupiny používateľov. Zmeny v databáze sa vymieňajú v podobe logov, ktoré sa pri synchronizácii interpretujú.

3.4.1 Správy protokolu

V protokole sa používa niekoľko správ. Hlavnými informačnými jednotkami sú pod-transakcia, transakcia a skupina s verziou.

Pod-transakcia

Pod-transakcia (výpis 3.2) sa viaže k jednému konkrétnemu objektu uloženému v databáze. Popisuje konkrétne zmeny v jednom časovom okamihu nad týmto objektom. Okrem časovej známky obsahuje úplnú identifikáciu objektu skladajúcu sa z názvu triedy a identifikátora objektu. Štruktúra obsahuje aj zmeny jednotlivých atribútov.

Zmeny atribútov sú reprezentované štruktúrou obsahujúcou názov atribútu a jeho novú hodnotu zakódovanú ako `Data?`.

```
1  public protocol AttributeUpdate {
2
3      var attributeName: String { get }
4      var newValue: Data? { get }
5  }
6
7  public protocol SubTransaction {
8
9      var className: String { get }
10     // ‘objectId‘ can be ‘String‘ or ‘UUID‘
11     var objectId: UUID { get }
```

```
12      var timestamp: Date { get }
13      var updates: [AttributeUpdate] { get }
14  }
```

Výpis 3.2: Protokol pre pod-trasakciu (`SubTransaction`) a zmeny atribútov (`AttributeUpdate`).

Transakcia

Ako bolo zmienené v predchádzajúcich sekciách, transakcia rozdeľuje pod-transakcie podľa významu na tri skupiny. Protokol transakcie je uvedený vo výpise 3.3. Okrem tohoto rozdelenia využíva synchronizačný protokol aj nasledujúce sémantické obmedzenia:

- každá podtransakcia v `deletions` (zmazania) neobsahuje žiadne zmeny atribútov,
- každá podtransakcia reprezentujúca vytvorenie nového objektu je spojená aj s identifikátorom skupiny objektov, do ktorej sa má nový objekt pridať,
- pokiaľ by mal novovytvorený objekt obsahovať atribút s referenciu na iný objekt vytvorený rovnakou transakciou, musí byť táto zmena v `creations` vynechaná a nastavenie hodnoty tohoto atribútu sa vykoná pomocou novej pod-transakcie pridanej do modifikácií (`modifications`). Toto obmedzenie zaručuje, že pri určovaní referencie na iný objekt už bude tento objekt vytvorený.

```
1  public protocol CreateSubTransaction {
2
3      var group: UUID { get } // or String
4      var object: SubTransaction { get }
5  }
6
7  public protocol Transaction {
8
9      var deletions: [SubTransaction] { get }
10     var creations: [CreateSubTransaction] { get }
11     var modifications: [SubTransaction] { get }
12 }
```

Výpis 3.3: Protokol pre transakciu.

Správa PUSH

Slúži k zaslaniu nových zmien v aplikácii centrálnemu prvku. Správa obsahuje kolekciu skupín spolu s ich verziou a transakciu s novými zmenami. Protokol je uvedený vo výpise 3.4.

```
1  public protocol Group {
2
3      var id: UUID { get } // or String
4      var version: UInt64 { get }
5  }
6
```

```
7  public protocol Push {
8
9      var groups: [Group] { get }
10     var transaction: Transaction { get }
11 }
```

Výpis 3.4: Protokol popisujúci skupinu (`Group`) a správu PUSH (`Push`).

Správa DIFF

Správa je určená k distribúcii rozdielu medzi serverovou master-kópiou a klientskou kópiou databázy. Obsahuje transakcie so zmenami, ktoré si klient musí integrovať, aby bola jeho kópia databázy zhodná s master-kópiou a stavy skupín, v ktorých používateľ je.

Stav skupiny obsahuje jej verziu a práva používateľa v tejto skupine. Klient pomocou porovnania týchto stavov s jeho lokálnou kópiou vie zistiť, či sú nejaké skupiny nové alebo či bol z nejakej odstránený.

```
1  public protocol GroupState {
2      var groupId: UUID { get } // or String
3      var version: UInt64 { get }
4      var role: Role { get }
5  }
6
7  public protocol Diff {
8      var transactions: [Transaction] { get }
9      var groups: [GroupState] { get }
10 }
```

Výpis 3.5: Protokol popisujúci stav skupiny (`GroupState`) a správu DIFF (`Diff`).

Správa PULL

Pre stiahnutie zmien zo serveru je potrebné poslať na server kolekciu obsahujúcu štruktúry implementujúce protokol `Group`. Server následne určí všetky nové rozdiely a na túto správu vhodne odpovie pomocou správy DIFF.

Správa GET_INFO

Klient vie pomocou tejto správy zistiť, v akých skupinách je. Odpoveď je tvorená správou INFO.

Správa INFO

Správa INFO obsahuje kolekciu tvorenú štruktúrami implementujúce `GroupState`.

Správa CREATE_GROUP

Slúži k vytvoreniu novej skupiny. Správa obsahuje identifikátor novej skupiny. Novovytvorená skupina implicitne obsahuje jedného používateľa – tvorcu skupiny.

Správa UNSUBSCRIBE

Správa obsahuje identifikátor skupiny, používateľa a je určená k odstráneniu používateľa z danej skupiny.

Správa SUBSCRIBE

Noví používatelia môžu byť pridaní do skupiny pomocou správy SUBSCRIBE. Správa obsahuje rovnaké atribúty ako správa UNSUBSCRIBE doplnené o oprávnenia používateľa v tejto skupine. Ak užívateľ už je v skupine, tak sa aktualizujú práva používateľa.

Správa SYNC

Server môže notifikovať klienta o vzniknutých zmenách v databáze pomocou správy SYNC čím server signalizuje, aby si klient stiahol nové zmeny.

3.4.2 Výmena správ

Navrhnutý protokol je určený pre synchronizáciu medzi serverom a klientom (klienti sa medzi sebou môžu synchronizovať jedine pomocou servera). Komunikácia je vždy inicializovaná klientom a je vo forme požiadavka-odpoveď. Požiadavka je tvorená správou a autorizačným tokenom pre overenie klienta. Odpoveď je tvorená iba správou. Jedinou výnimkou je správa SYNC, ktorá by mala byť doručená klientovi s využitím notifikácie. Keďže protokol je bezstavový je možné ho využiť v bezstavových protokoloch ako je napríklad REST API.

3.4.3 Riešenie konfliktov

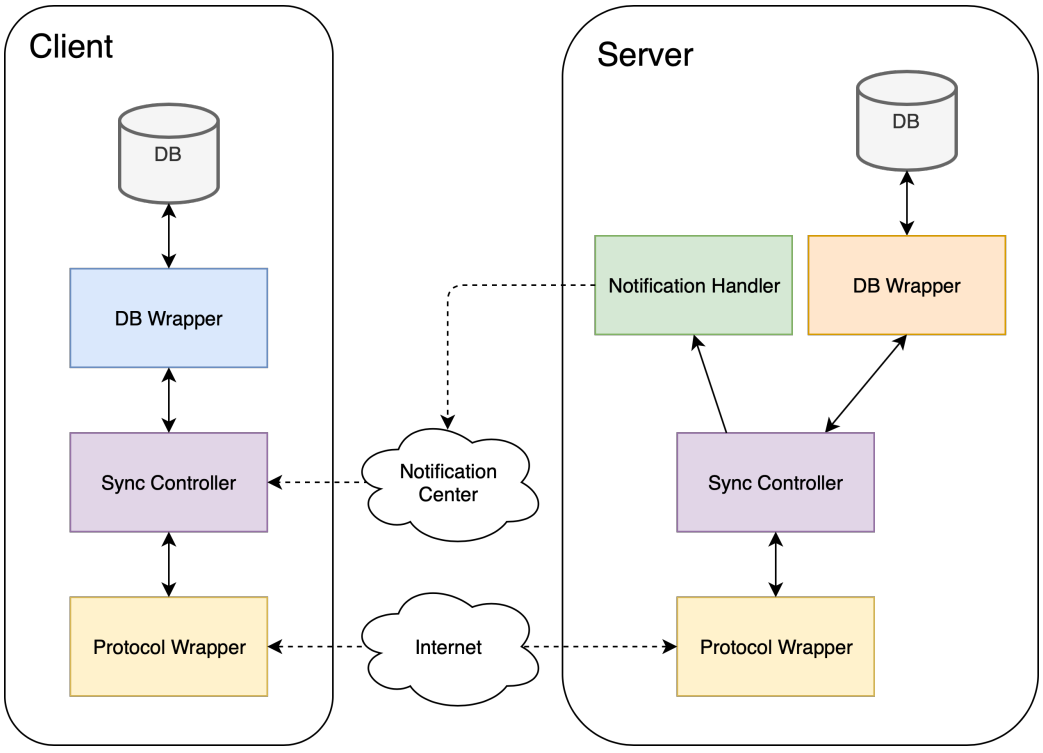
Navrhnutý protokol nemá v sebe zabudovaný, žiadny priamy spôsob riešenia konfliktov. Konflikt je možné vyriešiť priamo na zariadení, na ktorom bol detekovaný. Jedna z možných stratégií riešenia konfliktu je, že v prípade konfliktu sa použije novšia verzia. Ďalšou vhodnou stratégiou je, že v prípade konfliktu sa použijú zmeny uložené na servery.

Pokiaľ konflikt vznikol pri použití správy PULL, mal by byť vyriešený na strane klienta a klient môže informovať o riešení server pomocou správy PUSH. Pokiaľ konflikt vznikol naopak na serveri pri odpovedaní na správu PUSH, je možné zaslať riešenie konfliktu klientovi v správe DIFF.

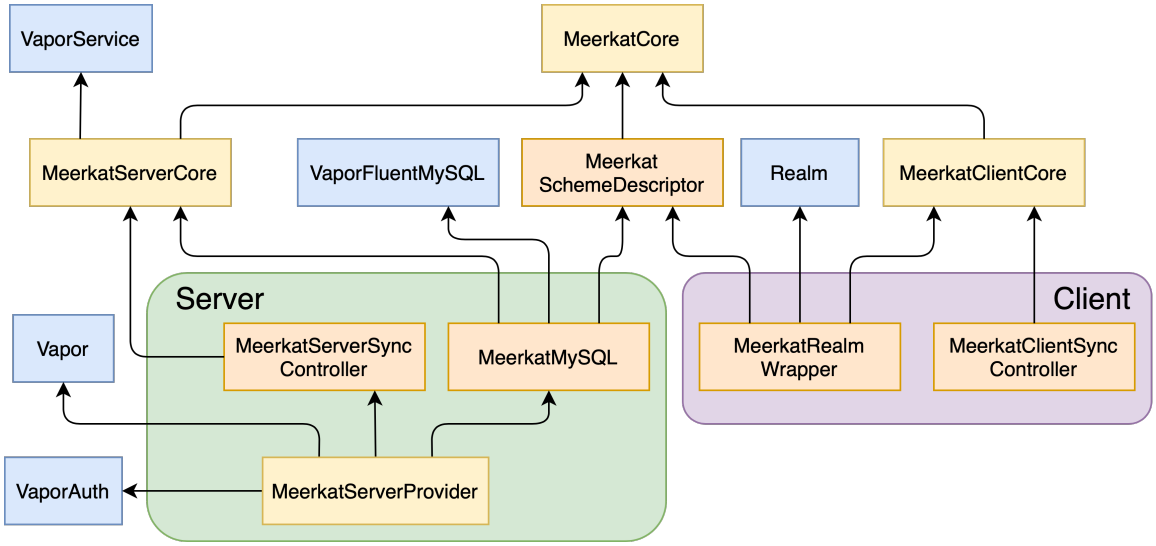
Kapitola 4

Implementácia

Táto práca bola implementovaná s veľkým dôrazom na správne oddelenie logiky jednotlivých častí, a teda na správnu modularitu a to tak, aby jednotlivé časti mohli byť použité samostatne alebo prípadne nahradené za iné. Pri implementácii boli využívané existujúce riešenia a knižnice pre efektívnejšiu a bezpečnejšiu implementáciu samostatných modulov. Základné moduly sú zobrazené na obrázku 4.1.



