BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ORM LIBRARY FOR ANDROID

ORM KNIHOVNA PRO ANDROID

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PAVEL SALVA**
AUTOR PRÁCE

SUPERVISOR **Ing. ZBYNĚK KŘIVKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Zadání bakalářské práce/19934/2016/xsalva02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Salva Pavel**
Obor: Informační technologie
Téma: **ORM knihovna pro Android**
ORM Library for Android
Kategorie: Databáze

Pokyny:

1. Seznamte se s relačními databázemi pro platformu Android a proveďte rešerši existujících ORM knihoven pro Android.
2. Dle konzultace s vedoucím navrhněte vlastní ORM knihovnu včetně API. V návrhu se zaměřte na odstranění vybraných nevýhod či problémů existujících ORM knihoven.
3. Dle návrhu ORM knihovnu implementujte a podrobně dokumentujte.
4. Výslednou knihovnu důkladně otestujte s využitím jednotkových testů a demonstrační aplikace.

Literatura:

- Sunny Kumar Aditya, Vikash Kumar Karn: *Android SQLite Essentials*. Packt Publishing, 2014.
- Terry Halpin: *Object-Role Modeling Fundamentals: A Practical Guide to Data Modeling with ORM*. Technics Publications, 2015.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část bodu 3.

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

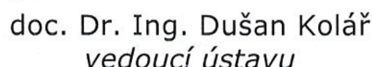
Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L.S.



doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstract

This bachelor thesis describes the topic of ORM library development for Android platform. It analyses the currently most used ORM libraries and based on the analysis it specifies new library. Furthermore, it describes its implementation in Java programming language with code generation. The library is tested throughout the development and a sample application is created for the final version.

Abstrakt

Tato bakalářská práce popisuje problematiku vývoje ORM knihovny pro platformu Android. Obsahuje analýzu dnes nejpoužívanějších ORM knihoven a na základě zjištění z těchto analýz se snaží nadefinovat vlastní knihovnu. Dále popisuje implementaci této knihovny v programovacím jazyce Java s využitím generování kódu. Knihovna je během vývoje testována a na závěr je pro ni vytvořena ukázková aplikace.

Keywords

ORM, library, database, development, Android, mobile platform, SQLite

Klíčová slova

ORM, knihovna, databáze, vývoj, Android, mobilní platforma, SQLite

Reference

SALVA, Pavel. *ORM library for Android.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Křivka Zbyněk.

ORM library for Android

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Pavel Salva
May 17, 2017

Contents

1 Úvod **3**

2 Vývoj ORM pro Android **4**
2.1 ORM . . . 4
2.2 Android . . . 4
2.3 SQLite . . . 5

3 Průzkum vybraných ORM **6**
3.1 Sugar ORM . . . 6
3.1.1 Definice entit a vztahů . . . 6
3.1.2 Inicializace knihovny a vytvoření databáze . . . 7
3.1.3 Migrace databáze . . . 8
3.1.4 Dotazování na data . . . 8
3.1.5 Optimalizace . . . 9
3.2 SquiDB . . . 9
3.2.1 Definice entit a vztahů . . . 9
3.2.2 Inicializace knihovny a vytvoření databáze . . . 10
3.2.3 Migrace databáze . . . 10
3.2.4 Dotazování na data . . . 11
3.2.5 Optimalizace . . . 12
3.3 Nalezené nedostatky . . . 12
3.3.1 Inicializace . . . 12
3.3.2 Vytvoření schématu . . . 12
3.3.3 Migrace a verze databáze . . . 13
3.3.4 Reflexe . . . 13
3.3.5 Indexy . . . 13
3.3.6 Write-ahead logging . . . 14

4 Vlastní návrh **15**
4.1 Inicializace . . . 15
4.2 Migrace . . . 15
4.3 Definice schémat . . . 16
4.3.1 Anotace @Table . . . 16
4.3.2 Anotace @ColumnName . . . 16
4.3.3 Anotace @PrimaryKey . . . 17
4.3.4 Anotace @Unique . . . 17
4.3.5 Anotace @NotNull . . . 17
4.3.6 Anotace @Igonre . . . 17

4.4 Práce s daty 17
4.4.1 Insert 17
4.4.2 Update 18
4.4.3 Save 18
4.4.4 Delete 19
4.4.5 Čtení dat 19
4.5 Zjištění počtu záznamů 19
4.6 Indexace 20

5 Implementace **22**
5.1 Annotations 22
5.1.1 Anotace v Javě 23
5.1.2 Vlastní anotace 23
5.2 Compiler 23
5.2.1 Anotační procesor 23
5.2.2 Generování kódu pomocí JavaPoet 24
5.3 Joogar 25
5.3.1 Vytváření databází 25
5.3.2 Aktualizace schématu databáze 25
5.3.3 Uživatelské migrace 26

6 Testování **27**
6.1 Příprava na testování 27
6.2 Základní testy 27
6.3 Pokročilé testy 27

7 Ukázková aplikace **29**
7.1 Databázový návrh 29
7.2 Rozhraní aplikace 29
7.2.1 Práce s databází 30

8 Závěr **33**

Bibliography **34**

Chapter 1

Úvod

V dnešní době je trh v oblasti mobilních aplikací jedním z nejvíce se rozvíjejících odvětví v informačních technologiích. Uživatelé dnes denně používají spíše mobilní telefony než obyčejné stolní počítače nebo notebooky. Trhu mobilních telefonů dnes dominují primárně dva operační systémy: Android a iOS. Největší část z trhu[1] obsazuje Android s 86,8%, dále iOS, který tvoří 12,5%, zbylých 0,7% tvoří ostatní operační systémy. Z těchto údajů je zřejmé, že největší uživatelskou základnu má systém Android, pro který také vzniká nejvíce aplikací. To je důvod, proč autor svoji ORM knihovnu bude vyvíjet právě pro tuto platformu.

Téměř každá mobilní aplikace vyžaduje ukládat nějaká perzistentní data. Na Androidu existují primárně tři způsoby jak tyto data zachovat. První možností je ukládat si data do souboru. Tento způsob se ale v praxi moc nevyužívá. Další možností je pomocí Shared Preferences [2], které umožní uložit data ve tvaru klíč-hodnota. Poslední možností je využít databázi. Systém Android umožňuje práci s databází SQLite. Ovšem funkcionalita systémové knihovny obstarávající práci s touto databází je poněkud široká a používání této databáze bez pomocné knihovny je složité. Proto také vznikají různé databázové knihovny, stavějící na té systémové, které tuto funkcionalitu upravují například přidáním mapování dat do objektů. Takovýchto knihoven již vzniklo poměrně velké množství. Tyto existující knihovny autor bere jako inspiraci a poučení pro tvorbu vlastní ORM knihovny.

V první části práce bylo potřeba provézt analýzu vybraných ORM knihoven pro Android. V těchto knihovnách nalézt klady, ale i zápory a z těch se poučit a inspirovat při tvorbě rozhraní pro vlastní knihovnu. O této části práce pojednává kapitola 3.

Na základě informací získaných analýzou ORM knihoven v kapitole 3, si v kapitole 4 nadefinujeme vlastní ORM knihovnu, u které je brán důraz na odstranění nedostatků analyzovaných knihoven.

Následující kapitola 5 pojednává o implementaci námi nadefinované ORM knihovny. Kapitola vysvětluje, jak probíhá generování kódu a následné vytvoření databáze.

Každá open-source knihovna by měla obsahovat jednotkové a integrační testy na správnou funkčnost. O těchto testech pojednává kapitola 6.

Poslední kapitola 7 představuje ukázkovou aplikaci vyvinutou pro účely prezentování funkčnosti knihovny. Kapitola popisuje ovládání a funkčnost aplikace.

[1] Zdroj: http://www.idc.com/promo/smartphone-market-share/os

Chapter 2

Vývoj ORM pro Android

Tato kapitola si klade za cíl uvést do problematiky vývoje ORM knihovny pro Android. Kapitola obsahuje informaci o ORM jako takovém. Dále o vývoji pro systém Android, který převážně čerpá z dokumentace pro vývojáře [2]. A nakonec informace o databázi SQLite [1], která je na Androidu požívána.

2.1 ORM

ORM – tedy „Objektově-relační mapování“. Vychází z myšlenky OOP – „Objektově orientovaného programování“. Je to mapování dat na fyzické databázové schéma a naopak. V relační databázi jsou data uloženy jako jednotlivé řádky v tabulkách. U objektově orientovaných jazyků jsou data reprezentovány instancemi jednotlivých tříd, které reprezentují tabulku v databázi. Díky tomu může vývojář pracovat pouze s objekty a do databáze jako takové nemusí ve většině případů vůbec zasahovat. V ideálním případě by tedy vývojář vůbec nemusel použít jazyk SQL a k databázi by mohl přistupovat a upravovat ji pouze pomocí ORM knihovny [12].

ORM by měla poskytovat všechny CRUD operace (create, retreive, delete, update). Dále by měla poskytnout automatickou konverzi datových typů mezi databází a programovacím jazykem.

2.2 Android

Android je operační systém původně vytvoření Andy Rubinem a jeho týmem v roce 2003. V roce 2005 ale tento systém koupil Google a ten jej vyvíjí nadále. Ze začátku byl Android považován jen jako operační systém pro mobilní telefony. Dnes ale můžeme Android nalézt nejen v telefonech, ale třeba v tabletech, televizích či různých malých desktopových počítačích.

Operační systém je založen na Linuxovém jádře 2.6 různých verzí. Toto jádro se stará o zabezpečení systému jako celku, správu paměti, správu procesů, přístup k síti a ovladačům všech vnitřních senzorů a komponent. K funkcím jádra jednotlivé aplikace nepřistupují přímo, ale prostřednictvím Android API [13].

