

BACHELOR PAPER

Term paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Computer Science

Analysis of different data-storage technologies within Android applications

By: Florian Mötz

Student Number: 1610257017

Supervisor: Dipl.-Ing. Dr.techn. Thomas Polzer

Vienna, January 26, 2018



Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, January 26, 2018

Signature

Kurzfassung

Die Verwendung von mobilen Applikationen steigt stetig und erschließt dabei immer neue Anwendungsfelder. Dadurch ergeben sich neue Anforderungen, wie, z.B., ein steigender Bandbreitebedarf für die Übertragung der Applikationsdaten beziehungsweise die ununterbrochene Verfügbarkeit der Daten. Auch wenn das mobile Internet immer schneller wird und mittlerweile in westlichen Ländern beinahe flächendeckend zur Verfügung steht, gibt es immer wieder Situationen, bei denen die Daten lokal auf den Geräten gespeichert werden müssen. Das Android Framework bietet nativ mehrere Möglichkeiten an, Daten auf dem Gerät zu persistieren, es existiert aber außerdem eine Vielzahl an Open Source Libraries, um Daten lokal zu speichern. In dieser Arbeit wird ein Überblick über eine ausgewählte Anzahl von Technologien und Frameworks gegeben, welche zur Persistierung von Daten in Android Applikationen angewendet werden können. Diese erstrecken sich von SharedPreferences über SQLite und Android Room bis hin zu Cloud-Datenbanken.

Schlagworte: Android, Applikation, Datenspeicherung, SharedPreferences, SQLite, Android Room, Realm, Cloud Datenbanken

Abstract

Mobile application usage is rising rapidly. This leads to new challenges, like the requirement for increased data transmission bandwidth or an uninterrupted data availability. Although the mobile network is becoming faster and more comprehensive, situations exist, where data needs to be persisted on the devices. The android framework offers multiple options to store data on the device out of the box, but in the developer community additionally a variety of open source libraries to persist data is available. This paper will give an overview on a selected number of technologies and frameworks, which can be used to persist data in Android applications, such as SharedPreferences, Android Room or Cloud-Databases.

Keywords: Android, Application, data storage, SharedPreferences, SQLite, Android Room, Realm, Cloud Databases

Acknowledgements

I would like to thank Thomas Polzer for his supervision and his ever-present advice, and everyone who supported me during this thesis.

I also thank my colleagues at *Tailored Apps* for their shared knowledge and help in any matter. Furthermore, I would like to thank my family and friends for their help, support and understanding during the study.

Contents

1	Introduction	1
1.1	Problem Definition and Research Questions	3
1.2	Overview	4
2	Background	4
2.1	Types of data	5
2.1.1	Flags	5
2.1.2	Primitive data types	5
2.1.3	Strings	6
2.1.4	Collection of primitive types	6
2.1.5	Lists of primitive types	6
2.1.6	Array of primitive types	6
2.1.7	Structured data types	6
2.1.8	Binary Data	7
3	Data storage techniques	7
3.1	Key-Value storage	8
3.1.1	Suitability	8
3.1.2	Implementation in Android (SharedPreferences)	8
3.2	File-based storage	11
3.2.1	Suitability	11
3.2.2	Implementation in Android	12
3.3	SQL-based storage	16
3.3.1	General	16
3.3.2	Suitability	18
3.3.3	Implementation in Android	18
3.4	Third Party data storage libraries	26
3.4.1	General	26
3.4.2	Suitability	26
3.4.3	Implementation in Android	26
3.5	Cloud-based database	27
3.5.1	General	27
3.5.2	Suitability	27
3.5.3	Services	28

4 Findings	29
4.1 Comparison of technologies	29
4.1.1 Limits of technologies	29
4.1.2 Usability of the Frameworks	30
4.2 Data Security	31
4.2.1 Rooted Devices	31
4.2.2 Encryptability	32
5 Conclusion	32
6 Future Work	32
Bibliography	33
List of Figures	39
List of Tables	40
List of Listings	41
List of Abbreviations	42
A Comparison of Internal and External storage	43

1 Introduction

According to a study from *eMarketer* [72], in 2014 and 2015 the smartphone usage reached the highest value ever seen (see Figure 1). The study further predicted that it will grow year over year, reaching 2.87 billion in 2020. With the rise of smartphones, mobile applications have become more important, as the number of apps available in the *Google Play Store* and the *Apple App Store* show. In more detail, in April 2011 200.000 apps were available in the Play Store and in June 2010 425.000 in the App Store, compared to 3.300.000 available apps in September 2017 in the Play Store and 2.200.000 in January 2017 in the App Store [73] [74] (see Figures 2 and 3). As these application stores are open for any developer to publish applications (over 900.000 app publisher on the Google Play Store, nearly 500.000 publishers in Apple's App Store, and 75.000 publisher in the Amazon App Store [35]), a large variety of apps is available and smartphones can and are used for various tasks.

