Database storage

Structured Query Language

Example project

Structured Query Language

- we will have a live coding session a bit later
- checkout repo:
  git@github.com:ikust/infinumacademy-phonebook.git

SQL

Structured Query Language

- relational databases
- data insert, query, update and delete
- schema creation and modification
- data access control

SQL commands

Structured Query Language

- data definition:
  CREATE TABLE, DROP TABLE
- data manipulation:
  SELECT, INSERT, UPDATE, DELETE
- SQL tutorial:
  http://www.w3schools.com/sql/default.asp
- available functions in SQLite:
  https://www.sqlite.org/lang.html

SQL vs SQLite

Structured Query Language

- SQLite = stripped down SQL
- SQLite works directy with files, no database server (no deamons running in the background)
- https://www.sqlite.org/docs.html

Transactions

Structured Query Language

- executing more statements atomically
- everything succeeds or everything fails

SQLite

Using it on Android

Implementing database helper

SQLite

- create a new class that extends **SQLiteOpenHelper**
- more details: http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html
- manages database creation and version management
- only **one** instance of helper should be used

Implementing database helper

SQLite

- database name and version passed through constructor

```
public PhoneBookDatabaseHelper(Context context) {
    super(context, "PhoneBookDatabase", null, 1);
}
```

Implementing database helper

SQLite

- database should be created in **onCreate()** method
- called only once, first time database is accessed

```
@Override
public void onCreate(SQLiteDatabase db) {
    //SQL commands for creating the database first time it is accessed.
    db.execSQL(
            "CREATE TABLE PhoneBook ("
                    + "id INTEGER PRIMARY KEY, "
                    + "firstName TEXT, "
                    + "lastName TEXT, "
                    + "phone TEXT, "
                    + "mail TEXT)"
    );
}
```

Implementing database helper

SQLite

- database migration code goes in **onUpgrade()** method

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    //Database migration.
}
```

Reading data

SQLite

- get database instance and call **query()** method:

  query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)

- or use **rawQuery(String sql, String[] selectionArgs)** method and pass SQL statement

- for more details check: http://developer.android.com/reference/android/database/sqlite/

Adding, updating and deleting data

SQLite

- Adding - **insert()** method
- Updating - **update()** method
- Deleting - **delete()** method
- or use **execSql(String sql)** method and pass SQL statement

Extracting values from cursor

SQLite

- methods that add data return Cursor object
- use **moveToNext()** to move along table rows
- use **getString(int)**, **getLong(int)**, … to get value from column
- use **getColumnIndex(String)** to get index of column with given name

```
while(cursor.moveToNext()) {
    Contact contact = new Contact();
    contact.setFirstName(cursor.getString(cursor.getColumnIndex("firstName")));
...

}
```

Storing values to ContentValues

SQLite

```
ContentValues contentValues = new ContentValues();
contentValues.put("firstName", contact.getFirstName());
contentValues.put("lastName", contact.getLastName());
contentValues.put("phone", contact.getNumber());
contentValues.put("mail", contact.getMail());
```

SQLite

Coding session

Object Relational Mapping

ORM

Object Relational Mapping

- “magic” that maps your Java object into database tables

Android ORM libraries

Object Relational Mapping

- https://github.com/Raizlabs/DBFlow
- https://realm.io/news/realm-for-android/
- active android
- ORM lite

DBFlow

Android ORM library

DBFlow

Android ORM library

- open source library, developed by Andrew Grosner (Raizlabs)
- current stable version is 2.2.1
- available at https://github.com/Raizlabs/DBFlow

Including it in your project

DBFlow

- in project level **build.gradle** add:

```
buildscript {
    repositories {
        jcenter()
    }

    repositories {
        maven { url "https://raw.github.com/Raizlabs/maven-releases/master/reelases"}
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.3'
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.4'
    }
}

allprojects {
    repositories {
        jcenter()
        maven { url "https://raw.github.com/Raizlabs/maven-releases/master/reelases"}
    }
}
```

Including it in your project

DBFlow

- in app level **build.gradle** add:

```
apply plugin: 'com.android.application'
apply plugin: ‘com.neenbedankt.android-apt'

dependencies {
    apt 'com.raizlabs.android:DBFlow-Compiler:2.2.1'
    compile 'com.raizlabs.android:DBFlow-Core:2.2.1'
    compile 'com.raizlabs.android:DBFlow:2.2.1'
}
```

Including it in your project

DBFlow

- extend **Application** class
- initialize / destroy FlowManager

```java
public class PhoneBookApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        FlowManager.init(this);
    }

    @Override
    public void onTerminate() {
        super.onTerminate();
        FlowManager.destroy();
    }
}
```

Database configuration

DBFlow

- annotate with **@Database**
- more than one database can be defined

```java
@Database(name = PhoneBookDatabase.NAME, version = PhoneBookDatabase.VERSION)
public class PhoneBookDatabase {

    public static final String NAME = "PhoneBookDatabase";

    public static final int VERSION = 1;
}
```

Model creation

DBFlow

- model defines a table in a given database

```
@Table(databaseName = PhoneBookDatabase.NAME)
public class Contact extends BaseModel implements Serializable {

    @PrimaryKey (autoincrement = true)
    @Column
    private long id;

    @Column(name = "firstName")
    private String firstName;

}
```

Model creation rules

DBFlow

- All Models **MUST HAVE A DEFAULT CONSTRUCTOR**
- **Subclassing** works as one would expect: the library gathers **all inherited fields** annotated with **@Column** and count those as rows in the current class's database.
- **Column names default** to the **field name** as a convenience
- All **fields** must be **public** or **package private** as the ModelAdapter class needs access to them.
- All model **class definitions** must be **top-level** (in their own file) and **public** or **package private**

Reading, adding, updating and deleting

DBFlow

- SQL statement wrapper methods:

```
new Select().from(Contact.class).queryList();
```

- BaseModel methods for adding, updating and deleting

  save()

  update()

  delete()

Migrations

DBFlow

- SQL statement wrapper methods:

```
@Migration(version = 2, databaseName = PhoneBookDatabase.NAME)
public class SomeMigration extends BaseMigration {

    @Override
    public void migrate(SQLiteDatabase sqLiteDatabase) {

    }
}
```

- documentation:
  https://github.com/Raizlabs/DBFlow/blob/master/usage/Migrations.md

DBFlow

Coding session

Transactions

DBFlow

- for those who want more
- documentation: https://github.com/Raizlabs/DBFlow/blob/master/usage/Transactions.md

Zadaća

Database storage

Nadograditi aplikaciju iz prošle zadaće:
 - dodati registraciju korisnika
 - podatke dohvaćene preko REST API-ja cacheirati u bazu.

U slučaju da internet veza nije dostupna, ako je korisnik ulogiran mora moći vidjeti podatke o brodovima i detaljima. Slike brodova nije potrebno cacheirati, već prikazati placeholder sliku.

Zadaća - za one koji žele više

Database storage

Implementirati cacheiranje slika brodova.