Aplikace se na systém Android nejčastěji vyvíjí pomocí programovacího jazyku Java. Tyto aplikace dále Android spouští pomocí virtuálního stroje. Díky virtuálnímu stroji dokáže Android lépe optimalizovat paměť a hardwarové prostředky v mobilním prostředí.

Do roku 2013 byl jako virtuální stroj využívám Dalvik. Toho ale nahradil ART (Android Runtime), který se používá do dnes.

2.3 SQLite

SQLite reprezentuje relační databázový sytém obsažen v jedné malé knihovně. Jedná se o systém velmi šetrný k paměti zařízení, což je velký výhoda při použití například pro mobilní telefony. Další plus této knihovny je snadné použití v rámci aplikace. Nepracuje totiž na principu klient-server, jako většina databázových systémů, ale představuje knihovnu, která je přilinkována k aplikaci [1].

SQLite je databázový systém využívající dialekt SQL, který je postaven na základech standardu SQL-92. Vykazuje ovšem určité odlišnosti. Zde je jejich výčet:

- Neexistuje datový typ `DATE`, pouze `DATETIME`,
- příkaz `ALTER TABLE` je omezený,
- neexistuje kontrola datových typů na vstupu.
- do tabulky nelze vložit cizí klíč,
- nelze použít spojení tabulek typu `RIGHT OUTER JOIN`, `FULL OUTER JOIN`.

Chapter 3

Průzkum vybraných ORM

Tato kapitola obsahuje průzkum a hodnocení vybraných aktuálně používaných ORM knihoven pro Android. Autor si klade za cíl u každé knihovny zhodnotit tyto části:

- Velikost uživatelské základny,
- počet aktivních přispěvatelů na serveru Github,
- poslední příspěvek na serveru Github - aktuálnost projektu,
- jak lze definovat entity a vztahy,
- jak lze inicializovat knihovnu a vytvořit databázi,
- jak se provádí migrace,
- jaké jsou možnosti dotazování na data,
- optimalizace.

3.1 Sugar ORM

Sugar ORM [11] je jeden z nejoblíbenějších ORM pro Android. Má nejen velkou uživatelskou základnu, ale i velkou základnu přispěvatelů do kódu, kteří se tak na vývoji této knihovny podílí. Poslední příspěvek byl ale v době psaní této práce před 6ti měsíci, což může naznačovat nejen možnou stagnaci vývoje knihovny, ale i možnou vyzrálost.

3.1.1 Definice entit a vztahů

Definice entit probíhá stejně jako u všech ORM definicí třídy. Jsou dvě možnosti jak tyto entity vytvořit. První možnost je, že třída entity dědí od `SugarRecord` - ukázka v příkladu 3.1.

```
public class Book extends SugarRecord {

String title;
String edition;

}
```

Příklad 3.1: Definice entit v Sugar ORM (1)

V tomto případě se nemusí definovat ID, které bude vždy `private long id`. Definují se pouze tedy jednotlivé atributy dané entity.

Druhá možnost je definovat entitu bez dědění `SugarRecord`, ale s pomocí anotací viz. příklad 3.2.

```
@Table
public class Book {

private Long id;
String title;
String edition;

}
```

Příklad 3.2: Definice entit v Sugar ORM (2)

Oba dva předchozí příklady dokáží namapovat/vytvořit entitu s názvem `Book`, která bude mít atributy id, title, edition. Sugar dále nabízí přidat do entity atribut, který se nebude mapovat ani ukládat - stačí přidat anotaci `@Ignore` nad daný atribut.

Co se týče definic vztahů mezi entitami, Sugar ORM aktuálně nabízí pouze možnost definovat vztah ONE:MANY. Definuje se tak, že do entity, která je na straně kardinality MANY, se přidá odkaz na entitu ze strany ONE. Rozšiřme si tedy v příkladu 3.3 naši entitu `Book` ještě o autora.

```
public class Author extends SugarRecord {
String name;
}

public class Book extends SugarRecord {
String title;
String edition;
Author author;
}
```

Příklad 3.3: Definice vztahů v Sugar ORM

3.1.2 Inicializace knihovny a vytvoření databáze

Knihovna se přidává do projektu přidáním závislosti do souboru build.gradle. Pro inicializaci knihovny v projektu je ale bohužel třeba upravit Manifest soubor. Dále hlavní třída projektu `Application` musí dědit od `SugarApp`, což není moc vhodné. Dále se v Manifest souboru definují další potřebné atributy, jako je název databáze, aktuální verze, povolení logů atd.

Dle autorova názoru by se všechny tyto informace měli od Manifestu a potažmo od celé třídy `Application` odstínit a nastavovat se přímo v kódu při inicializaci knihovny. V příkladu 3.4 je část souboru Manifest, kde se inicializuje knihovna:

```
<application android:label="@string/app_name"
android:name="com.orm.SugarApp">
.
.
```

```
<meta-data android:name="DATABASE"
   android:value="sugar_example.db" />
<meta-data android:name="VERSION" android:value="2" />
<meta-data android:name="QUERY_LOG" android:value="true" />
<meta-data android:name="DOMAIN_PACKAGE_NAME"
   android:value="com.example" />
.
.
</application>
```

Příklad 3.4: Inicializace knihovny v Sugar ORM

Samotná struktura tabulek a atributů se při vytváření databáze použije z nadefinovaných entit, jejichž vytvoření je ukázáno výše. Tyto entity musí být uloženy v jednom balíku (package) a tento balík musí být zapsán v Manifestu pod klíčem DOMAIN_PACKAGE_NAME. Dle autorova názoru to není úplně ideální cesta. Může se například stát, že když uživatel bude mít package pro entity, tak tam budou i entity, ze kterých nebude chtít vytvářet databázi, ale budou se používat třeba na komunikaci se serverem.

3.1.3 Migrace databáze

Migrace v Sugar ORM se provádí čistými SQL příkazy, které se ukládají do souborů číslovaných od 1, tedy například 1.sql atd. Tyto soubory musí být uloženy ve složce sugar_updates, ve složce assets. Dále je potřeba zvednout verzi databáze v Manifestu. Když nastavíme verzi na 3, tak se provedou sql příkazy ze souborů 1.sql, 2.sql a 3.sql.

3.1.4 Dotazování na data

Dotazování na data jde v Sugar ORM třemi způsoby. První příklad 3.5 je čistý SQL dotaz:

```
List<Note> notes = Note.findWithQuery(Note.class, "Select *
   from Note where name = ?", "satya");
```

Příklad 3.5: SQL dotaz v Sugar ORM

V příkladu 3.6 je ukázka dotazování pomocí funkce find, která splňuje konvence SQLiteDatabase query method:

```
Note.find(Note.class, "name = ? and title = ?", "satya",
   "title1");
```

Příklad 3.6: Dotaz pomocí find v Sugar ORM

A nakonec v příkladu 3.7 dotaz pomocí QueryBuilder:

```
Select.from(TestRecord.class)
.where(Condition.prop("test").eq("satya"),
Condition.prop("prop").eq(2))
.list();
```

Příklad 3.7: Dotaz pomocí QueryBuilder v Sugar ORM

Sugar ORM splňuje všechny základní způsoby dotazování na data. Nejvhodnější způsob dotazování je zcela jistě QueryBuilder, který však bohužel u Sugar ORM nepodporuje operaci spojení, která je při reálné práci s databází často využívána.

3.1.5 Optimalizace

Je několik druhů optimalizace, které lze v databázi použít. Na databázové úrovni jsou to například indexy. Ty jdou ale v Sugar ORM vytvořit jen pomocí čistého SQL dotazu a bohužel ne pomocí anotace. Dále je více způsobů, jak mapovat entity v ORM. V Sugar ORM probíhá mapování pomocí reflexe, která ale není tak rychlá jako například generování kódu.

3.2 SquiDB

SquidDB [3] je aktuálně jeden z nejlépe hodnocených ORM pro Android. SquiDB ovšem není jen na Android. Díky tomu, že je jeho hlavní logika napsaná v čisté Javě, jde jeho jádro zkompilovat pomocí nástroje J2ObjC do jazyka Objective C, který se využívá na platformě iOS. Je to aktuálně vyvíjená knihovna - poslední příspěvek je v době psaní této práce starý sedm dní.

3.2.1 Definice entit a vztahů

Tabulky a jejich entity se definují stejně jako u všech ORM v Javě v POJO objektech (plan old Java object). SquiDB dbá na co největší minimálnost co se týče definicí databázové struktury, tudíž se nedefinují žádné ID, které jsou vytvářeny automaticky. Nicméně je možné definovat si i vlastní primární klíč pomocí anotace @PrimaryKey.

Díky tomu, že SquiDB předgenerovává kód, je možné v anotaci `@TableModelSpec` definovat nejen název tabulky, ale i název třídy, do které se bude tato tabulka mapovat. Díky tomu je možné nadefinovat celou databázi v jedné třídě pomocí vnořených tříd, ze kterých se pak vygenerují entity, se kterými se pak bude reálně pracovat.

Názvy sloupců je možné generovat z jednotlivých atributů třídy anebo definovat pomocí anotace `@ColumnSpec`. V příkladu 3.8 je ukázková definice entity:

```
// Toto je schéma tabulky
@TableModelSpec(className = "Person", tableName = "people")
class PersonSpec {
// Textový sloupec pojmenovaný "firstName"
String firstName;

// Textový sloupec pojmenovaný "lastName"
String lastName;

// Sloupec typu long pojmenovaný "creationDate", ale
   odkazovaný jako "birthday" při práci s modelem
@ColumnSpec(name = "creationDate")
long birthday;
}
```

Příklad 3.8: Schéma tabulky v SquiDB

SquiDB nezprostředkovává žádnou možnost jak definovat vztahy ve schématech. Ošetřování vztahů tudíž zůstává plně v rukou programátora, který bude knihovnu používat.