Obr. 4.1: Prepojenie jednotlivých častí navrhnutých v tejto práci. Popis: žltá – knižnice implementujúce navrhnutý protokol, fialová – knižnice riadiace synchronizáciu, modrá: zaobalenie klientskej databázy (Realm alebo CoreData), oranžová – zaobalenie serverovej databázy, zelená – knižnica zaisťujúca odosielanie notifikácií.



Obr. 4.2: Závislosti medzi jednotlivými knižnicami implementovanými v tejto práci. Význam farieb je následovný: modrá – knižnice tretích strán, žltá – knižnice popisujúce protokoly alebo slúžiace ako spojenie viacerých knižníc, oranžová – knižnice implementujúce časť riešenia nad konkrétnou technológiou alebo princípom, zelená – serverové knižnice, ktoré je možné nahradiť po implementovaní určitých protokolov, fialová – klientske knižnice, ktoré je možné nahradiť po implementovaní určitých protokolov.

4.1 Implementované knižnice

Celková implementácia je rozdelená do deviatich knižníc. Niektoré knižnice sú zamerané na definíciu protokolov a obsahujú funkcie, pri ktorých sa očakáva, že budú používané vo viacerých knižniciach a iné knižnice priamo implementujú tieto protokoly. Medzi jednotlivými knižnicami vznikajú závislosti, ktoré sú uvedené na obrázku 4.2. Vďaka dobre definovaným protokolom určujúcich rozhranie je možné jednoducho vymeniť knižnice v zelenom a fialovom obdĺžniku (napríklad vymeniť Realm za CoreData alebo MySQL za iný databázový systém).

4.1.1 Integrovanie knižníc

Knižnice sú implementované tak, aby ich bolo možné integrovať do iných projektov pomocou Swift Package Manager (SPM). Z tohoto dôvodu sú zdrojové kódy voľne prístupné na verzovacom systéme gitlab[1]. Pomocou SPM je určená aj minimálna verzia cieľového systému a aj jazyka swift. Serverové knižnice je možné skompilovať na podporovaných verziách Ubuntu a na operačnom systéme MacOS s verziou 10.15+. Pre klientske knižnice je minimálna podporovaná verzia operačného systému iOS13 a iPadOS13. Minimálna verzia jazyka Swift pre kompiláciu je 5.1

[1] https://gitlab.com/MeerkatSync

4.2 Implementácia servera

Na strane servera bolo potrebné naimplementovať tri základné samostatné časti, ktoré spolu komunikujú. Správne zaobalenie databázy (wrapper) tak, aby bolo čo najmenej viazané na konkrétny typ použitej databázy. Synchronizačný kontrolér riadiaci synchronizačný proces, a teda prístup klientov k jednotlivým zdrojom uložených v databáze s využitím daného wrappera a spracovávanie jednotlivých požiadaviek klienta za použitia daného synchronizačného kontroléra.

Pri implementácii bol primárne využívaný framework Vapor, ktorý obsahuje komplexné serverové riešenie v jazyku Swift. Kvôli veľkému množstvu asynchrónneho kódu sa v tomto frameworku bohato využíva princíp Future-Promise.

4.2.1 Riadenie synchronizácie

Synchronizačný kontrolér je určený k riadeniu celého synchronizačného procesu. Musí zabezpečovať konkurenčný prístup klientov k zdrojom v databáze tak, aby sa nestalo, že v strede aktualizácie objektov sa odošlú neúplné inému klientovi. Kontrolér ďalej spracováva požiadavky od klientov a pomocou databázového wrappera ich vykonáva. Na obrázku 3.3 je zobrazené rozhranie kontroléra pomocou diagramu tried. Celý synchronizačný kontrolér je naimplementovaný v knižnici MeerkatServerSyncController.

Notifikovanie klientov

Po obslúžení klienta, pri ktorom vznikla zmena v databáze, je potrebné informovať pomocou notifikácie všetkých klientov, ktorých sa zmena týkala. Aby bolo možné využívať viacej notifikačných služieb, kontrolér obsahuje pole jednotlivých notifikátorov zodpovedných za odoslanie notifikácie. Kontrolér má funkciu `notify(groups:)`, pomocou ktorej notifikuje všetkých používateľov v danej skupine. Pre notifikovanie používateľov slúži funkcia `notify(users:)`, v ktorej sa na základe parametra `users` odošle notifikácia na všetky zariadenia používateľov[2].

Uzamknutie zdrojov

Aby sa zaručila maximálna možnosť využívať kontrolér bezpečne aj z viacerých vlákien tak boli vytvorené triedy `FutureLocks` a `PromiseLock`.

Synchronizačný kontrolér si drží referenciu na objekt typu `FutureLocks`, ktorý slúži pre uzamykanie jednotlivých skupín. Tento objekt disponuje privátnym asociatívnym poľom chráneným mutexom, v ktorom je názov skupiny kľúč a inštancia objektu typu `PromiseLock` ako hodnota. Tento objekt ponúka funkcie pre dva typy uzamknutia a odomknutia skupín. Uzamknutie pre čítanie zo skupiny a uzamknutie pre zápis do skupiny.

V týchto funkciách sa vždy najprv uzamkne mutex pre prístup k asociatívnemu poľu a načítajú sa objekty typu `PromiseLock` pre požadované skupiny. Následne sa z týchto objektov získajú prísľuby, že niekedy v budúcnosti bude daná skupina rezervovaná pre požadovanú operáciu (čítanie alebo zápis) spolu s identifikátorom prísľubu. Po získaní všetkých prísľubov sa mutex odomkne a nastaví sa zavolanie požadovaného bloku kódu, ktorý sa má zavolať keď budú všetky skupiny rezervované pre daný proces. Po dokončení bloku sa s využitím identifikátora prísľubu uvoľnia skupiny a môžu byť rezervované pre ďalšie procesy.

[2]Pokiaľ každý notifikátor odmietne obslúžiť nejaké zariadenie tak sa na toto zariadenie nič neodosiela.

Samotná správa rezervácií je obstarávaná v objektoch typu `PromiseLock`, ktoré si držia dve polia, v ktorých sa požiadavky pre čítanie a zápis radia do fronty spolu s poradovým číslom, podľa ktorého sa určuje, z ktorej fronty sa požiadavka vyberie.

4.2.2 Serverová databáza

V tejto práci je naimplementované rozhranie pre správu master-kópie aplikačnej databázy uloženej v databázovom systéme MySQL. Pre prácu s týmto systémom sa využíva knižnica Fluent z frameworku Vapor, vďaka čomu nie je problematické doplniť podporu aj pre iné databázové systémy.

Celá práca potrebná pre obsluhu master-kópie na databázovej úrovni je implementovaná v knižnici MeerkatMySQL. Knižnica obsahuje triedu s rovnakým názvom (`MeerkatMySQL`) implementujúcu rozhranie `DBWrapper` a `NotificatorDBWrapper`, čím je možné ju využívať pre prácu s master-kópiou a aj ako databázu v notifikátore.

Ukladanie logov

Knižnica umožňuje dve stratégie ukladania logov. V prvej sa ukladajú všetky logy a nevykonávajú sa žiadne optimalizácie. Výhodou je možnosť histórie zmien za cenu rýchlo rastúcej veľkosti dát uložených v databáze. Druhá stratégia redukuje a maže redundantné logy. Stratégia je vybraná na základe parametru pri inicializácii objektu `MeerkatMySQL`.

Redukcia sa vykonáva po aktualizácii každého objektu. Pokiaľ bol objekt zmazaný, sú zmazané všetky logy modifikujúce daný objekt a v databáze ostanú iba dva záznamy (záznam o vytvorení a zmazaní). Tieto záznamy sú potrebné aby bolo možné rozhodnúť o tom či je potrebné daný log poskytnúť klientovi pri synchronizácii. Pri modifikácii objektu sa zmažú všetky predchádzajúce zmeny aktualizovaných atribútov. Výnimku tvoria atribúty, ktorých typ je pole, keďže pri riešení konfliktov je potrebné vedieť, aké prvky malo dané pole v čase poslednej synchronizácie.

Definícia tabuliek

Jednotlivé tabuľky v databáze sú definované pomocou knižnice FluentMySQL a Fluent. Pre jednotlivé entity v relačnom diagrame z obrázku 3.5 sú vytvorené štruktúry s požadovanými atribútmi. Ako identifikátor entity je zvolený dátový typ `Int`. Vzťahy medzi tabuľkami sú definované pomocou atribútov s generickým typom `Siblings`, `Children` alebo `Parent`. Definícia tabuľky pre skupiny a jej vzťahov je uvedená vo výpise 4.1. Knižnica Fluent ponúka aj možnosť definovať migrácie tabuliek pre ich automatické vytvorenie, modifikáciu alebo pre uloženie počiatočných dát v tabuľke[3]. Migráciu je možné definovať pomocou protokolu `MySQLMigration`, ktorý má prednastavenú implementáciu migrácie pre vytvorenie tabuľky.

```
1 import FluentMySQL
2
3 struct GroupTable {
4     var id: Int?
5     let name: String
6
7     static var name = "sync_group"
8     static var entity = "sync_groups"
```

[3]Bežne sa uloženie počiatočných hodnôt v tabuľke označuje ako seedovanie.

```
}

extension GroupTable {
    var users: Siblings<GroupTable, UserTable, RoleTable> {
        siblings()
    }

    var signatures: Children<GroupTable, SignatureTable> {
        children(\.groupId)
    }

    var objects: Children<GroupTable, ObjectTable> {
        children(\.groupId)
    }
}

extension GroupTable: MySQLModel { }
extension GroupTable: MySQLMigration { }
```

Výpis 4.1: Definícia databázovej tabuľky pre skupiny.