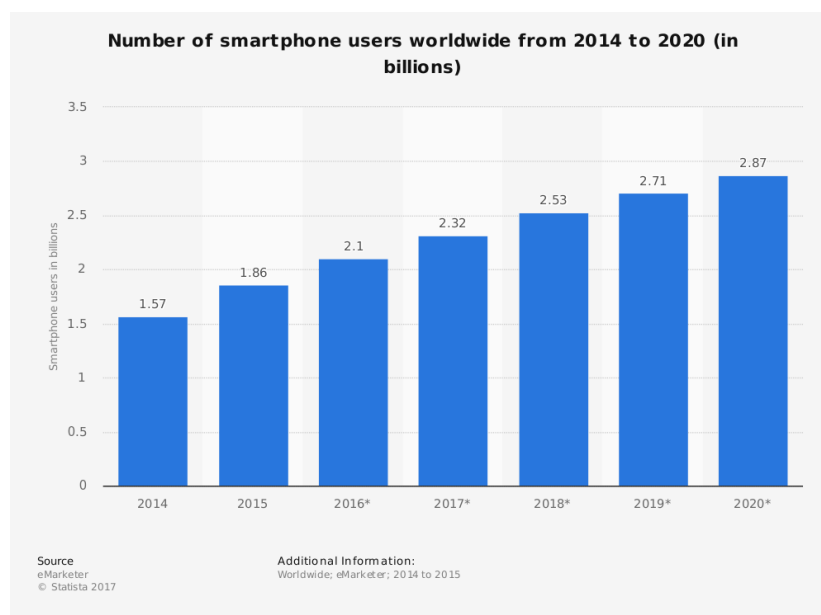


Figure 1: Number of smartphone users worldwide from 2014 to 2020 (in billions) [72]

Today a large percentage of mobile applications is connected to the internet, online services and/or databases. Although the mobile network connectivity has evolved with the rise of smartphones, which lead to applications being able to load data at any given time, mobile applications show trends towards caching more data locally to provide a fluent user experience without content loading times. [79] The data caching minimizes the required network bandwidth and maximizes the battery lifetime. [26]

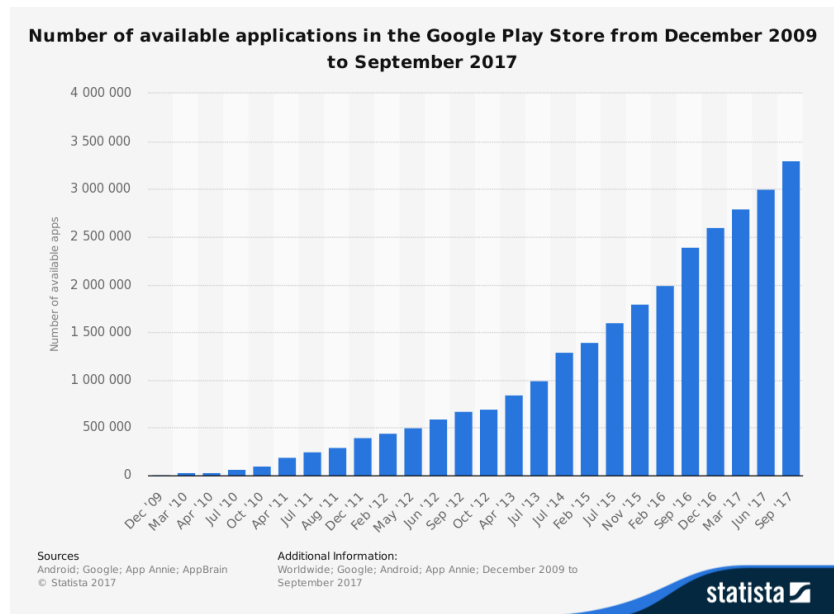


Figure 2: Number of available applications in the Google Play Store from December 2009 to September 2017 [73]