3.2.2 Inicializace knihovny a vytvoření databáze

Jako skoro každá knihovna pro Android se přidává do projektu pomocí přidání závislosti do gradle souboru. Pro inicializaci knihovny je potřeba vytvořit třídu, která dědí od `SquidDatabase` a implementuje její abstraktní třídy. V těchto metodách se definují veškeré informace potřebné pro vytvoření/inicializaci databáze. V příkladu 3.9 je ukázková třída, která tyto metody implementuje:

```
public class MyDatabase extends SquidDatabase {
private static final int VERSION = 1;

public MyDatabase() {
super();
}

@Override
public String getName() {
return "my-database.db";
}

@Override
protected Table[] getTables() {
return new Table[]{
Person.TABLE // Zde se přidá odkaz na všechny entity
};
}

@Override
protected int getVersion() {
return VERSION;
}
// Možné přidat další metody na migrace atd...
}
```

Příklad 3.9: Vytvoření databáze v SquiDB

Pro inicializaci je potřeba vytvořit singleton instanci této třídy při spuštění aplikace. Tedy například při inicializaci třídy `Application`.

3.2.3 Migrace databáze

Pro provedení změny v databázi je potřeba ve třídě `MyDatabase` v metodě `getVersion()` zvednout verzi. Při inicializaci databáze se tato verze porovná s verzí databáze a pokud se tyto verze neshodují, volá se metoda `onUpgrade()`, kde se tyto změny provedou. Ukázka uživatelské migrace je vidět na příkladu 3.10.

```
public class MyDatabase extends SquidDatabase {
/* ... ostatní metody ... */
@Override
protected int getVersion() {
return 3;
```

```
}

@Override
protected boolean onUpgrade(ISQLiteDatabase db, int
   oldVersion, int newVersion) {
// Díky konstrukci switch-case je můžou provolat všechny
   migrace pro jednotlivé verze.
switch(oldVersion) {
case 1:
tryAddColumn(User.BIRTHDAY);
case 2:
tryAddColumn(User.PROFILE_IMAGE_ID);
tryCreateTable(ProfileImage.TABLE);
}
return true;
}
}
```

Příklad 3.10: Migrace databáze v SquiDB

Pro provedení migrací má SquiDB nachystáno několik pomocných metod, které vykonání migrace zjednoduší. Zde je seznam metod:

- tryCreateTable() & tryDropTable(),
- tryCreateView() & tryDropView(),
- tryAddColumn(),
- tryCreateIndex() & tryDropIndex(),
- tryExecStatement(),
- tryExecSql().

3.2.4 Dotazování na data

Díky tomu, že SquiDB funguje na bázi generování kódu a v rámci tohoto generování vytváří i pro jednotlivé entity tzv. „QueryBuilder“, jdou téměř všechny dotazy poskládat právě z tohoto builderu. SquiDB prezentuje dotazování na data pomocí jejich „QueryBuilderu“, jako doporučenou možnost, jak se na tyto data dotazovat.

Pro porovnání v příkladu 3.11 je dotaz tvořený pomocí čistého jazyka SQL.

```
String sql = "select " + PersonColumns.AGE + ", " +
   ProfileImageColumns.URL +
" from " + PERSON_TABLE + " left join " + PROFILE_IMAGE_TABLE
   + " on " +         PersonColumns._ID + " = " +
   ProfileImageColumns.PERSON_ID +
" where " + PersonColumns.NAME + " = ?" + " AND "
+ PersonColumns.AGE + " >= ?" + " ORDER BY " +
   PersonColumns.AGE + " ASC";
```

```java
String[] sqlArgs = new String[]{"Sam", Integer.toString(18)};
```
Příklad 3.11: Příklad obyčejného SQL dotazu

Dále je v příkladu 3.12 ten samý dotaz vytvořený pomocí třídy na sestavování dotazů:

```java
Query query = Query.select(Person.AGE, ProfileImage.URL)
.from(Person.TABLE).leftJoin(ProfileImage.TABLE, Person.ID
.eq(ProfileImage.PERSON_ID)).where(Person.NAME.eq("Sam")
.and(Person.AGE.gte(18)));
```
Příklad 3.12: Dotaz na data v SquiDB

Jak je z příkladu vidět, použití SquiDB `Query` díky generování kódu výrazně zjednodušuje dotazování na data. I díky podpoře join dotazů se z SquiDB Query stává plnohodnotný nástroj na dotazování dat.

3.2.5 Optimalizace

SquiDB knihovna umožňuje plnohodnotné využívání indexace databáze a to nejen díky čistým SQL dotazům, ale i přímo přes knihovnu v rámci inicializace databáze. Dále díky tomu, že SquiDB používá generování kódu a ne reflexi, stává se z něj vcelku optimalizovaná a výkonná databázová knihovna.

3.3 Nalezené nedostatky

V této kapitole si shrneme nedostatky, které dle autorova názoru tíží zkoumané ORM knihovny. Autor se dále pokusí tyto nedostatky napravit u návrhu API pro svou ORM knihovnu.

3.3.1 Inicializace

U obou případů zkoumaných knihoven je zbytečně složitý proces inicializace aplikace. Přidání knihovny do závislostí do souboru gradle je nutný krok, který se musí provést u každé knihovny. Nicméně dále by dle autorova názoru mělo stačit jen inicializovat knihovnu v kódu vytvořením její instance a předáním aplikačního kontextu.

Většina mobilních aplikací bude používat pouze jednu databázi, tak je zbytečné definovat její název, umístění atd. Ve snaze o co nejjednodušší použití autor navrhuje vytvořit automaticky jednu výchozí databázi při základní inicializaci knihovny a tu používat. Ale databáze by měla mít i možnost vytvořit více databází, například pomocí rozšířené inicializace nebo přidáním anotace databáze k jednotlivým schématům tabulek.

3.3.2 Vytvoření schématu

Samotná definice schémat tabulek je ve všech ORM pro Android stejná a to pomocí POJO tříd. Většina ORM knihoven se však rozchází v tom, jak jí tyto třídy předat.

Nejčastější možností je předání pole těchto tříd při inicializaci knihovny nebo při vytváření databáze pomocí tříd na jejich vytváření (záleží na konkrétní knihovně). Tento přístup je však zbytečně zdlouhavý a při každém vytvoření nové entity se musí knihovně předat nové pole. Často se může stát, že na to programátor zapomene a tím vznikají chyby.

Další možností je definovat package, ve kterém se jednotlivé třídy entit nachází. Tento přístup je dle autorova názoru lepší než ten předchozí. Bohužel je dnes u nových verzí

systému Android i nový systém kompilace do .apk souborů, u kterého je problém správně detekovat třídy obsažené v konkrétním package.

Poslední navrženou možností je mít třídy definující entity kdekoliv v projektu. Třída se pozná podle toho, že bude mít anotaci `@Table`. Díky anotačnímu procesoru se tak dají najít všechny třídy, které jsou takto anotované. Díky tomu má programátor používající knihovnu volnou ruku v tom, jakou dá svému projektu strukturu a navíc se nemusí zabývat předáváním jednotlivých tříd entit, protože to za něj udělá anotační procesor.

3.3.3 Migrace a verze databáze

Databázové migrace si můžeme pro naše účely rozdělit na dva druhy: Automatické - ty které provádí ORM knihovna automaticky a uživatelské - ty definuje sám programátor a předává je knihovně. S migracemi úzce souvisí i pojem verze databáze. Může to být jak číslo, tak i nějaká textová hodnota, která udává aktuální verzi databáze. Znamená to tedy, že pokud vyvíjíme aplikaci, kterou již někde používáme a vydáme update, který zasahuje i do struktury databáze, musíme v nové verzi aplikace zjistit verzi databáze a pokud se liší od té, se kterou nová verze aplikace pracuje, musíme nad touto databází provést migraci a upravit ji do nové podoby.

Automatické migrace některé knihovny vůbec neumožňují a veškeré migrace si musí programátor řešit sám pomocí uživatelských migrací. Pak jsou knihovny, které dokáží automaticky aktualizovat schéma tabulek databáze. Jelikož je ale nepraktické kontrolovat při každém spuštění aplikace, zda se schéma tabulek ve třídách shoduje se schématem tabulek v databázi, tak se nejdříve porovnají verze aktuální databáze a verze, kterou aplikace očekává. Až pokud se tyto verze neshodují, pouští se kontrola schémat. Z pohledu programátora mobilních aplikací to znamená, že při každé změně schématu ve třídě musí v kódu zvednout i verzi databáze. Což je dle autorova názoru zbytečný krok navíc a díky generování kódu se tomu lze jistě vyhnout.

Uživatelské migrace si definuje programátor sám. Obvykle jsou to čisté SQL příkazy, které se spustí nad databázovým systémem. Je více způsobů, jak tyto migrace předat. Nejčastější způsob je vytvoření jednotlivých migračních souborů ve složce assets. Tyto soubory jsou pojmenovány verzí databáze, do které by se měla databáze dostat po aplikování migrace. Tento způsob je dle autora zbytečně zdlouhavý a navíc zahrnuje vytváření nových souborů s informacemi, které jdou zcela jistě předat lépe.

Další přístup, jak uživatelské migrace řešit, je ten, že se v kódu zavolá například metoda `onUpdate`, které se předává aktuální a očekávaná verze databáze, a uživatel si tyto migrace provede v této metodě sám. Toto je dle autorova názoru již lepší přístup než ten předchozí.

3.3.4 Reflexe

Většina ORM knihoven pro Android používá reflexi. Nejčastěji ke čtení nadefinovaného schématu ze třídy, nebo pro mapování dat do těchto tříd. Reflexe je obvykle jednou z nejpomalejších částí celé knihovny, proto je lepší se jí vyhýbat a použít raději místo ní anotační procesor a generování kódu.