4.2.3 Integrácia s frameworkom Vapor

Vapor má veľkú podporu pre využívanie kódu tretích strán pomocou protokolu `Service` a `Provider`. Protokoly predstavujú poskytovateľov služieb, ktoré je možné za behu programu využívať. Implementované serverové knižnice implementujú patričné protokoly, a tým je možné využívať knižnice bez problémov.

Registrácia služieb

Za účelom zjednotenia všetkých služieb využívaných v knižniciach je vytvorená trieda `MeerkatServerProvider`, vďaka ktorej je pre začlenenie systému do serverovej aplikácie potrebné napísať iba šesť riadkov kódu. Telo konfiguračnej funkcie `configure` Vaporu by malo obsahovať okrem iného aj kód uvedený vo výpise 4.2. Na prvom riadku sa inštancuje poskytovateľ služieb s modelom `UserTable`, ktorý reprezentuje používateľov v systéme a na druhom sa registrujú všetky služby poskytovateľa. Riadok tri registruje schému objektov ukladaných v master kópii.. Na riadku desať sa nastavia migrácie tabuliek. Cesty aplikačného rozhrania používané pri synchronizácii sa pridávajú na riadku šestnásť.

```
let prov = MeerkatServerProvider<UserTable>()
try services.register(prov) // register meerkat provider
services.register([Person.self, Dog.self].scheme) // register master-
    copy scheme

// register database

var mig = MigrationConfig()
prov.setUpMigration(&mig) // set up meerkat tables migration
// register other models
```

```
10 services.register(mig)
11
12 services.register(EmptyNotificator.self) // register some notificator
13
14 let router = EngineRouter.default()
15 try routes(router)
16 prov.setUpRoutes(router) // add meerkat sync routes
17 services.register(router, as: Router.self)
```

Výpis 4.2: Príklad kódu potrebného pre využívanie synchronizácie na strane servera.

Aplikačné rozhranie synchronizácie

Rozhranie obsahuje šesť ciest k zdrojom, ktoré môžu klienti využívať. Jedná sa o dve synchronizačné cesty:

- **PATCH** metóda s cestou */sync* určená pre správu PUSH,
- **POST** metóda s cestou */sync* určená pre správu PULL,

a pre správu skupín sa pridajú následovné cesty:

- **DELETE** metóda s cestou */groups/:id*, pre zmazanie skupiny s identifikátorom *:id*,
- **DELETE** metóda s cestou */groups/:groupId/subscribers/:userId*, pre odstránenie používateľa s identifikátorom *:userId* zo skupiny *:groupId*,
- **POST** metóda s cestou */groups/:id*, pre pridanie skupiny s identifikátorom *:id*,
- **POST** metóda s cestou */groups/:groupId/subscribers/:userId*, pre pridanie používateľa s identifikátorom *:userId* do skupiny *:groupId*.

Pre prístup k týmto cestám sa vyžaduje, aby sa klient autentifikoval pomocou tokenu. Pre vytvorenie nového používateľa je možné využiť pripravené rozhranie, v ktorom je potrebné určiť ako sa má vytvoriť alebo je možné spravovať používateľa samostatne. Vytvorenie používateľa môže vyzerať obdobne ako je zobrazené vo výpise 4.3.

```
1  func create(_ req: Request) throws -> Future<HTTPStatus> {
2      let user = try req.content.decode(PublicUser.self)
3      let password = user.map {
4          try req.make(BCryptDigest.self).hash($0.password)
5      }
6      let newUser = user.and(password).map { UserTable(email: $0.0.email,
           password: $0.1) }
7      let userNotExists = { /* check if user is not already registered */
           }
8      let createUser = {
9          req.withPooledConnection(to: .mysql) { newUser.save(on: $0) }
10     }
11     let createSyncUser = {
12         let sync = try req.make(ServerSyncController.self)
```

```
13          return createUser().flatMap { user in
14              sync.createUser(with: user.syncId).flatMap { sync.createGroup
                    (with: user.syncId, objects: [], by: $0) }
15                  .transform(to: .ok)
16          }
17      }
18      return userNotExists().flatMap { try createSyncUser() }
19  }
```

Výpis 4.3: Možná implementácia registrácie používateľa.

4.3 Implementácia klienta

Knižnice vytvorené pre prácu so synchronizačným procesom na klientskom zariadení sú založené na knižnici Combine a Realm. V systéme figuruje trieda implementujúca protokol `SyncController`, ktorá vytvára štruktúru dátového toku. Pomocou tejto triedy je možné riadiť základné činnosti synchronizačného procesu. Trieda ponúka metódy pre pozastavenie odosielania zmien na server, vyčistenie všetkých synchronizovaných objektov a logov. Pre sledovanie chýb vzniknutých počas synchronizačného procesu ponúka atribút typu `Publisher`.

4.3.1 Stiahnutie nových zmien

Počas behu aplikácie je často potrebné skontrolovať, či server neobsahuje novšiu verziu master-kópie ako má klient stiahnutú. Pre účel tejto kontroly a prípadného stiahnutia bola naimplementovaná trieda `SyncPullRequest`, ktorá má statické funkcie pre jednoduché vyvolanie požiadavky v synchronizačnom procese. Požiadavku je možné vyvolať z každého miesta kódu zavolaním statickej funkcie `send()`.

Obsluha notifikácie

Notifikácie predstavujú podporný mechanizmus pre detekovanie zmien v serverovej databáze. Pre obsluhu notifikácie ponúka `SyncPullRequest` statickú funkciu, ktorá na základe notifikácie stiahne nové zmeny zo servera. Pokiaľ notifikácia indikuje potrebu stiahnuť zmeny, tak sa odošle požiadavka do synchronizačného procesu. Okrem notifikácie berie funkcia aj closure, ktorou sa indikuje koniec obsluhy notifikácie. V návratovej hodnote je nová closure, ktorá plne nahrádza pôvodnú a vývojár by ju mal zavolať pri dokončení svojej obsluhy notifikácie. Vďaka tomuto je garantované, že pôvodná closure bude zavolaná až po tom, ako sa dokončí sťahovanie nových zmien a skončí aj vývojárová obsluha notifikácie. Vo výpise 4.4 je kód pre obsluhu notifikácie. Tento kód pomocou lokálnej notifikácie odošle synchronizačnému procesu požiadavku na stiahnutie nových dát zo serveru.

```
1 func application(_ application: UIApplication,
2     didReceiveRemoteNotification userInfo: [AnyHashable : Any],
3     fetchCompletionHandler completionHandler: @escaping (
          UIBackgroundFetchResult) -> Void) {
4
5     let completionHandler = SyncPullRequest.wrapRemoteNotification(from:
          userInfo, with: completionHandler)
```

```
6     // do something
7     completionHandler(.noData)
8 }
```

Výpis 4.4: Obsluha vzdialenej notifikácie vo funkcii triedy `UIApplicationDelegate`. Pokiaľ notifikácia obsahuje dáta indikujúce, že je potrebné stiahnuť zo servera nové zmeny, odošle sa požiadavka synchronizačnému procesu, ktorý ju obslúži.

Obsluha aktualizácie v pozadí

Vývojár môže určiť pravidelný interval, po ktorom sa aplikácia pokúsi stiahnuť nové zmeny. Obdobne ako pri notifikáciách je potrebné po dokončení obsluhy zavolať closure danú parametrom. Vďaka tejto podobnosti je možné obslúžiť požiadavku podobne s rovnakým nahradením closure indikujúcej koniec obsluhy. Vo výpise 4.5 je zobrazený kód pre vyvolanie požiadavky na stiahnutie.

```
1 func application(_ application: UIApplication,
2                  performFetchWithCompletionHandler completionHandler:
3                  @escaping (UIBackgroundFetchResult) -> Void) {
4
5     let completionHandler = SyncPullRequest.wrapFetch(with:
          completionHandler)
6     // do something else
7     completionHandler(.noData)
8 }
```

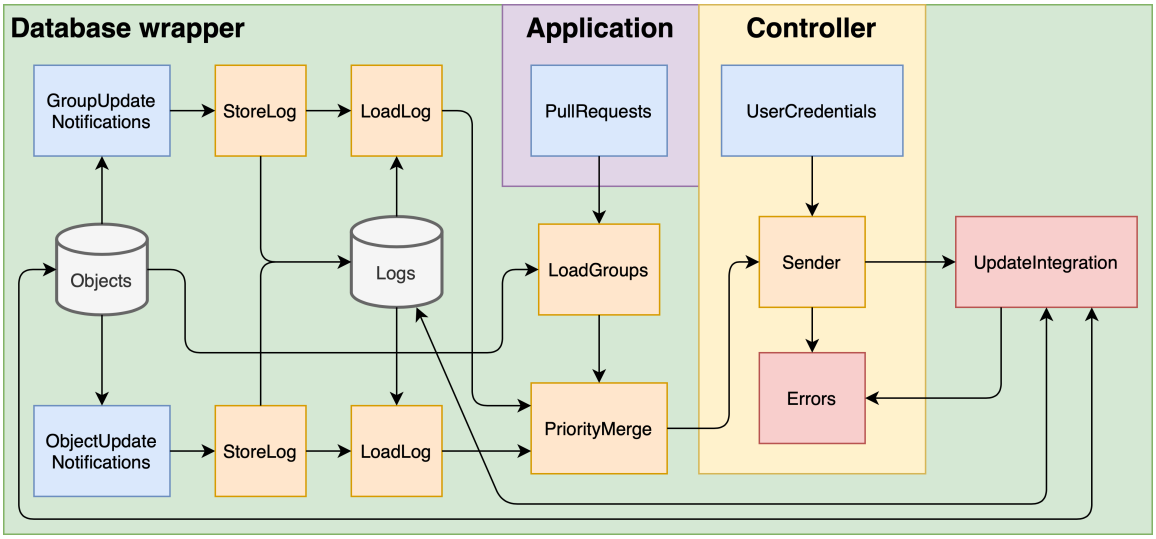
Výpis 4.5: Obsluha aktualizácie dát z pozadia vo funkcii triedy `UIApplicationDelegate`. Pokiaľ sa na serveri nachádzajú novšie zmeny, tak sa automaticky stiahnu.

4.3.2 Dátový tok v synchronizačnom procese

Synchronizačný proces spája štyri dátové toky do jedného, ako je zobrazené na obrázku 4.3. Dva dátové toky vznikajú na základe Realm notifikácií, a to pri zmene synchronizovateľných objektov alebo ich skupín (GroupUpdate Notifications a ObjectUpdate Notifications). Tieto toky generujú logy, ktoré sú ukladané do databázy (StoreLogs). Tretí dátový tok obsahuje požiadavky na stiahnutie nových zmien zo serverovej databázy (PullRequests). Spolu s logmi z predchádzajúcich tokov sa tento tok spája do jedného (PriorityMerge). Pokiaľ vzniknú dve správy naraz, sú vyberané na základe priority:

- najväčšia priorita – požiadavky z toku PullRequests,
- stredná priorita – logy upravujúce skupiny,
- najmenšia priorita – logy upravujúce objekty.