3.3.5 Indexy

S používáním databází je úzce spjata i její indexace. Většina knihoven neumožňuje indexaci sloupečků pomocí anotací kupříkladu. V takovém případě si musí programátor vytvořit indexy sám a to například využitím uživatelských migrací.

3.3.6 Write-ahead logging

Write-ahead logging (zkráceně WAL) je technika, která úzce souvisí s databázovými transakcemi . Tato technika zajišťuje, aby datové soubory, tedy tabulky a indexy, byly modifikovány až poté, co bude uložen jejich záznam do logu [10]. Z těchto transakčních logů lze pak restaurovat obsah databáze, aniž by se musel vynucovat zápis na disk po dokončení potvrzení transakce. V praxi to znamená to, že můžeme na Androidu pracovat s jednou databází naráz z více vláken, a to i při otevřené transakci.

WAL je na Androidu podporován od verze 3.0. Od této verze stačí na instanci `SQLiteDatabase` zavolat metodu `enableWriteAheadLogging`. Problém je, že když používáme ORM knihovnu, jsme obvykle od této třídy odstíněni. Většina ORM knihoven bohužel nepodporuje provolat přes ně tuto metodu. V tom případě se musí WAL zapínat pomocí čistého SQL příkazu, což není zrovna přívětivé pro uživatele knihovny.

Chapter 4

Vlastní návrh

V této kapitole bude autor prezentovat jeho návrh ORM knihovny pro Android. Při tvorbě návrhu bude brán zřetel na nedostatky, ale také na zajímavé návrhy u zkoumaných knihoven. Návrh knihovny předpokládá, že knihovna bude používat generování kódu pro vytváření, mapování a práci s entitami reprezentující tabulky databáze.

4.1 Inicializace

Veškeré entity, které tvoří databázi, se budou vyhledávat pomocí anotací. V těchto anotacích bude i informace o tom, do jaké databáze daná entita patří. Díky tomu, že si knihovna dokáže všechny tyto informace zjistit sama, na inicializaci knihovny a vytvoření databází stačí zavolat pouze metodu inicializace s předáním aplikačního kontextu viz příklad 4.1.

```
Joogar.init(context)
```

Příklad 4.1: Inicializace knihovny v Joogar2

Po zavolání této inicializace budou vytvořeny všechny databáze a entity, které uživatel anotoval knihovními anotacemi.

4.2 Migrace

Migrace budou probíhat dvěma způsoby. Prvním způsobem je úprava schématu tabulky. Pokud se změní něco ve schématu, tak to knihovna automaticky změní i v databázi. Problém nastává jen u změny datového typu již vytvořeného sloupce, tato změna povolena nebude.

Druhým způsobem jsou uživatelské migrace. Ty si může uživatel nadefinovat sám pomocí SQL příkazu. Ve třídě `Joogar` pro každou databázi vytvoří metoda s názvem `setJoogarDatabaseCustomMigrations`, kde Joogar je název databáze. Této metodě se předá pole textových hodnot obsahující SQL příkazy pro migraci. V příkladu 4.2 je vidět předání migrací pro databázi Cars.

```
String[] migrations = new String[] {"migration1....",
   "migration2..."};
Joogar.getInstance()
.setCarsDatabaseCustomMigrations(migrations);
```

Příklad 4.2: Uživatelské migrace v Joogar2

Jak se bude rozeznávat, která migrace se má provést, bude vysvětleno v kapitole Implementace.

4.3 Definice schémat

Jednotlivé třídy reprezentující tabulky v databázi mohou být uloženy kdekoliv v rámci aplikačního balíčku. Vše se bude vyhledávat pomocí anotačního procesoru, takže tyto třídy musí být řádně anotovány pomocí knihovních anotací. V příkladu 4.3 je ukázková definice entity a jejich atributů, tedy třídy a jejich sloupců.

```
@Table(name = "PersonTable", database = "database")
public class Person {

@PrimaryKey
public Long id;

@ColumnName(value = "nameColumn")
public String name;

@Unique
@NotNull
public String bornNumber;

@Ignore
public int age;

}
```

Příklad 4.3: Definice schématu v Joogar2

4.3.1 Anotace @Table

Anotace `@Table` je anotací třídy. Pomocí této anotace se označují třídy, které bude knihovna zpracovávat jako entity. Anotace má dva volitelné atributy:

- name - Volitelný atribut, definuje název tabulky. Pokud není zadán, generuje se z názvu třídy.
- database - Volitelný atribut. Definuje databázi pomocí jejího názvu. Pokud není zadán, používá se výchozí název databáze - `joogar`.

4.3.2 Anotace @ColumnName

Anotace `@ColumnName` je volitelná anotace na atribut entity. Ve výchozím stavu se názvy sloupce pro tabulku generují z názvů atributů tabulky. Pokud však chce mít uživatel z nějakého důvodu jiný název sloupce než atribut třídy, použije právě tuto anotaci. Název sloupečku se nastavuje do atributu value.

4.3.3 Anotace @PrimaryKey

Každá tabulka musí mít primární klíč. Stejně tak to je i u navrhované knihovny. Každá entita musí mít minimálně jeden atribut anotovaný pomocí `@PrimaryKey`. Pomocí této anotace lze vytvořit jednoduchý i složený primární klíč viz. příklad 4.4.

```
@Table
public class Car {

@PrimaryKey
private String car_uuid;

@PrimaryKey
private long driver_id;

}
```

Příklad 4.4: Složený primární klíč v Joogar2

4.3.4 Anotace @Unique

Anotace `@Unique` je anotace na atributy třídy. Pokud uživatel anotuje atribut touto anotací, knihovna označí sloupec této tabulky jako unikátní, což znamená, že nedovoluje ve sloupci duplicitní hodnoty.

4.3.5 Anotace @NotNull

Anotace `@NotNull` je další anotace na atributy třídy. Pokud je atribut třídy anotován touto anotací, v databázi se mu nastaví, že není povoleno, aby tento atribut a sloupec v databázi byl prázdný, tedy neměl žádnou hodnotu.

4.3.6 Anotace @Igonre

Anotace `@NotNull` je poslední anotace na atributy třídy. Tato anotace nesouvisí přímo s databází. Je informací pro knihovnu, že s daným atributem nemá nakládat jako se sloupečkem v tabulce. V tom případě jej bude knihovna ignorovat.

4.4 Práce s daty

Pro práci s databází s daty je vygenerována metoda `JoogarRecord`, která obsahuje základní metody potřebné pro tyto účely. Dále však jsou vygenerovány třídy pro každou entitu s postfixem `Table`. Tedy pro `Person` to bude `PesronTable`. Každá tato třída pro jednotlivé metody obsahuje nejen základní metody pro práci s danou entitou, ale i rozšířené metody, například o podporu transakcí.

4.4.1 Insert

Pokud bude uživatel chtít vytvořit nový záznam v tabulce, použije k tomu `JoogarRecord.insert(...)`, který umožní vložení všech databázových entit, které byly označeny pomocí anotace `@Table`. V příkladu 4.5 je ukázka vytvoření nového záznamu.

```
Person person = new Person("John", "Smith");
JoogarRecord.save(peson);
```

Příklad 4.5: Insert v Joogar2

Může nastat situace, kdy bude uživatel potřebovat uložit naráz více entit stejného typu. V tomto případě je vhodné to provést v transakci. Proto se tyto případy ve třídě reprezentující danou entitu generují dvě další metody pro provedení vložení v transakci. Pro entitu Person to budou tyto metody ve třídě `PersonTable`:

- PersonTable.insertInTx(final Collection<Person> objects)
- PersonTable.insertInTx(final Person first, final Person... others)

4.4.2 Update

Pokud chce uživatel upravit již vložený záznam v tabulce, použije k tomu `JoogarRecord.update(...)`. Ukázka příkazu update je v příkladu 4.6.

```
Person person = JoogarRecord.findPersonById(1L);
person.setName("Jim");
JoogarRecord.update(peson);
```

Příklad 4.6: Update v Joogar2

V ojedinělých případech se stává, že chceme upravit více záznamů naráz a použití transakcí přijde vhod. I na to je knihovna připravena dvěmi metodami ve třídě reprezentující danou entitu. Pro tabulku `Person` to jsou metody ve třídě `PersonTable`:

- PersonTable.updateInTx(final Collection<Person> objects)
- PersonTable.updateInTx(final Person first, final Person... others)

4.4.3 Save

Může nastat situace, kdy si uživatel není jistý, zda záznam s ID, který chce přidat nebo upravit, již v tabulce existuje. Pro tento případ je ideální použít metodu `JoogarRecord.save(...)`. Ta provede SQL příkaz `INSERT OR UPDATE`, který buď vloží nebo vytvoří nový záznam. Ukázka použití v příkladu 4.7.

```
Person person = new Person("James", "Smith");
JoogarRecord.save(peson);
```

Příklad 4.7: Save v Joogar2

Stejně jako příkaz insert, tak i příkaz save podporuje transakce. Pro `save` se také generují dvě metody pro uložení v transakci. Pro entitu `Person` jsou to tyto metody ve třídě`PersonTable`:

- PersonTable.saveInTx(final Collection<Person> objects)
- PersonTable.saveInTx(final Person first, final Person... others)

4.4.4 Delete

Pro smazání záznamu slouží v `JoogarRecord` metoda `delete`. Podobně jako ostatní metody, lze ji zavolat pro všechny entity databáze. V příkladu 4.8 je ukázka pro entitu Person.

```
Person person = JoogarRecord.findPersonByID(1L);
JoogarRecord.delete(person);
```

Příklad 4.8: Delete v Joogar2

Mazání více záznamů lze také provést i v transakci. Například pro entitu `Person` k tomu slouží tyto dvě metody ve třídě `PersonTable`:

- PersonTable.deletePersonInTx(final Collection<Person> objects)
- PersonTable.deleteInTx(final Person first, final Person... others)

4.4.5 Čtení dat

Čtení dat z databáze lze provádět několika způsoby. První způsob je dotaz na entitu pomocí primárního klíče. Pro každou entitu se v `JoogarRecord` vygeneruje metoda pro vyhledání dané entity pomocí primárního klíče. Metoda má název `find<název_entity>By<název_PK>`. Použití pro entitu `Person` s primárním klíčem ID je vidět v příkladu 4.9.

```
Person person = JoogarRecord.findPersonByID(1L);
```

Příklad 4.9: Find pomocí primárního klíče v Joogar2

Všechny ostatní dotazy by se měly dělat pomocí metod na vytváření dotazů. Pro tyto účely je v každé pomocné třídě vygenerovaná pro každou entitu metoda `where()`. Tato metoda vrací instanci další pomocné třídy pro entitu, tentokrát s postfixem `Where` (tedy pro `Person` je to `PersonWhere`). Pomocí této třídy se dá jednoduchým způsobem nadefinovat `WHERE` část `SQL` dotazu. Pomocí tohoto nadefinovaného dotazu se pak dají vyhledávat jednotlivé záznamy nebo třeba zjišťovat počet těchto záznamů. V příkladu 4.10 je ukázka vyhledávání záznamů tímto způsobem.

```

PersonWhere where = new PersonTable.where()
.name().in("Joe", "John", "Alice")
.and(new PersonWhere()
.birthYear().between(1970, 1990)
.or()
.birthYear().eq(1969).birthMonth().gtEq(10)
);

JoogarCursor<Person> = PersonTable.find().where(where).find();
```

Příklad 4.10: Vyhledávání záznamů pomocí nadefinovaného WHERE v Joogar2

4.5 Zjištění počtu záznamů

Pro zjištění počtu záznamů pro danou entitu jsou vygenerovány dvě metody. Obě metody se nachází v pomocné třídě pro danou entitu. První metoda vrací celkový počet záznamů. Ukázka použití pro entitu `Person` je v příkladu 4.11.

```java
int count = PersonTable.count();
```

Příklad 4.11: Zjištění počtů záznamů v Joogar2

Pro zjištění počtu záznamů podle nějaké podmínky je potřeba si nadefinovat `WHERE` stejně tak, jak to bylo v příkladu 4.10. Po nadefinování se dotaz předá do metody `count`, která vrátí počet záznamů. V příkladu 4.12 je ukázka použití - dotaz na počet lidí, kteří se narodili mezi 1970 až 1990.

```java
PersonWhere where =
new PersonTable.where().birthYear().between(1970, 1990);
int count = PersonTable.count(where);
```

Příklad 4.12: Zjištění počtu záznamů s podmínkou v Joogar2

4.6 Indexace

Knihovna bude podporovat vytváření indexů v entitách. Indexy se budou definovat pomocí anotací. Pokud bude uživatel chtít vytvořit index jen na jeden atribut, stačí jej anotovat pomocí anotace `@TableIndex`. V příkladu 4.13 je vidět ukázka vytvoření indexu pro atribut `bornNumber`.

```java
@Table
public class Person {
@PrimaryKey
public Long id;
public String name;
public String surname;

@Index
public String bornNumber;
}
```

Příklad 4.13: Vyvoření indexu pro jeden atribut v Joogar2

Pro optimalizaci složitějších SQL dotazů je potřeba vytvořit složený index z více atributů. Na to slouží anotace třídy `@TableIndex`, která má atribut `columns`, kterému se přiřazuje pole textových hodnot. Při vytváření složených indexů je potřeba dávat pozor i na pořadí jednotlivých atributů. Ukázka použití složeného indexu je ukázána v příkladu 4.14.

```java
@TableIndex(columns = {"name", "surname", "bornNumber"})
@Table
public class Person {
@PrimaryKey
public Long id;
public String name;
public String surname;A o
public String bornNumber;
}
```

Příklad 4.14: Vytvoření složeného indexu v Joogar2

V případě, že bude chtít uživatel vytvořit několik složených indexů pro jednu entitu, nemůže použít vícekrát anotaci `@TableIndex`. Pro tyto případy existuje anotace `@TableIndexes`, která má atribut `indexes`, kterému se předává pole jednotlivých `@TableIndex` anotací. V příkladu 4.15 je ukázka použití `@TableIndexes`.

```
@TableIndexes (indexes = {
@TableIndex (columns = {"name", "surname"}),
@TableIndex (columns = {"name", "surname", "bornNumber"})
})
@Table
public class Person {
@PrimaryKey
public Long id;
public String name;
public String surname;A o
public String bornNumber;
}
```

Příklad 4.15: Vytvoření složeného indexu v Joogar2

Chapter 5

Implementace

V rámci této kapitoly autor popíše základní postupy při implementaci knihovny. Protože knihovna, jako taková, je nadefinována jako poměrně robustní, bude na ni autor v rámci open-source spolupracovat s Janem Bínou z ČVUT v Praze. Z komentářů ve zdrojovém kódu je vidět, kdo na které třídě pracoval. Pokud na jedné třídě pracovali oba, jsou okomentovány jednotlivé metody. Knihovna, jako taková, staví na open-source knihovně Joogar. V konečném výsledku je však z této knihovny použito jen pár tříd.

Celá knihovna včetně ukázkové aplikace je tvořena z několika modulů, potažmo balíčků. Tyto balíčky oddělují jednotlivé logické části aplikace. Schéma balíčků a jejich závislosti můžete vidět na obrázku 5.1. V následujících podkapitolách autor rozebere jednotlivé balíčky.

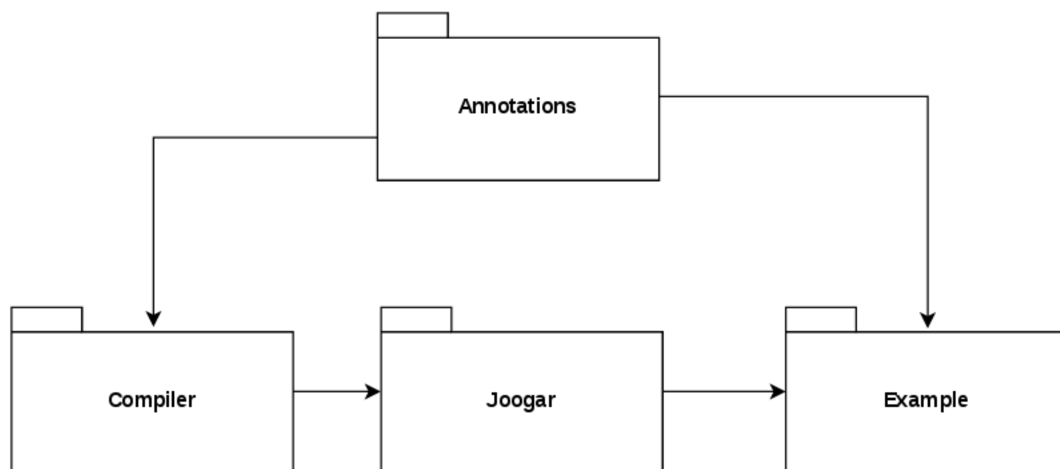


Figure 5.1: Závislost balíčků v Joogar2

5.1 Annotations

Balíček `Annotations` obsahuje veškeré anotace, které bude uživatel potřebovat při pracování s naší knihovnou a definici entit.

5.1.1 Anotace v Javě

Anotace je jistá forma metadat, jinými slovy je anotace informace, která popisuje ostatní data [6]. Anotace jdou aplikovat na třídu, atribut nebo metodu. Na co bude anotace používána se nastavuje pomocí anotace `@Target` a jejího atributu `value`.

Existují tři druhy anotací podle toho, kdy budou zpracovány. Tato hodnota se nastavuje pomocí anotace `@Retention` při vytváření souboru. Zde je jejich seznam a popis:

SOURCE Anotace jsou zpracovávány při kompilaci programu, za běhu programu už k těmto anotacím nemáme přístup.

CLASS Anotace jsou přístupné ještě při překladu do .class souborů. Toto je defaultní hodnota anotace retention. Ovšem v praxi se moc nevyužívá, nejčastěji na optimalizaci na úrovni byte-kódu.

RUNTIME Anotace jsou přístupné za běhu programu. To znamená, že k nim můžeme přistupovat přímo v kódu za běhu aplikace.

Dále se dá anotaci přidat i několik atributů. Tyto atributy mohou mít i výchozí hodnoty a v tom případě není potřeba při použití dané anotace tyto atributy nastavovat.

5.1.2 Vlastní anotace

Jak již autor zmiňuje výše, balíček `Annotations` obsahuje veškeré anotace potřebné pro Joogar2 knihovnu. Všechny anotace mají nastaveny zpracování při kompilaci programu - tedy „@Retention(RetentionPolicy.SOURCE)“. Většina knihovních anotací je na atributy třídy, pouze anotace entity `@Table` a anotace indexů `@TableIndex` a `@TableIndexes` jsou anotace třídy.

5.2 Compiler

Balíček `Compiler` zajišťuje generování kódu pro aplikaci, která knihovnu používá. Celá logika je postavena na anotačním procesoru [5]. Tento balíček je závislý na balíčku `Annotations`, ve kterém jsou jednotlivé anotace, které anotační procesor zpracovává. Generování kódu probíhá pomocí knihovny JavaPoet [14].