Keďže požiadavky pre stiahnutie nových dát z databázy sú často inicializované z notifikácií, musia byť čím skôr obslúžené, aby celková doba obsluhy notifikácie bola čo najmenšia. Aby bolo možné spravovať skupiny spolu s objektami bez vzniku konfliktov (napríklad by zmena objektu v novovytvorenej skupine bola odoslaná skôr ako vytvorenie tejto skupiny), sú všetky úpravy skupín spracovávané s vyššou prioritou ako objektov.



Obr. 4.3: Dátový tok v klientskom synchronizačnom procese. Význam farieb: zelená – určuje komponenty pracujúce s databázou, fialová – komponenty priamo využívané aplikáciou, žltá – komponenty v správe kontroléra, modrá – publisher, oranžová – operátor, červená – subscriber, šedá – databáza. Význam jednotlivých komponentov je popísaný v sekcii 4.3.2

Štvrtý tok obsahuje údaje potrebné pre autentifikáciu používateľa na serveri (UserCredentials). Pomocou neho je možné jednoducho pozastaviť odosielanie správ na server so zachovaním sledovania všetkých zmien. Ďalšia výhoda je, že v prípade, kedy by sa používateľ odhlásil z aplikácie, je možné sledovať všetky zmeny v objektoch bez obmedzení a po prihlásení sa všetky zmeny zosynchronizujú.

Komunikáciu so serverom má na starosti komponent Sender, ktorý prijímané dáta z toku vhodne upraví a odošle na server. Odpoveď zo servera je rovnako vhodne upravená a na základe typu odpovede ďalej distribuovaná. Každá chyba, ktorá vznikne pri komunikácii so serverom je distribuovaná do komponentu Error a prebehne zotavenie. Pokiaľ odpoveď obsahuje zmeny v databáze tak sú odoslané komponentu UpdateIntegrator, ktorý ich zaintegruje do lokálnej databázy a prípadne upraví logy.

Vytvorenie dátového grafu

Celý graf je možné vytvoriť v dvoch krokoch. V prvom kroku sa vytvoria všetky komponenty pracujúce s databázou. V druhom kroku sa vytvoria komponenty kontroléra a následne sa zostrojí celý graf. Vo výpise 4.6 sú tieto kroky znázornené na riadkoch jeden a štyri. Od momentu, kedy je zostrojený dátový graf, je celý synchronizačný proces pripravený. Pokiaľ sú v databáze dáta, ktoré nie sú zosynchronizované, spustí sa synchronizačný proces, v inom prípade sa čaká na zmeny.

```
1 // Observe all ‘Person‘ and ‘Dog‘ objects
2 let db = MeerkatClientDBWrapper(objects: [Person.self, Dog.self])
3
4 // Server and user configuration
5 let networkConfig = // server IP address, port, etc.
6 let credentials =  // user’s credentials if any, nil otherwise
```

```
7
8  // Set up controller
9  let controller = SyncController(db: db,
10         clientConfig: networkConfig,
11         syncConfig: credentials)
```

Výpis 4.6: Vytvorenie dátového grafu synchronizačného procesu z obrázku 4.3.

4.3.3 Spravovanie skupiny

Ako bolo spomenuté v prechádzajúcej kapitole, trieda `SyncGroup` slúži aj pre správu používateľov v tejto skupine. Pridať nového používateľa do skupiny je možné pomocou funkcie `subscribeUser`, ktorá pridá nový prvok do poľa `toSubscribe`. Takáto zmena vyprodukuje Realm notifikáciu, ktorá je ďalej spracovaná. Pri spracovaní sa skontroluje toto pole a pre každý prvok sa vytvorí log popisujúci pridanie nového používateľa do skupiny. Po vytvorení logov sú z poľa odstránené všetky spracované prvky tak, aby nevznikla ďalšia notifikácia. Pole je kvôli internému spracovávaniu neverejné a vývojár k nemu nemá priamy prístup. Odstránenie používateľov funguje obdobným spôsobom za použitia funkcie `unsubscribeUser` a poľa `toUnsubscribe`.

Okrem iného je v každej skupine prítomný atribút `role` s hodnotou práv používateľa v skupine. Zmena práv sa vykonáva pomocou funkcie `changeUsersRole`, ktorá sa vnútorne správa rovnako ako funkcia pre pridanie používateľa do skupiny[4]. Táto funkcia aktuálne slúži len ako syntaktický cukor.

4.3.4 Napojenie na používateľské rozhranie

Používateľské rozhranie býva často priamo závislé na dátach v databáze. Pri ich zmene je potrebné zobraziť ich aktuálny stav a v prípade ich zmazania je potrebné správne zareagovať. Používateľské rozhranie, ktoré nevyužíva knižnicu SwiftUI, môže reagovať na zmeny v databáze pomocou odchytávania Realm notifikácií a na ich základe prezentovať aktuálny stav zobrazených objektov. Pre používanie spolu s knižnicou SwiftUI bolo potrebné implementovať triedy, ktoré umožnia jednoduché definovanie chovania aplikácie pri zmene sledovaných objektov. Zmeny v jednom objekte je možné sledovať pomocou triedy `ObservableSyncObject` a pre kolekciu objektov je určená trieda `ObservableSyncObjects`. Využitie týchto tried je zobrazené vo výpise 4.9. Tieto objekty je možné využívať aj bez knižnice SwiftUI v aplikáciách využívajúcich Storyboardy.

Zaobalenie objektov

Knižnica SwiftUI ponúka zobrazovanie kolekcie pohľadov (views) na základe inej kolekcie objektov, ktoré sú identifikovateľné pomocou jednoznačného identifikátora. Pri aktualizácii pohľadov pravidelne pristupuje k tomuto identifikátoru. Pokiaľ sa objekt zmaže z databázy, prístup k identifikátoru spôsobí pád aplikácie. Pre zabránenie vzniku tohoto problému je implementovaná trieda `SyncObjectWrapper`, ktorá si drží lokálne uložený identifikátor a všetky ostatné správy deleguje danému objektu. Implementácia triedy je zobrazená vo výpise 4.7.

[4] V navrhnutom synchronizačnom protokole sa pri pridaní používateľa do skupiny, v ktorej už je aktualizujú jeho práva.

```
@dynamicMemberLookup
public struct SyncObjectWrapper<T: SyncObject>: Identifiable {
    public let id: String
    public let wrappedObject: T

    public init(object: T) {
        id = object.id
        wrappedObject = object
    }

    public subscript<U>(dynamicMember
        keyPath: ReferenceWritableKeyPath<T, U>) -> U {

        get {
            wrappedObject[keyPath: keyPath]
        }
        nonmutating set {
            wrappedObject[keyPath: keyPath] = newValue
        }
    }

    public subscript<U>(dynamicMember keyPath: KeyPath<T, U>) -> U
            wrappedObject[keyPath: keyPath]
    }
}
```

Výpis 4.7: Zaobalenie objektu tak, aby bolo možné pristupovať k identifikátoru aj po odstránení objektu.

Sledovanie zmien v kolekcii objektov

Sledovanie zmien v kolekcii pomocou `ObservableSyncObjects` využíva Realm notifikácie. Trieda ponúka konštruktor pre základné typy kolekcií používaných v Realme, v ktorých sa pred začatím sledovania odfiltrujú zmazané objekty. Pri vzniku notifikácie sa aktualizuje lokálne uložené pole sledovaných objektov. Skrátená implementácia je zobrazená vo výpise 4.8. Základné kolekcie Realmu majú atribút `observableObjects`, ktorý pre nich automatický vytvorí inštanciu `ObservableSyncObjects`.

```
public final class ObservableSyncObjects<T: SyncObject>:
    ObservableObject {
    @Published
    public var elements: [SyncObjectWrapper<T>]

    private var token: NotificationToken?

    private let results: Results<T>

    public init(results rs: Results<T>) {
        results = rs.filter("isDeleted = false")
```

```
11          elements = results.wrappedArray
12
13          token = results.observe { [unowned self] e in
14                  guard case .update = e else { return }
15                  self.elements = results.wrappedArray
16          }
17      }
18
19      deinit {
20          token?.invalidate()
21      }
22  }
```

Výpis 4.8: Trieda pre sledovanie zmien v kolekcii objektov (skrátená implementácia).

Sledovanie zmien jedného objektu

Sledovanie zmien nad jedným objektom je možné pomocou inštancie `ObservableSyncObject`, ktorá je implementovaná obdobne ako `ObservableSyncObjects`. Táto trieda ponúka verejný atribút, v ktorom je sledovaný objekt uložený. Pokiaľ sa objekt zmaže z databázy tak sa atribút automaticky nastaví na hodnotu `nil`. V tejto inštancii je možné definovať closure, ktorá bude zavolaná v prípade, ak sa objekt zmaže. Táto closure slúži k tomu, aby aplikácia mohla vhodne zareagovať (napríklad zobrazením iného pohľadu, v ktorom sa zmazaný objekt už nenachádza). Táto trieda ponúka aj možnosť definovať, ktoré atribúty sa majú sledovať alebo opačne, ktoré sa majú ignorovať. Každý synchronizovateľný objekt má funkciu `observableObject(...)`, pomocou ktorej sa automaticky vytvorí inštancia `ObservableSyncObject` s prípadnými sledovanými alebo ignorovanými atribútmi.

```
1  struct ContentView: View {
2      @Environment(\.presentationMode)
3      private var presentationMode: Binding<PresentationMode>
4
5      @ObservedObject
6      private var dogs: ObservableSyncObjects<Dog>
7
8      @ObservedObject
9      private var person: ObservableSyncObject<Person>
10
11     init(person: Person) {
12         self.person = person.observableObject(ignore: \.age)
13         dogs = person.dogs.observableObjects.onDelete(close)
14     }
15
16     var body: some View {
17         VStack {
18             Text("\(person.element!.name)'s dogs:")
19             List(dogs.elements, id: \.id) { dog in
20                 DogView(dog)
21             }
```

```
22          }
23      }
24
25      private func close() { presentationMode.wrappedValue.dismiss() }
26  }
```

Výpis 4.9: Prepojenie používateľského rozhrania s databázou spolu so sledovaním zmien. Výpis obsahuje pohľad, ktorý by mal byť prezentovaný ako modálne okno. Pokiaľ sa zmení nejaký zo sledovaných objektov, tak sa používateľské rozhranie automaticky aktualizuje. Pokiaľ sledovaný objekt v atribúte `person` bude zmazaný tak sa modálne okno automaticky zavrie.

4.4 Kódovanie správ synchronizačného protokolu

Nakoľko je navrhnutý protokol bezstavový, jednotlivé správy sú kódované do správ aplikačného rozhrania REST s využitím HTTPS pre zaistenie bezpečnosti. REST API umožňuje jednoduchú integráciu a rozšíriteľnosť. V nasledujúcom texte je uvedený spôsob kódovania základných správ navrhnutého protokolu do správ rozhrania REST.