5.2.1 Anotační procesor

Jak už název napovídá, anotační procesor zprostředkovává zpracování anotací. V tomto případě anotací, které jsme si vytvořili v balíčku `Annotations`, a se kterými bude uživatel knihovny anotovat jednotlivé entity a jejich atributy. Existují dva druhy anotačních procesorů a tedy i přístupů, jak anotace zpracovávat. První z nich je zpracování anotací až za běhu aplikace. Tento přístup využívá i většina ORM knihoven pro Android. Jelikož tento přístup značně zpomaluje běh aplikace, Joogar2 bude využívat anotační procesor již při kompilaci. Díky tomu, že se všechny potřebné třídy nachystají a nastaví již při kompilaci, nemusí se tímto problémem zabývat spouštěná aplikace a její běh se zrychlí.

Nastavit anotační procesor zpracovávající anotace při kompilaci vyžaduje několik kroků. Je potřeba si nachystat daný balík v naší knihovně, který bude tento anotační procesor obsahovat. Není potřeba, aby v tomto balíku byla celá logika knihovny, stačí jen třídy potřebné pro správné zpracování a generování kódu. V Joogar2 máme pro to balík `Compiler`. Dále

je potřeba v tomto balíku vytvořit třídu, která bude anotační procesor obsahovat. Tato třída musí dědit od abstraktní třídy `javax.annotation.processing.AbstractProcessor` a implementovat všechy její abstraktní metody. V Joogar2 je to třída `JoogarProcessor`.

Dále je potřeba, aby překladač věděl, že toto je náš anotační procesor a aby jej ve správnou chvíli zavolal. Pro tyto účely je potřeba si do závislostí modulu přidat knihovnu `AutoService` [8] od Google. Tato knihovna obsahuje anotaci `@AutoService`, kterou je potřeba anotovat naši třídu obsahující anotační procesor. Jakmile bude knihovna AutoService vědět o umístění anotačního procesoru, vygeneruje a zaregistruje vše potřebné pro správný běh procesoru.

Poslední krok musí udělat vývojář, který používá Joogar knihovnu ve vlastní aplikaci. Je potřeba, aby v gradle souboru v závislostech nastavil modul pro anotační procesor - tedy: `annotationProcessor project(':compiler')`.

5.2.2 Generování kódu pomocí JavaPoet

JavaPoet je knihovna od firmy Square. Tato firma vytvořila již několik knihoven převážně pro Android. Většina těchto knihoven je v komunitě kolem Androidu hojně využívána. I autor sám několik knihoven od této firmy používá. Existuje spousta dalších knihoven na generování kódu, ale JavaPoet je dle autorova názoru momentálně nejpoužívanější.

Generování kódu přichází na scénu v momentě, kdy anotační procesor zpracuje veškeré anotace. V tuhle chvíli už má knihovna přehled o jednotlivých entitách a jejích atributech včetně indexů, primárních klíčů atd. Úkolem generování kódu je v této fázi vygenerovat hlavní třídu `Joogar` s informacemi o jednotlivých entitách.

Do této třídy se vygenerují metody pro jednotlivé databáze. Například pro databázi `joogar` to bude metoda `CreateJoogarDatabase`. V této metodě se namapuje celá databáze včetně entit do pomocných tříd, které budou tuto databázi a její entity reprezentovat. Jako základní třída je `DatabaseJoogar`, které reprezentuje celou databázi. Databázová třída obsahuje dále třídy tabulek, tedy `TableJoogar`. A nakonec třída tabulek obsahuje třídy sloupců `ColumnJoogar`. Všechy tyto třídy obsahují informace nutné k vytvoření kompletní nadefinované databáze. Dále se v této metodě vytvoří proměnná obsahující hash celého kódu, ve kterém se databáze definuje. Tento hash bude dále použit pro rozpoznání změn ve schématu databáze. Na konci metody se informace o schématu celé databáze předá společně s hashem dále knihovně na vytvoření/aktualizaci databáze.

Dále se vytvoří metoda na inicializaci, kterou bude muset uživatel zavolat při spuštění aplikace, a které se předává aplikační kontext. Tato metoda zavolá všechny metody na vytvoření všech databází, které jsme si popisovali v předchozím odstavci.

Po nachystání inicializace a vytváření databáze je na řadě práce s daty. Pro tyto účely se generuje třída `JoogarRecord`. Do této třídy se vygenerují veškeré funkce pro ukládání a vyhledávání dat pro jednotlivé entity - tak jak je to popsáno v kapitole 4.

V této fázi se pro každou entitu generuje třída s postfixem `Table`. Tedy například pro `Car` to bude `CarTable`. Tato třída bude obsahovat veškeré metody pro práci s daty jako `JoogarRecord`, ovšem některé metody rozšiřuje o ukládání, mazání a aktualizaci v transakcích. Pro zjištění počtu záznamů obsahuje metodu `count`. Dále se generuje pomocná třída na vytváření `WHERE` dotazů pro entitu podle namapovaných atributů. Třída je pojmenovaná názvem entity a postfixem `Where`.

5.3 Joogar

`Joogar` je hlavní balíček celé knihovny. Obstará vytváření, aktualizaci a spojení s databází. Balíček obsahuje dva podbalíčky - `Android` a `Shared`. V podbalíčku `Shared` je veškerá logika knihovny a byla psána se snahou vyhnout se platformě závislým komponentám - to znamená v čisté Javě, i když ne vždy se to povedlo. V balíčku `Android` se již nachází platformě závislé třídy pro Android. Například vytvoření a napojení na databázi. Důvod tohoto rozdělení je, že je díky tomu jednodušší možná budoucí rozšířitelnost i na jiné platformy - například na iOS (který také umožňuje používat SQLite databázi) pomocí nástroje J2ObjC [9]. Ovšem toto rozšíření by vyžadovalo ještě značnou práci a možná se již v dnešní době ani nevyplatí.

Hlavní třída tohoto modulu je abstraktní třída `JoogarBase`, od které dědí generátorem vytvářená třída `Joogar`, přes kterou uživatel knihovnu inicializuje. Tato třída si drží reference na všechny databáze. Má je uloženy v hashovací tabulce s klíčem typu `String` a databázi, kterou reprezentuje instance třídy `JoogarDatabase`.

5.3.1 Vytváření databází

Jak již bylo zmíněno, po zavolání inicializace se volají metody na vytvoření databází. Nejdříve si popíšeme, jak knihovna funguje při prvním spuštění aplikace - tedy ve stavu, kdy žádné databáze vytvořené nejsou. Po zavolání metody se na konci vytvoří instance třídy `JoogarDatabaseBuilder`, která obsahuje veškeré informace o databázi. Po vytvoření této instance se volá metoda `addDB` na třídu `Joogar`, které se tato instance předává. V té se již pak na `JoogarDatabseBuilder` volá metoda `build`, kde započne vytváření nové databáze.

V metodě `build` knihovna zjistí, že databáze ještě není vytvořena a požádá pomocí nativní knihovny `Android` o vytvoření databáze. Nyní již máme databázi vytvořenou, ale zatím neobsahuje žádné tabulky. Pro vytvoření jednotlivých tabulek, sloupečků a indexů slouží třída `SchemaGenerator`. Tato třída skládá ze schématu objektové reprezentace databáze, tak jak jsme si vygenerovali, na jednotlivé SQL příkazy pro vytvoření tohoto schématu v databázi a ty dále provádí. Postupuje metodou rozděl a panuj. Tedy nejdřív iteruje přes všechny entity. Pro každou entitu si nachystá SQL řetězec na vytvoření tabulky (`CREATE TABLE ...`). V každé entitě se pak dále iteruje přes jednotlivé atributy, pro které se vytváří další SQL řetězce pro vytvoření sloupečků, včetně všech jejich vlastností. Po nachystání tabulek a sloupců ještě třída vytvoří SQL příkazy na všechny definované indexy. Jakmile jsou všechny SQL příkazy nachystány, začnou se provádět nad vytvořenou databází. Po dokončení provádění těchto příkazů je nová databáze vytvořena a připravena na použití.

5.3.2 Aktualizace schématu databáze

V předchozí podkapitole 5.3.1 jsme se zabývali vytvářením nových databází při prvním spuštění aplikace. Ovšem při vývoji aplikace se dosti často stává, že nová verze aplikace potřebuje upravit schéma té staré. V první řadě je potřeba při spuštění aplikace zjistit, jestli se liší schéma databáze aktuální aplikace od té staré.

První metodou je si z databáze při každém spuštění aplikace vytáhnout schéma a to porovnávat s tím v aplikaci. To by ale zbytečně zdržovalo, každé spuštění aplikace. Další možností je, že uživatel vždy při změně schématu nastaví vyšší verzi databáze. Na to ale může uživatel lehce zapomenout a proto to není moc dobré řešení.

A přesně pro tyto účely nám slouží hash schématu databáze, který jsme si generovali při generování kódu viz. podkapitola 5.2.2. Při spuštění aktualizované aplikace se pak

jen porovná nový hash s tím již použitým v aplikaci a až poté na základě shody/neshody se rozhodne co dál. Při implementaci autor řešil otázku kde tyto hashe aktuální verze schématu uchovávat. Nakonec je to tyto účely při inicializaci knihovny vytvořena ještě jedna databáze `joogar_settings`. Od této databáze je uživatel knihovny odstíněn a je použita pouze pro účely knihovny. V této databázi je tabulka které má primární klíč název databáze a další sloupec, který obsahuje aktuální hash dané databáze. Po aktualizování schématu je pak potřeba jen uložit do databáze nový hash.

Aktualizace databáze probíhá dost podobně, jako vytváření nové databáze. Jen se ve třídě `SchemaGenerator` negenerují nové tabulky a sloupečky. Namísto toho se vytáhne aktuální schéma z databáze SQLite pomocí dotazu na tabulku `sqlite_master`. Tato tabulka obsahuje veškeré informace o aktuálním schématu databáze. Dále se porovnávají jednotlivé tabulky a jejich atributy. Momentálně jdou atributy v aktualizaci schématu přidávat a odebírat, ale upravovat například na jiný datový typ povoleno není. Takové úpravy si bude muset uživatel provést sám pomocí uživatelských migrací.

5.3.3 Uživatelské migrace

V tuto chvíli už máme vyřešeny aktualizace schématu. Může se ale stát, že bude uživatel potřebovat provést nějaké složitější úpravy nebo přidat nějaká data a aktualizace schématu mu už nebude stačit.

Pro tyto případy jsme si v podkapitole 4.2 nadefinovali uživatelské migrace. Metoda, která je zprostředkovává, se generuje společně s metodou na vytvoření databáze. Té se předává pole textových hodnot. Chce-li uživatel přidat další migraci, jen přidá další hodnotu na konec pole. Aplikace si musí pamatovat, který index v poli migrací provedla naposledy. Při spuštění aplikace pak zjistí velikost pole s migracemi a porovná jej s indexem, který má uložený. Pokud knihovna zjistí, že přibyly nějaké nové migrace, tak je provede a uloží si nový index.

Při implementaci zase vyvstala otázka, kde informaci o posledním provedeném indexu uložit. Ale naštěstí už máme vytvořenou databázi `joogar_settings`, kde akorát přidáme k názvu databáze a hash hodnotě ještě poslední sloupeček a to bude index uživatelské migrace.

Chapter 6

Testování

Každá správná knihovna zveřejněná pro open-source komunitu by měla obsahovat i jednotkové testy pro kontrolu, že funguje správně. Usnadní to práci nejen autorovi, ale i celé komunitě. Může se stát, že nějaký programátor z komunity bude chtít přispět úpravou do knihovny. Po úpravě si může spustit sadu testů a zkontrolovat, zda vše funguje správně.

Existuje několik testovacích knihoven pro Android. Autorovi se však nejvíce zalíbila knihovna `AndroidJUnit4` [7]. Nachází se přímo v SDK a díky tomu není potřeba definovat další závislosti. Testy se v naší knihovně nachází v balíčku `net.skoumal.joogar2`.

6.1 Příprava na testování

Pro každý test je potřeba si připravit prostředí, to znamená inicializovat knihovnu a po dokončení testů databázi zase smazat. Pro to je vytvořena abstraktní třída CrudTestCase, která má dvě metody:

before Tato metoda je spuštěna na začátku každého testu. Nejprve si získá aplikační kontext a dále pomocí tohoto kontextu inicializuje knihovnu. Tato metoda je označena JUnit anotací `@Before`.

after Tato metoda je volána po skončení testu. Projde všechny vytvořené databáze a smaže je. Tato metoda je označena JUnit anotací `@After`.

Všechny testovací případy/třídy budou dědit od třídy `CrudTestCase`.

6.2 Základní testy

Tyto testy se nachází v balíčku `net.skoumal.joogar2.test.basic`. Testy pracují s entitami nadefinovanými v balíčku `net.skoumal.joogar2.util.model`. V modelech jsou nadefinované entity se všemi základními datovými typy, se kterými knihovna pracuje.

Většina těchto testů testuje čtení a zápis daného datového typu, a to včetně prázdných hodnot. Dále se v těchto testech nachází test na použití dědičnosti při definování entit.

6.3 Pokročilé testy

Pokročilé testy se nacházejí v balíčku `net.skoumal.joogar2.test.basic`. Tyto testují knihovnu ze všech možných pohledů. Zde je jejich výčet:

BulkSaveTests Testy na hromadné ukládání hodnot.

ConstraintsTests Testy na integritní omezení.

DatabaseTests Testy na základní operace s daty.

FindStatementTests Testy na metodu find - offset a řazení.

JoogarTests Testy na inicializaci knihovny.

TransactionTests Testy na funkčnost transakcí.

WalTests Testy na write-ahead lock.

WhereStatementTests Testy na správné vytváření WHERE dotazů pomocí metody where.

Chapter 7

Ukázková aplikace

Jako ukázkovou aplikaci autor zvolil jednoduchý seznam úkolů. Aplikace umožňuje přidávání a mazání úkolů. Přidávání štítků. Dále pak řazení úkolů dle jednotlivých atributů. A filtrování úkolů dle přiřazených štítků.

7.1 Databázový návrh

Aplikace má jednu databázi a dvě tabulky viz. obrázek 7.1.

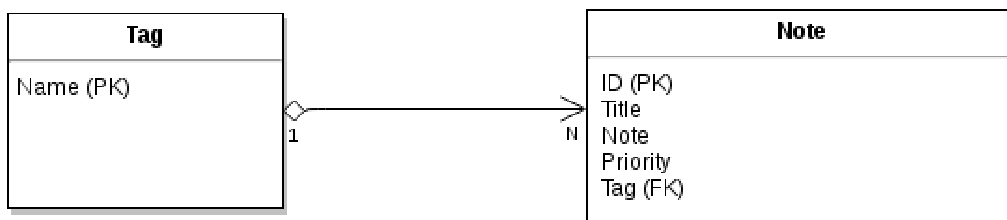


Figure 7.1: Návrh databáze pro ukázkovou aplikaci

Záznam v tabulce Note obsahuje veškeré informace úkolu. Tabulka Tag pak obsahuje jednotlivé štítky, které se úkolu přiřazují.

7.2 Rozhraní aplikace

Aplikace se skládá z jedné hlavní obrazovky. Na hlavní obrazovce je menu, vyjíždějící z levé strany. Dále aplikace obsahuje dva dialogy na vytvoření nového úkolu a štítku.

Hlavní obrazovka obsahuje jednotlivé úkoly. Tyto úkolu jdou označit jako dokončené dvojím přetažením na jakoukoliv stranu. Po kliknutí na tlačítko „+“ v pravém dolním rohu, otevře se dialog na přidání nového úkolu. Popisované obrazovky jsou k vidění na obrázku 7.2.

Pokud uživatel na hlavní obrazovce přejede prstem z levého kraje obrazovky směrem doprostřed vyjede mu menu aplikace. Toto menu je rozděleno na tři části. V první se nastavuje pořadí jednotlivých úkolů dle atributů. V druhé části se nastavuje filtrování

pomocí jednotlivých štítků. A ve třetí části se nachází tlačítko, které po kliku otevírá dialog na přidání nového štítku. Obě dvě obrazovky jsou k vidění na obrázku 7.3.

7.2.1 Práce s databází

Inicializace databáze probíhá ve třídě `ClientApp`. To se pouští jako první při spuštění aplikace. Po inicializaci se spouští hlavní aktivita `MainActivity`. Z té se dá dostat na dva dialogy - `NewNoteFragmentDialog` a `NewTagFragmentDialog`. V této aktivitě a dvou dialozích jsou následující

MainActivity.getNotes() Tato metoda si přečte z třídních atributů dle čeho má řadit data a jestli se mají filtrovat podle štítku. Dle těchto údajů následně provede dotaz na databázi a výsledek zobrazí v seznamu.

MainActivity.deleteNote(Note note) Metoda na smazání úkolu předaného v parametru metody.

MainActivity.initTags() Tato metoda načte všechny štítky z tabulky Tag a zobrazí je k filtrování v menu.

NewNoteFragmentDialog.prepareTag() Tato metoda načte všechny štítky a připraví je do komponenty Spinner, ze které se bude vybírat štítek pro přidávaný úkol.

NewNoteFragmentDialog.saveNote() Metoda uloží nový úkol a skryje dialog na přidávání úkolů.

NewTagFragmentDialog.saveNote() Metoda uloží nový štítek a skryje dialog na přidávání štítků.