4.4.1 Kódovanie pod-transakcie, transakcie a správy DIFF

Pod-transakciu, transakciu a aj odpovede serveru je možné kódovať vo formáte JSON. Kódovanie pod-transakcie je zobrazené vo výpise 4.10 a kódovanie transakcie vo výpise 4.11. Správa DIFF je zobrazená vo výpise 4.12.

```
1  {
2      "class": className,
3      "object": objectId,
4      "timestamp": timestamp,
5      "updates": [
6          {
7              "attribute1": newValue1
8          },
9          {
10             "attribute2": newValue2
11         },
12             // ...
13         {
14             "attributeN": newValueN
15         }
16     ]
17 }
```

Výpis 4.10: Kódovanie pod-transakcie vo formáte JSON. Význam atribútov je nasledovný: `className` – názov triedy, `objectId` – identifikátor objektu, `timestamp` – časová známka, `updates` – zmenené atribúty, `attributeN` – názov atribútu, `newValueN` – nová hodnota.

```
1  {
2      "deletions": [
3          // deletion1, deletion2, ..., deletionN
```

```
    ],
    "creations": [
        {
            "group": groupId,
            "object": subTransaction
        },
            // creation1, creation2, ..., creationN
    ],
    "updates": [
        // update1, update2, ..., updateN
    ],
}
```

Výpis 4.11: Kódovanie transakcie vo formáte JSON. Atribúty `deletions`, `updates` sú polia obsahujúce pod-transakcie a `creations` obsahujú pod-transakcie s identifikátorom skupiny, do ktorej nové objekty patria.

```
{
    "groupsDiff": [
        {
            "group": groupId,
            "version": groupDBVersion,
            "role": role
        },
            // ...
    ],
    "transactions": [
        // transaction1, transaction2, ..., transactionN
    ]
}
```

Výpis 4.12: Kódovanie správy DIFF vo formáte JSON. Atribúty: `groupsDiff` – pole obsahujúce popis zmien pre každú skupinu, `group` – identifikátor skupiny, `groupDBVersion` – nová verzia databázoveho pohľadu skupiny, `newObjects` – pole popisujúce pridané nové objekty s informáciou o názve triedy `class` a identifikátorom `object`, `role` – určuje aké právomoci v danej skupine, `transactions` – pole jednotlivých transakcií.

4.4.2 Kódovanie správy PUSH

Správa používa HTTP metódu `PATCH`. Zvyšok správy je kódovaný vo formáte JSON zobrazenom vo výpise 4.13.

```
{
    "scheme": schemeVersion,
    "groups": [
        {
            "id": groupId,
            "version": groupVersion
        },
            // ...
```

```
    ],
    "transactions": [
        // transaction1, transaction2, ..., transaction3
    ]
}
```

Výpis 4.13: Kódovanie správy PUSH vo formáte JSON. Atribúty: `schemeVersion` – verzia schémy databázy, `groupId` – identifikátor skupiny, `version` – verzia databázového pohľadu, `transactions` – jednotlivé transakcie.

4.4.3 Kódovanie správy PULL

Správa používa HTTP metódu `GET` s cestou rovnakou ako v správe PUSH. Telo správy je zobrazené vo výpise 4.14.

```
{
    "scheme": schemeVersion,
    "groups": [
        {
            "id": groupId,
            "version": groupVersion
        },
            // ...
    ]
}
```

Výpis 4.14: Kódovanie správy PULL vo formáte JSON. Atribúty: `schemeVersion` – verzia schémy databázy, `groupId` – identifikátor skupiny, `version` – verzia databázového pohľadu

4.4.4 Autorizácia

Pre autorizáciu sa používa HTTP hlavička `Authorization` s hodnotou `Bearer <token>`, kde `<token>` je autorizačný token pridelený serverom.

Kapitola 5

Testovanie

Výrazná časť knižníc bola vyvíjaná pomocou princípu test driven development (TDD), v ktorom sa najprv na základe návrhu zadefinuje programové rozhranie, napíšu sa pre neho testy a až potom sa implementujú funkcie rozhrania tak, aby všetky testy prešli.

Andrea Koutifaris vo svojom článku [25] popisuje rozdelenie vývoja pri TDD do troch fáz. Prvá „červená“ fáza sa zameriava na zadefinovanie funkcií, ktoré sa budú používať. Namiesto tela funkcie sa implementujú testy, ktoré budú overovať funkčnosť funkcie. Názov fázy je odvodený od toho, že pri spustení testov by nemal prejsť ani jeden test. Druhá fáza, nazývaná „zelenou“, už implementuje telá funkcií tak, aby všetky testy prešli. Veľký dôraz sa kladie na to, aby sa zbytočne neimplementovala žiadna funkcionalita, ktorá nie je potrebná. V poslednej fáze sa upravuje (refaktoruje) implementovaný kód. Príkladom môže byť odstránenie duplicitného kódu.

Aby bolo možné testovať implementované knižnice aj skutočnými vývojármi, bol vytvorený tutoriál pre vytvorenie aplikácie využívajúcej synchronizáciu.

5.1 Jednotkové testy

Jednotkové (unit) testy sa využívajú pre testovanie jednotlivých funkcií. Narozdiel od ručného testovania je možné ich automatizovať a spúšťať pravidelne. Veľká výhoda unit testov sa ukáže, pokiaľ pridaním nejakej funkcionality prestane iná funkcionalita fungovať. Pri ručnom testovaní by sa ľahko mohlo stať, že by takáto chyba ostala neodhalená, ale pri pravidelných automatizovaných testoch sa odhalí.

Implementované knižnice sú uložené vo webovej službe verzovacieho systému gitlab. Táto služba ponúka možnosť spúšťať automatizované testy pri každom nahratí zmien do repozitára. Pomocou tejto služby sa spúšťajú automatizované unit testy, a tak je možné pravidelne sledovať stav testov. Pre tento účel bolo potrebné zadefinovať konfiguračné parametre do súboru *.gitlab-ci.yml*. Ukážka tohoto súboru z knižnice MeerkatMySQL je zobrazená vo výpise 5.1.

```
1 image: "swift:5.1"
2
3 services:
4 - mysql
5
6 variables:
7   MYSQL_DATABASE: meerkat
```

```
  MYSQL_ROOT_PASSWORD: mysql
  C_MYSQL_USERNAME: root
  C_MYSQL_HOST: mysql
  C_MYSQL_PORT: 3306

before_script:
  - apt-get update
  - apt-get install libssl-dev
  - swift package resolve

build:
  script:
    - swift build
    - swift test
```

Výpis 5.1: Obsah konfiguračného súboru *.gitlab-ci.yml* spúšťajúci testy pre knižnicu MeerkatMySQL.

Testované boli hlavne knižnice obsahujúce kritickú funkcionalitu synchronizačného procesu. V tabuľke 5.1 je zobrazené zhrnutie unit testov v niektorých knižniciach.

Knižnica	Počet testov	Počet assertov	Pokrytie kódu [%]
MeerkatSchemeDescriptor	22	216	93.0
MeerkatClientCore	12	47	91.4
MeerkatMySQL	57	379	90.4
MeerkatServerSyncController	36	188	89.0
MeerkatRealmWrapper	11	123	65.0

Tabuľka 5.1: Zhrnutie testov v niektorých testovaných knižniciach.

5.2 Priebežné testovanie systému

Počas vývoja knižníc sa priebežne testovala ich funkcionalita vrámci celého systému. Týmto spôsobom bolo možné lepšie identifikovať nedostatky rozhrania a správne ich upraviť tak aby bolo použitie rozhrania čo najjednoduchšie.

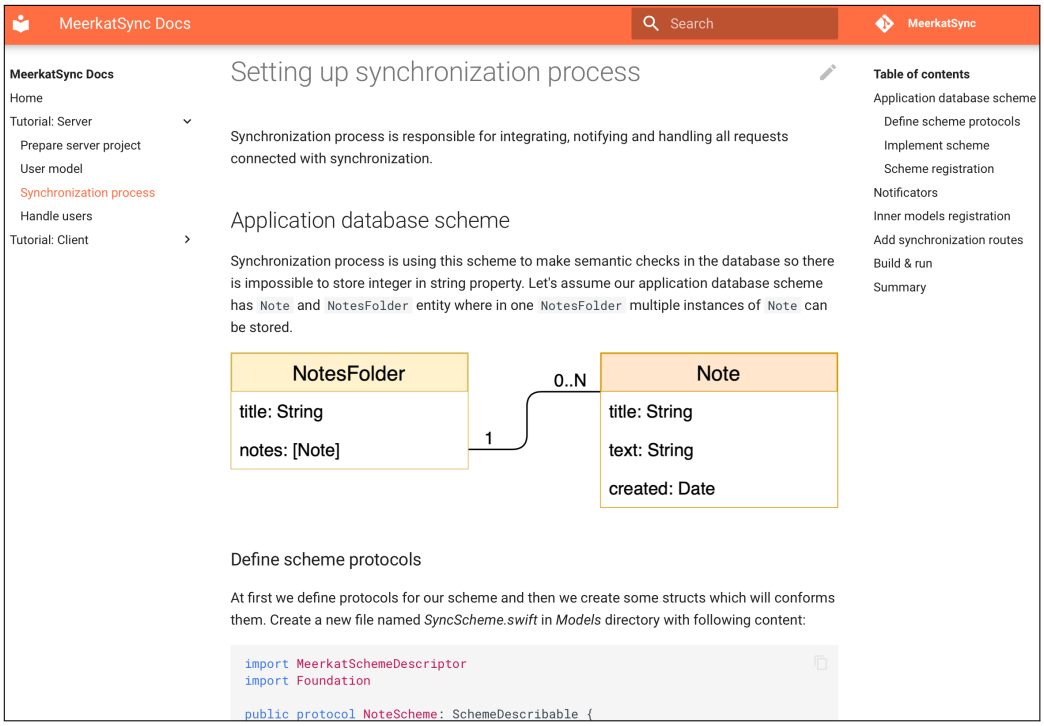
Pre tento účel bolo vytvorených niekoľko rôznych aplikácií, ktoré boli použité pre ručné testovanie celého systému. Pri testovaní boli použité viaceré zariadenia patriace rôznym používateľom.

5.3 Testovanie používateľmi

Aby sa overila kvalita výslednej aplikácie skutočnými vývojármi, bol vytvorený tutoriál skladajúci sa z dvoch častí pre vytvorenie aplikácie využívajúcej knižnicu MeerkatSync. Tutoriál je verejne dostupný na webovej stránke `meerkatsync.gitlab.io/docs`.

V prvej časti tutoriálu sa vytvára server, ktorý si drží master-kópiu aplikačnej databázy a obsluhuje klientov. Druhá časť sa zameriava na klientsku aplikáciu určenú pre zariadenia s operačným systém iOS s minimálnou verziou 13[1].

[1] Aplikáciu je možné spustiť aj na operačnom systéme iPadOS a MacOS.



Obr. 5.1: Webová stránka s tutoriálom pre serverovú aplikáciu.

Pre vytvorenie webových stránok sa využíval program *mkdocs*[2] a šablóna bola prevzatá z *Material for MkDocs*[3].

5.3.1 Tutoriál – tvorba serveru

V tutoriále je ukázaný spôsob, ako jednoducho vytvoriť serverovú aplikáciu, ktorá okrem synchronizácie riadi aj správu používateľov. Celý vývoj aplikácie je rozdelený do štyroch kapitol:

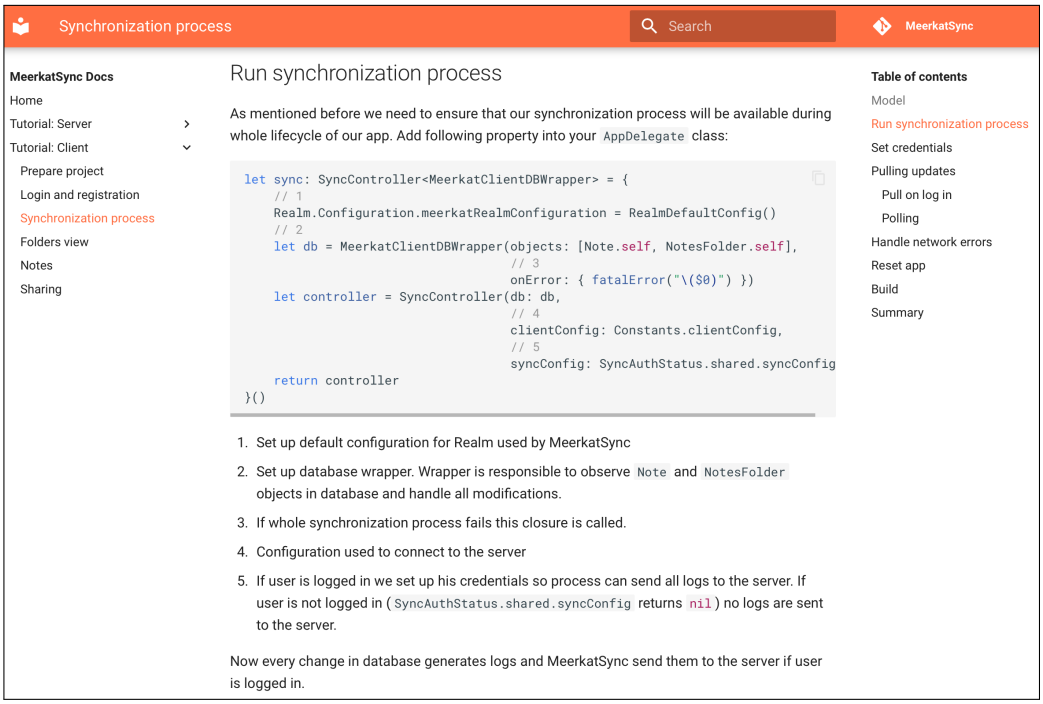
- príprava prostredia a stiahnutie šablóny,
- definovanie modelu používateľa,
- spustenie synchronizačného procesu (obrázok 5.1),
- správa používateľov.

5.3.2 Tutoriál – klientska aplikácia

Vývojár sa v tomto tutoriále naučí, ako napojiť aplikáciu na synchronizačný proces, ako bezpečne pristupovať ku objektom, ktoré môžu byť zmazané vrámci synchronizácie a ako riešiť situáciu, kedy sa prezentovaný objekt pozmení alebo zmaže. Tutoriál je rozdelený do šiestich kapitol:

- stiahnutie šablóny a jej príprava,

[2] https://www.mkdocs.org
[3] https://squidfunk.github.io/mkdocs-material



Obr. 5.2: Webová stránka s tutoriálom pre klientskú aplikáciu.

- registrácia a prihlásenie používateľa,
- spustenie synchronizačného procesu (obrázok 5.2),
- sledovanie zmien v kolekcii,
- sledovanie zmien v jednom objekte a možné riešenia zmien spôsobených synchronizáciou zobrazovaného objektu,
- zdieľanie objektov medzi viacerými používateľmi.

5.3.3 Výsledky používateľského testovania

Do testovania sa zapojili celkovo traja vývojári s rôznymi skúsenosťami. Po dokončení tutoriálu odpovedali na päť otázok:

1. Ako hodnotíte náročnosť pochopenia tutoriálu?
2. Ako hodnotíte náročnosť integrácie synchronizačného procesu do aplikácie?
3. Ako hodnotíte intuitívnosť rozhrania MeerkatSync?
4. Odporučili by ste MeerkatSync svojim kolegom?
5. Do akej miery splnil MeerkatSync vaše očakávania?

Úroveň vývojára	Otázka 1	Otázka 2	Otázka 3	Otázka 4	Otázka 5
Začiatočník	3	5	3	4	5
Stredne pokročilý	4	4	4	5	5
Pokročilý	4	4	5	5	4

Tabuľka 5.2: Odpovede na jednotlivé otázky po dokončení tutoriálu.

Na každú z týchto otázok bolo možné odpovedať číselne z intervalu 0-5, kde 5 bola najpriaznivejšia odpoveď a 0 najmenej priaznivá. Odpovede na jednotlivé otázky sú zobrazené v tabuľke 5.2. Z tabuľky je vidno, že oslovení vývojári hodnotili knižnicu MeerkatSync pozitívne. Z bodového ohodnotenia vyplýva, že tutoriál nie je vhodný pre začiatočníkov.

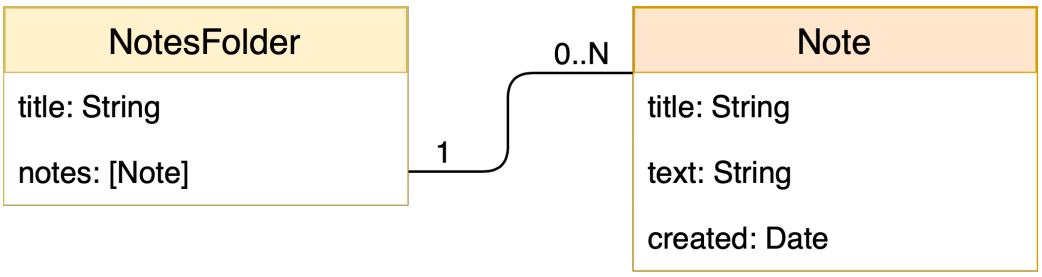
Okrem bodového hodnotenia bola vykonaná aj konzultácia o celkových dojmoch a radách, čo by sa mohlo zlepšiť, prípadne zmeniť. Celkovo bolo veľmi pozitívne hodnotené, že pre integráciu synchronizácie sú potrebné minimálne zmeny v klientskej aplikácii. Jednou z hlavných nevýhod na ktorých sa zhodli bolo, že serverová aplikácia je v jazyku Swift, vďaka čomu ju nie je možné spúšťať na väčšine bežných hostingoch[4]. Túto nevýhodu označili aj ako kompenzovateľnú, nakoľko je možné pri tvorbe serverovej aj klientskej aplikácie využívať iba jeden jazyk, a tým prispieť k znovupoužiteľnosti kódu potrebného v oboch aplikáciách. Jedno z hlavných doporučení na zlepšenie bolo pridať ďalší spôsob pre distribúciu zmien tak, aby sa zaručilo, že každá zmena bude v reálnom čase doručená na všetky aplikácie (bežiace v popredí) nakoľko bežné notifikácie túto funkcionalitu negarantujú.

[4]Väčšina webových hostingov ponúka podporu iba pre jazyk PHP.

Kapitola 6

Prípadová štúdia – aplikácia pre správu a zdieľanie poznámok

Pre demonštráciu používania vytvorených knižníc bola zvolená aplikácia pre správu poznámok. Jednotlivé poznámky sú zoskupené v priečinkoch, ktoré je možné zdieľať medzi viacerými používateľmi. Dátová schéma tejto aplikácie je zobrazená na obrázku 6.1.



Obr. 6.1: Dátová schéma aplikácie.

Táto kapitola sa zaoberá popisom dôležitých častí kódu z klientskej aplikácie, ktoré sú naviazané na synchronizačný proces. Aplikácia je založená na knižnici SwiftUI, aby sa ukázala jednoduchosť napojenia synchronizácie s novým princípom programovania aplikácií pre operačné systémy od firmy Apple.

Popisovaná aplikácia sa odvíja od aplikácie vytvorenej pomocou tutoriálu spomenutého v sekcii 5.3.

6.1 Zaistenie synchronizácie

Pre spustenie synchronizačného procesu je potrebné vhodne upraviť aktuálny model aplikácie a počas celého behu aplikácie držať referenciu na synchronizačný proces.

6.1.1 Model aplikácie v systéme Realm

Model aplikácie sa skladá z dvoch tried, `Note` a `NotesFolder`, dediacich od triedy `Object`. Na to, aby bolo možné objekty týchto tried synchronizovať, je potrebné pridať atribúty `id`, `isDeleted`, `group` a prepísať dedenie z `Object` na `SyncObject`. Vo výpise 6.1 je zobrazená trieda reprezentujúca priečinok s poznámkami a vo výpise 6.2 je prerobená tak, aby ju bolo možné synchronizovať.

```
1  import RealmSwift
2
3  final class NotesFolder: Object {
4      typealias Notes = List<Note>
5
6      @objc dynamic var title = ""
7
8      let notes = Notes()
9  }
```

Výpis 6.1: Trieda reprezentujúca priečinok poznámok uložený v databáze.

```
1  import MeerkatRealmWrapper
2
3  final class NotesFolder: SyncObject {
4      typealias Notes = List<Note>
5
6      @objc dynamic var id = UUID().uuidString
7
8      @objc dynamic var title = ""
9
10     let notes = Notes()
11
12     @objc dynamic var isDeleted = false
13
14     @objc dynamic var group: SyncGroup? = nil
15 }
```

Výpis 6.2: Synchronizovateľná trieda pre priečinok poznámok.

6.1.2 Spustenie synchronizačného procesu

Aby bolo možné zachytiť všetky zmeny v databáze, je potrebné. aby synchronizačný proces fungoval po celý životný cyklus aplikácie. Toto je možné zaistiť pokiaľ AppDelegate bude držať referenciu na kontrolér synchronizačného procesu. Vo výpise 6.3 je zobrazený kód, v ktorom sa vytvorí celý synchronizačný proces a uloží sa do premennej `sync`, ktorá by mala byť umiestnená ako atribút v AppDelegate.

```
1  let sync: SyncController<MeerkatClientDBWrapper> = {
2      let db = MeerkatClientDBWrapper(objects: [Note.self,
3                                                NotesFolder.self],
4                                      onError: { fatalError("\($0)") })
5      let controller = SyncController(db: db,
6                                      clientConfig: clientConfig,
7                                      syncConfig: syncConfig)
8      return controller
9  }()
```

Výpis 6.3: Zostavenie synchronizačného procesu a jeho uloženie do premennej `sync`. Kód predpokladá existenciu premennej `syncConfig` so synchronizačným a autorizačným tokenom a premennej `clientConfig` s IP adresou a portom serveru. V prípade, ak sa nepodarí zostaviť synchronizačný proces, spôsobí sa pád aplikácie (riadok 4). Takáto situácia môže nastať, len ak sa nepodarí pristúpiť k databáze.