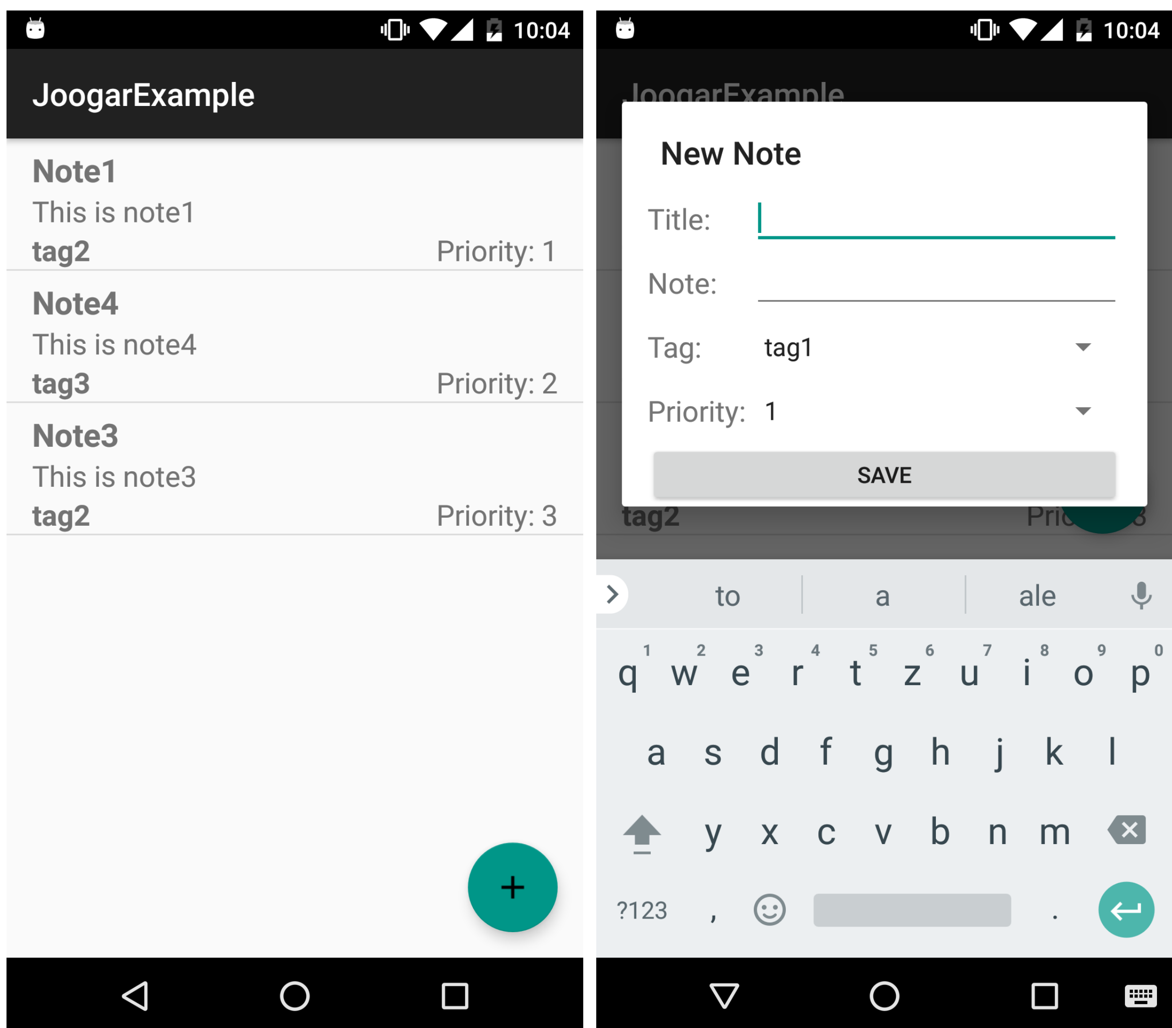


Figure 7.2: Hlavní obrazovka a přidání úkolu v ukázkové aplikaci

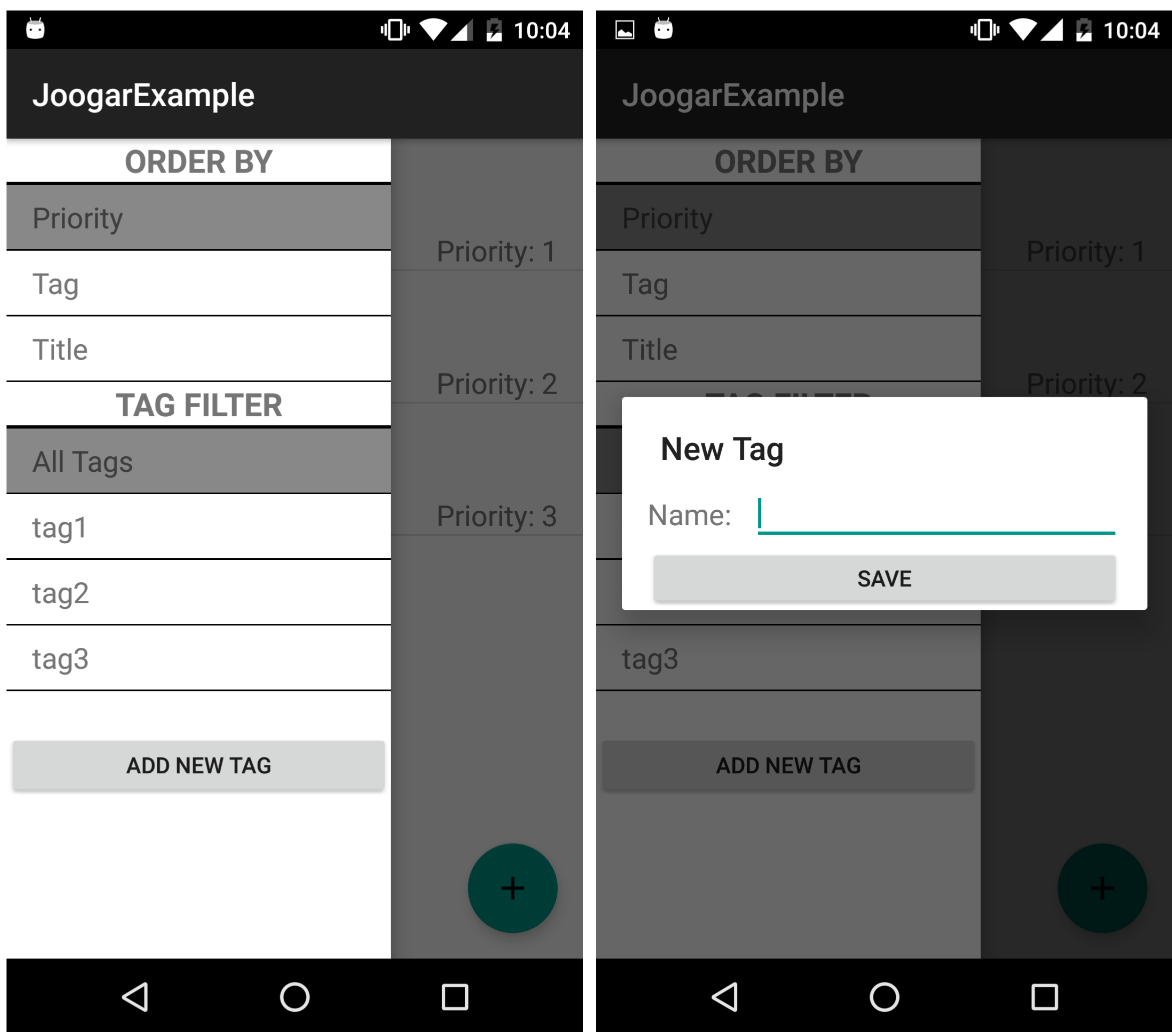


Figure 7.3: Hlavní obrazovka s menu a přidání štítku v ukázkové aplikaci

Chapter 8

Závěr

Cílem bakalářské práce bylo vytvoření ORM knihovny pro Android. V první fázi bylo potřeba analyzovat aktuálně používané ORM knihovny a zaměřit se zejména na jejich nedostatky. V další fázi autor vytvářel návrh rozhraní naší ORM knihovny. Při vytváření návrhu bylo potřeba se zaměřit na nedostatky analyzovaných ORM knihoven a pro ty následně navrhnout řešení.

Následně proběhla implementace knihovny. Při implementaci byl dbán důraz na rychlost knihovny a čitelnost kódu. Předpokládá se, že po zveřejnění knihovny na server Github se přidají další vývojáři a pomohou tuto knihovnu dále rozšiřovat. Knihovna byla během implementace řádně testována pomocí jednotkových a integračních testů.

Výsledná knihovna Joogar2 nabízí efektivní a jednoduchou práci s databází na systému Android. K pohodlné práci napomáhá i generování metod podporující jednoduché vyhledávání dat dle zadaných kritérií. Při porovnání výkonnosti s ostatními knihovnami vychází knihovna Joogar2 jako jedna z nejrychlejších [4]. Knihovna je připravena pro používání při vývoji mobilních aplikací.

Autor této práce má v plánu dále knihovnu rozvíjet. Jako první zveřejní knihovnu pro komunitu na serveru Github. Dále uloží knihovnu na jcenter a umožní tak linkování knihovny do projektu pomocí závislostech v souboru gradle.

Knihovně nyní chybí podpora relací, které si zatím musí uživatel řešit sám. Nicméně v další fázi vývoje autor tuto podporu doplní, včetně podpory `JOIN` v metodách na vytváření dotazů. Pokud bude mít komunita o knihovnu zájem, autor by rád knihovnu prezentoval na některé z konferencí o vývoji pro mobilní zařízení.

Bibliography

[1] Aditya, S. K.; Karn, V. K.: *Android SQLite Essentials*. Packt Publishing. 2014. ISBN 1783282959.

[2] Android: *Android Developers*. [online]. [cit. 20.4.2017].
Retrieved from: https://developer.android.com/index.html

[3] Bosley, S.; Koren, J.: *SquiDB*. [online]. [cit. 21.4.2017].
Retrieved from: https://github.com/yahoo/squidb/

[4] Bína, J.: *Android knihovna pro objektově-relační mapování*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. 2017.

[5] Dorfmann, H.: *Annotation Processing 101*. [online]. 2015. [cit. 25.4.2017].
Retrieved from:
http://hannesdorfmann.com/annotation-processing/annotationprocessing101

[6] Friesen, J.: *Learn Java for Android Development*. Berkley: APress. 2010. ISBN 9781430257226.

[7] Gamma, E.; Beck, K.; et al.: *JUnit*. [online]. [cit. 25.4.2017].
Retrieved from: http://junit.org/junit4/

[8] Google Inc.; aj.: *AutoService*. [online]. 2017. [cit. 24.4.2017].
Retrieved from: https://github.com/google/auto/tree/master/service

[9] Google Inc.; aj.: *J2ObjC*. [online]. 2017. [cit. 23.4.2017].
Retrieved from: https://github.com/google/j2objc

[10] Gray, J.; Reuter, A.: *Transaction processing*. San Francisco: Morgan Kaufmann Publishers. c1993. ISBN 1558601902.

[11] Narayan, S.: *Sugar ORM*. [online]. [cit. 21.4.2017].
Retrieved from: http://satyan.github.io/sugar/index.html

[12] Roebuck, K.: *Object-Relational Mapping: High-impact Strategies - What You Need to Know*. Lightning Source. 2011. ISBN 9781743044759.

[13] Vávrů, J.; Ujbányai, M.: *Programujeme pro Android*. Praha: Grada. second edition. 2013. ISBN 9788024748634.

[14] Wharton, J.; Wilson, J.; et al.: *Javapoet*. [online]. 2017. [cit. 22.4.2017].
Retrieved from: https://github.com/square/javapoet