6.1.3 Sledovanie zmien na serveri pomocou pollingu

Keďže notifikácie nie sú spoľahlivé, je potrebné využívať aj iný spôsob ako priebežne sledovať, či sú na serveri nové zmeny, ktoré je potrebné synchronizovať. Najľahšie implementovateľné sledovanie týchto zmien je pomocou pollingu s využitím objektu typu `Timer`. Pre tento účel je možné pridať do AppDelegate ďalší atribút (`pullTimer`), ktorý bude držať referenciu na časovač zodpovedný za periodické sledovanie stavu na serveri. Ukážkový kód je zobrazený vo výpise 6.4.

```
1 pullTimer = Timer.scheduledTimer(withTimeInterval: 120, repeats: true) {
      [weak self] timer in
2     guard self != nil else {
3         timer.invalidate()
4         return
5     }
6     SyncPullRequest.send()
7 }
```

Výpis 6.4: Časovač zodpovedný za periodické stiahnutie prípadných zmien zo servera každých 120 sekúnd.

6.2 Správa priečinkov s poznámkami

Poznámky sú uložené v jednotlivých priečinkoch. Priečinky slúžia pre organizáciu poznámok a je možné ich zdieľať medzi viacerými používateľmi.

6.2.1 Pridanie priečinku

Pridanie nového priečinku je možné vykonávať rovnako ako pred použitím knižnice MeerkatSync. Jediným rozdielom je potreba vytvorenia novej skupiny, do ktorej bude priečinok patriť. Funkcia pre uloženie nového priečinku je zobrazená vo výpise 6.5.

```
1  func saveFolder(withTitle title: String) {
2      let realm = try! Realm()
3      try! realm.write {
4          // create a new group
5          let newGroup = SyncGroup()
6          realm.add(newGroup)
7
8          // create a new folder
9          let folder = NotesFolder(title: title)
10         folder.group = newGroup
11         realm.add(folder)
12     }
13 }
```

Výpis 6.5: Uloženie nového priečinku.

6.2.2 Pridanie poznámky

Nakoľko sa poznámka viaže na konkrétny priečinok, je potrebné vykonať pred samotným uložením poznámky kontrolu, či je priečinok stále valídny a nebol zmazaný vrámci synchronizácie. Ukážka uloženia novej poznámky je vo výpise 6.6. Oproti kódu bez synchronizácie sa pridala kontrola priečinku na riadku 6 a nastavenie skupiny, do ktorej poznámka patrí na riadku 11.

```
1 func saveNote(title: String,
2               text: String,
3               folder: ObservableSyncObject<NotesFolder>) {
```

```
4
5       // check if folder is still valid
6       folder.ifValid { folder in
7           let realm = try! Realm()
8           try! realm.write {
9               let note = Note(title: title, text: text)
10              // Note's group must be same as its folder
11              note.group = folder.group
12
13              realm.add(note)
14              folder.notes.append(note)
15          }
16      }
17  }
```

Výpis 6.6: Uloženie novej poznámky.

6.2.3 Mazanie priečinku

Pri mazaní priečinku je potrebné kontrolovať, či je ešte valídny. Namiesto bežného zmazania je potrebné využiť soft-delete pomocou atribútu `isDeleted`, ako je zobrazené vo výpise 6.8. Výpis 6.7 zobrazuje mazanie priečinku, ak by nebola použitá synchronizácia.

```
1  func delete(_ folder: NotesFolder) {
2    let realm = try! Realm()
3    try! realm.write {
4        // delete all notes
5        realm.delete(folder.notes)
6        // delete folder
7        realm.delete(folder)
8    }
9  }
```

Výpis 6.7: Zmazanie priečinku bez synchronizácie.

```
1  func delete(_ folder:
2      ObservableSyncObject<NotesFolder>) {
3    folder.ifValid { folder in
4      let realm = try! Realm()
5      try! realm.write {
6        // delete all notes
7        folder.notes.forEach {
8          $0.isDeleted = true
9        }
10       // delete folder
11       folder.isDeleted = true
12       // delete folder's group
13       folder.group?.isDeleted = true
14     }
15   }
16 }
```

Výpis 6.8: Zmazanie priečinku so synchronizáciou.

6.2.4 Sledovanie zmien v kolekcii

Každý objekt dediaci od `SyncObject` má atribút `observableObjects`, pomocou ktorého je možné získať sledovateľný objekt s kolekciou všetkých objektov daného typu uložených v databáze. Kolekcia je automaticky aktualizovaná pri každej zmene v databáze. Ukážka je na riadkoch 8-9 vo výpise 6.9.

6.2.5 Sledovanie zmien jedného objektu

Každý synchronizovateľný objekt má funkciu `observableObject`, ktorá vytvorí inštanciu `ObservableSyncObject`. Táto inštancia si drží referenciu na objekt, z ktorého vznikla a je možné ju využiť pre sledovanie zmien v danom objekte.

Reakcia na zmazanie priečinku

`ObservableSyncObject` ponúka možnosť určiť closure, ktorá sa má vykonať v prípade, kedy sa sledovaný objekt odstráni z databázy pomocou funkcie `onDelete(_:)`. Vo výpise 6.9 je pomocou funkcie `detailView(_:)` definovanej na riadku 30 vytvorená inštancia `ObservableSyncObject` sledujúca konkrétny priečinok a pohľad zobrazujúci detail tohoto priečinku. V prípade ak sa priečinok odstráni z databázy, tak sa zavolá funkcia `returnToRoot()`, ktorá spôsobí, že sa prezentovaný detail zavrie, a tým sa zabráni interakcii používateľa so zmazaným objektom.

Pre reakciu na zmazanie sledovaného objektu sa môže využiť aj atribút `cache`, ktorý ponúka bezpečný prístup k atribútom objektu aj po jeho zmazaní.

```
1  import SwiftUI
2  import MeerkatRealmWrapper
3
4  struct FoldersView: View {
5      @State
6      private var activeFolder: String? = nil
7
8      @ObservedObject
9      private var folders = NotesFolder.observableObjects
10
11     var body: some View {
12         NavigationView {
13             List {
14                 ForEach(folders.elements) { folder in
15                     NavigationLink(destination: detailView(for: folder),
16                                    tag: folder.id,
17                                    selection: self.$activeFolder) {
18                         HStack {
19                             Text(folder.title)
20                             Spacer()
21                             Text("(\(folder.notes.count))")
22                                 .foregroundColor(.secondary)
23                         }
24                     }
25                 }
26             }
27         }
28     }
29
30     func detailView(for folder: SyncObjectWrapper<NotesFolder>) -> some
           View {
```

```
31          let observableFolder = folder.observableObject().onDelete(
                returnToRoot)
32          return FolderDetailView(folder: observableFolder)
33      }
34
35      func returnToRoot() {
36          activeFolder = nil
37      }
38  }
```

Výpis 6.9: Pohľad zobrazujúci všetky uložené priečinky. Pre každý priečinok je zobrazený jeho názov a počet poznámok v ňom.

6.3 Zdieľanie priečinku

Keďže každý priečinok je vo vlastnej skupine, tak je jednoduché zdielať tieto priečinky medzi viacerými používateľmi. V každej skupine môžu figurovať používatelia v troch úrovniach právomocí. Používateľ môže mať práva iba na čítanie, na čítanie a zapisovanie alebo na čítanie, zapisovanie a správu používateľov v skupine. Používateľ s posledným typom práv je v nasledujúcom texte označovaný ako vlastník skupiny.

6.3.1 Práva v skupine

Práva používateľa v skupine určujú, aké rozhranie a akú funkcionalitu môže používateľ v danej skupine vykonávať. Keďže iba vlastník skupiny môže pridávať používateľov, je vo výpise 6.10 atribút `isOwner`, na základe ktorého sa zobrazí formulár pre pridanie používateľa. Atribút využíva funkciu `transform` pre bezpečný prístup ku skupine. Pokiaľ je skupina valídna, tak je hodnota parametra v closure referencia na skupinu, inak má parameter hodnotu `nil`.

```
1  struct AddUserView: View {
2      @ObservedObject
3      var group: ObservableSyncObject<SyncGroup>
4
5      @State
6      private var username = ""
7
8      private var isOwner: Bool {
9          group.transform { $0?.role == .owner }
10     }
11
12     var body: some View {
13         if isOwner {
14             VStack {
15                 Text("Add a user:")
16                 TextField("Username", text: $username)
17                 Button(action: add) {
18                     Text("Add")
19                 }
20             }
```

```
21          } else {
22              Text("Only owner can add users")
23          }
24      }
25  }
```

Výpis 6.10: Pohľad zobrazujúci formulár pre pridanie nového používateľa do skupiny.

6.3.2 Správa používateľov v skupine

Spravovať používateľov je možné aj v režime off-line. Skupina ponúka funkcie pre manipuláciu s používateľmi. Každá požiadavka je uložená a pokiaľ je prístupné internetové pripojenie, tak sa odosiela na server, kde bude vybavená. Výpis 6.11 zobrazuje pridanie, zmazanie a upravenie práv používateľov v režime off-line. Volanie týchto funkcií je možné len vrámci databázovej transakcie.

```
1  // check group
2  group.ifValid { group in
3      let realm = try! Realm()
4      try! realm.write {
5          // add ‘user1‘ to ‘group‘
6          group.subscribeUser(withId: user1, as: .readAndWrite, on: realm)
7          // remove ‘user2‘ from ‘group‘
8          group.unsubscribeUser(withId: user2, on: realm)
9          // update ‘user3‘’s role in ‘group‘ to ‘.owner‘
10         group.changeUsersRole(id: user3, newRole: .owner, on: realm)
11     }
12 }
```

Výpis 6.11: Správa používateľov v skupine. Používateľ s identifikátorom `user1` je pridaný do skupiny s právami na čítanie a zápis. Používateľ s identifikátorom `user2` je odstránený zo skupiny a používateľovi s identifikátorom `user3` sa nastavia práva vlastníka.

Kapitola 7

Záver

V práci sa podarilo vytvoriť knižnicu MeerkatSync skladajúcu sa z viacerých balíkov pre jednoduchú integráciu synchronizácie dát do klientskych aplikácií s operačným systémom iOS, iPadOS a MacOS. Vytvorená knižnica umožňuje synchronizovať dáta medzi viacerými zariadeniami s podporou synchronizácie zdieľaných dát medzi viacerými používateľmi. Knižnica ďalej ponúka jednoduché napojenie synchronizačného procesu na používateľské rozhranie, správu používateľov v skupine a ponúka aj možnosť vyvolávať vývojárom definované akcie pri zmazaní objektu vrámci synchronizačného procesu. Okrem knižnice bol vytvorený aj tutoriál, ktorý bol použitý pri testovaní vývojármi. Dôležitým výsledkom tohoto testovania je, že vytvorená knižnica splnila očakávania opýtaných vývojárov, ktorí by odporučili knižnicu aj svojim kolegom.

Pokračovanie práce by sa mohlo venovať zlepšeniu distribúcie notifikácií pomocou websocketov, pridaniu podpory pre databázový systém CoreData, pridaniu podpory pre ďalšie serverové databázové systémy (PostgreSQL, MongoDB, ...) alebo aktualizovaniu serverových knižníc pre novú verziu frameworku Vapor. Ďalšie možné pokračovanie sa môže venovať vytvoreniu klientskej knižnice zaisťujúcej synchronizáciu pre aplikácie bežiace na operačnom systéme Android alebo pre webové aplikácie, prípadne vytvorením knižnice v jazyku PHP určenú pre obsluhu synchronizácie na centrálnom prvku, a tým umožniť beh serverovej aplikácie aj na bežných hostingoch.

Všetky časti zadania diplomovej práce boli naplnené a výsledná práca je pripravená na bežné používanie pri vývoji mobilných aplikácií vyžadujúcich synchronizáciu dát. Na základe veľmi dobrej spätnej väzby a vízie úspechu knižnice plánujem vytvoriť úplnú používateľskú dokumentáciu a publikovať knižnicu na rôznych fórach so zameraním na vývojárov aplikácií pre zariadenia ekosystému spoločnosti Apple. Následne plánujem začať s implementáciou niektorých bodov z možného pokračovania, a to konkrétne s rozšírením podpory knižnice aj pre databázový systém CoreData a zlepšením odosielania notifikácií pomocou websocketov.

Literatúra

[1] Apple. *Application(_:didReceiveRemoteNotification:fetchCompletionHandler:): Tells the app that a remote notification arrived that indicates there is data to be fetched.* [online]. [cit. 9. apríla 2020]. Dostupné z: https://developer.apple.com/documentation/uikit/uiapplicationdelegate/1623013-application.

[2] Apple. *Enumerations* [online]. [cit. 7. apríla 2020]. Dostupné z: https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html.

[3] Apple. *Managing Your App's Life Cycle: Respond to system notifications when your app is in the foreground or background, and handle other significant system-related events.* [online]. [cit. 8. apríla 2020]. Dostupné z: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle.

[4] Apple. Developer documentation. *NSManagedObjectModel* [online]. [cit. 17. novembra 2019]. Dostupné z: https://developer.apple.com/documentation/coredata/nsmanagedobjectmodel.

[5] Apple. *Pushing Background Updates to Your App: Deliver notifications that wake your app and update it in the background.* [online]. [cit. 8. apríla 2020]. Dostupné z: https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/pushing_background_updates_to_your_app.

[6] Apple. *Sending Notification Requests to APNs: Transmit your remote notification payload and device token information to APNs.* [online]. [cit. 9. apríla 2020]. Dostupné z: https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server/sending_notification_requests_to_apns.

[7] Apple. The powerful programming language that is easy to learn. *Swift* [online]. [cit. 7. apríla 2020]. Dostupné z: https://developer.apple.com/swift/.

[8] Apple. Developer documentation. *Persistent Store Types and Behaviors: Core Data Programming Guide* [online]. Marec 2017 [cit. 17. novembra 2019]. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/PersistentStoreFeatures.html.

[9] Apple. Developer documentation. *What Is Core Data?: Core Data Programming Guide* [online]. Marec 2017 [cit. 17. novembra 2019]. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/index.html.

[10] Bancroft, A. *NSPersistentCloudKitContainer Buggy Behavior List* [online]. 2019 [cit. 1. mája 2020]. Dostupné z: https://www.andrewcbancroft.com/blog/ios-development/data-persistence/nspersistentcloudkitcontainer-buggy-behavior-list/.

[11] Bulavin, V. *Asynchronous Programming with Futures and Promises in Swift with Combine Framework* [online]. Január 2020 [cit. 8. apríla 2020]. Dostupné z: https://www.vadimbulavin.com/asynchronous-programming-with-future-and-promise-in-swift-with-combine-framework/.

[12] Burk, N. *SQLite With Swift Tutorial: Getting Started: In this SQLite with swift tutorial, learn how to utilise the SQLite database with your Swift projects, including inserting, updating and deleting rows.* [online]. September 2017 [cit. 17. novembra 2019]. Dostupné z: https://www.raywenderlich.com/385-sqlite-with-swift-tutorial-getting-started.

[13] Dell Boomi. Boomi. *Sync Strategies Part 1: One-Way Syncs* [online]. Október 2018 [cit. 14. novembra 2019]. Dostupné z: https://community.boomi.com/s/article/syncstrategiespart1onewaysyncs.

[14] Dell Boomi. Boomi. *Sync Strategies Part 2: Two-Way Syncs* [online]. Október 2018 [cit. 14. novembra 2019]. Dostupné z: https://community.boomi.com/s/article/syncstrategiespart2twowaysyncs.

[15] Dell Boomi. Boomi. *Sync Strategies Part 3: Real-Time Syncs* [online]. Október 2018 [cit. 15. novembra 2019]. Dostupné z: https://community.boomi.com/s/article/syncstrategiespart3realtimesyncs.

[16] Ding, Z., Meng, X. a Wang, S. A transactional asynchronous replication scheme for mobile database systems. *Journal of Computer Science and Technology*. Jul 2002, roč. 17, č. 4, s. 389–396. Dostupné z: https://doi.org/10.1007/BF02943279. ISSN 1860-4749.

[17] Faiz, M. a Shanker, U., ed. Data Synchronization in Distributed Client-Server Applications. In: Faiz, M. a Shanker, U., ed. *2^{th} IEEE Internation Conference on Engineering and Technology (ICETECH), 17^{th} & 18^{th}*. Coimbatore, TN, India: [b.n.], Marec 2016. ISBN 978-1-4673-9916-6.

[18] Galante, D. *Higher Order Functions in Swift (Sorted, Map, Filter, Reduce)* [online]. Máj 2017 [cit. 7. apríla 2020]. Dostupné z: https://medium.com/@Dougly/higher-order-functions-in-swift-sorted-map-filter-reduce-dff60b5b6adf.

[19] Gio. *Swift Functors, Applicatives, and Monads in Pictures* [online]. Jún 2015 [cit. 7. apríla 2020]. Dostupné z: https://www.mokacoding.com/blog/functor-applicative-monads-in-pictures/.

[20] Haskell. *Functional programming* [online]. Február 2020 [cit. 7. apríla 2020]. Dostupné z: https://wiki.haskell.org/Functional_programming.

[21] Heck, J. *Using Combine* [online]. Marec 2017 [cit. 8. apríla 2020]. Dostupné z: https://heckj.github.io/swiftui-notes/.

[22] Hoffman, J. *Mastering Swift 4.* Packt Publishing Ltd, 2017. ISBN 978-1-78847-780-2.

[23] Hudson, P. Hacking with Swift. *Learn to make iOS apps with real projects: Projects 139* [online]. [cit. 17. novembra 2019]. Dostupné z: http://qeam.org/qeam/pdf/hackingwithswift.pdf.

[24] Jain, P. *Which Database Should You Choose for iOS Application Development?* [online]. Júl 2017 [cit. 17. novembra 2019]. Dostupné z: http://www.elitechsystems.com/which-database-you-should-choose-for-ios-application-development/.

[25] Koutifaris, A. *Test Driven Development: what it is, and what it is not.* [online]. Júl 2018 [cit. 18. apríla 2020]. Dostupné z: https://www.freecodecamp.org/news/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2/.

[26] Leach, P., Microsoft, Mealling, M., Refactored Networks, LLC, Salz, R. et al. *A Universally Unique IDentifier (UUID) URN Namespace* [Internet Requests for Comments]. RFC 4122. RFC Editor, July 2005. 1-32 s. Dostupné z: https://tools.ietf.org/html/rfc4122.

[27] Lee, Y., Kim, Y. a Choi, H. Conflict Resolution of Data Synchronization in Mobile Environment. In: Laganá, A., Gavrilova, M. L., Kumar, V., Mun, Y., Tan, C. J. K. et al., ed. *Computational Science and Its Applications – ICCSA 2004.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 196–205. Dostupné z: https://link.springer.com/chapter/10.1007/978-3-540-24709-8_21. ISBN 978-3-540-24709-8.

[28] Marinucci, A. codeburst. *Data Synchronization Primer: Step by step introduction to data synchronization strategies and solutions* [online]. Apríl 2018 [cit. 14. novembra 2019]. Dostupné z: https://codeburst.io/data-synchronization-primer-88ad04e1747b.

[29] Pál, M. a Láner, G. Mobile Data Synchronization Methods. *Hungarian Journal of Industry and Chemistry.* 2016, roč. 44, č. 2, s. 93–98. ISSN 2450-5102.

[30] Ramya, B. S., Koduri, S. B. a Seetha, M. A Stateful Database Synchronization Approach for Mobile Devices. *International Journal of Innovative Research in Computer and Communication Engineering.* 2012, roč. 2, č. 3, s. 316–320. ISSN 2231-2307.

[31] Realm. Docs. *Realm Swift 4.4.0* [online]. [cit. 5. apríla 2020]. Dostupné z: https://realm.io/docs/swift/latest/.

[32] Sonawane, N., Shaikh, S., Bhalerao, P., Kachroo, P. a Asmita Pete, S. P. Synchronization between Mobile Devices and Server: An Optimistic Approach. *International Journal of Innovative Research in Computer and Communication Engineering.* 2016, roč. 4, č. 4, s. 5017–5022. ISSN 2320-9798.

[33] SQLite. SQLite. *What Is SQLite?* [online]. [cit. 17. novembra 2019]. Dostupné z: https://www.sqlite.org/index.html.

[34] Tihomirovs, J. a Grabis, J. Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics. *Information Technology and Management Science*. December 2016, roč. 19, s. 92–97. ISSN 2255-9094.

[35] Vakhnenko, H. a Liptuga, M. *CoreData vs Realm: What to Choose as a Database for iOS Apps* [online]. [cit. 30. decembra 2019]. Dostupné z: https://agilie.com/en/blog/coredata-vs-realm-what-to-choose-as-a-database-for-ios-apps.

[36] Vekariya, A. *Core Data (CRUD) with Swift 4.2 for Beginners* [online]. Júl 2018 [cit. 17. novembra 2019]. Dostupné z: https://medium.com/@ankurvekariya/core-data-crud-with-swift-4-2-for-beginners-40efe4e7d1cc.

Príloha A

Obsah priloženého pamäťového média

Priložené pamäťové médium obsahuje nasledujúce adresáre a súbory:

- *docs/* - priečinok obsahujúci tutoriál,
- *libraries/* - priečinok obsahujúci vytvorené knižnice,
- *LICENSE* - súbor obsahujúci licenciu projektu,
- *MeerkatSync.pdf* - textová časť práce,
- *tex/* - priečinok obsahujúci zdrojové texty textovej časti práce,
- *README.md* - popis obsahu pamäťového média vo formáte markdown.