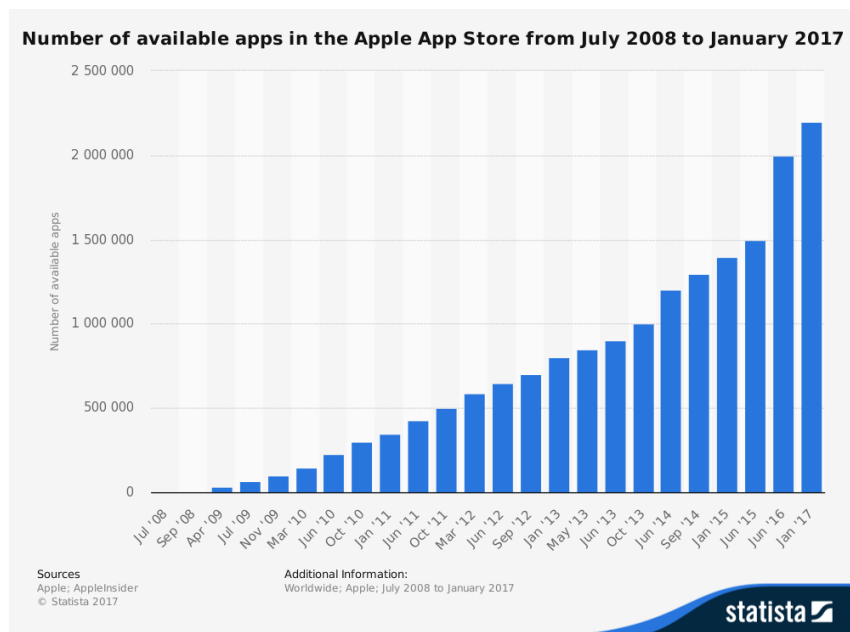


Figure 3: Number of available apps in the Apple App Store from July 2008 to January 2017 [74]

For example, in a real world scenario, a network request will take at least 100 ms in a fourth generation broadband cellular network (4G network), and at least 200 ms in a third generation broadband cellular network (3G network), if a DNS lookup as well as a TCP and TLS handshake are required. [46] Table 1 [46] summarizes the details. If, for example, an application needs to perform multiple requests for displaying a simple user profile page, the user will have to wait an unreasonable amount of time for the data to be displayed, if the data is not stored locally. The result will be a bad user experience. As caching and storing data locally is of high importance for a well designed mobile application, efficient and performant storage technologies are required. This thesis will therefore evaluate several storage options for the Android platform and create a guideline when and how to use which storage mechanism.

	3G	4G
Control plane	200 – 2.500 ms	50–100 ms
DNS lookup	200 ms	100 ms
TCP handshake	200 ms	100 ms
TLS handshake	200 – 400 ms	100 – 200 ms
HTTP request	200 ms	100 ms
Total latency overhead	200 – 3500 ms	100 – 600 ms

Table 1: Latency overhead of a single HTTP request [46]

Note, that for total latency overhead the control plane is not taken into consideration, as this overhead is not caused by HTTP or the transmission itself, but the underlying layer of connectivity management in UMTS networks.

1.1 Problem Definition and Research Questions

Depending on the type and amount of data, different technologies may be used to store data on a mobile device. This thesis will introduce a selection of different storage technologies available for developers to developers of Android applications. We will focus on built-in or easy to use third party options. The selected third party libraries are either promoted by Android itself or highly popular with other developers (which is shown by the number of stars on the project's *GitHub* repository, number of articles on *medium.com* or questions on *stackoverflow.com*) [15]. Furthermore, the suitability of the different technologies and libraries regarding the different types of data to be stored will be analysed and presented. Moreover, the implementation cost as well as security aspects of the stored data will be explored.

1.2 Overview

This thesis will list different techniques and frameworks for data storage in Android and will outline the limits of these technologies as well as their advantages and drawbacks. Furthermore, a comparison of the techniques will be performed, but as different applications and use cases have different mobile storage requirements, no globally optimum exists, and instead a comparison of advantages and drawbacks will be executed and presented.

2 Background

Mobile Applications are a major part of today's lives. A study published by *Flurry* [64] suggests that US consumers spend 5 hours per day using their mobile devices (see Figure 4).

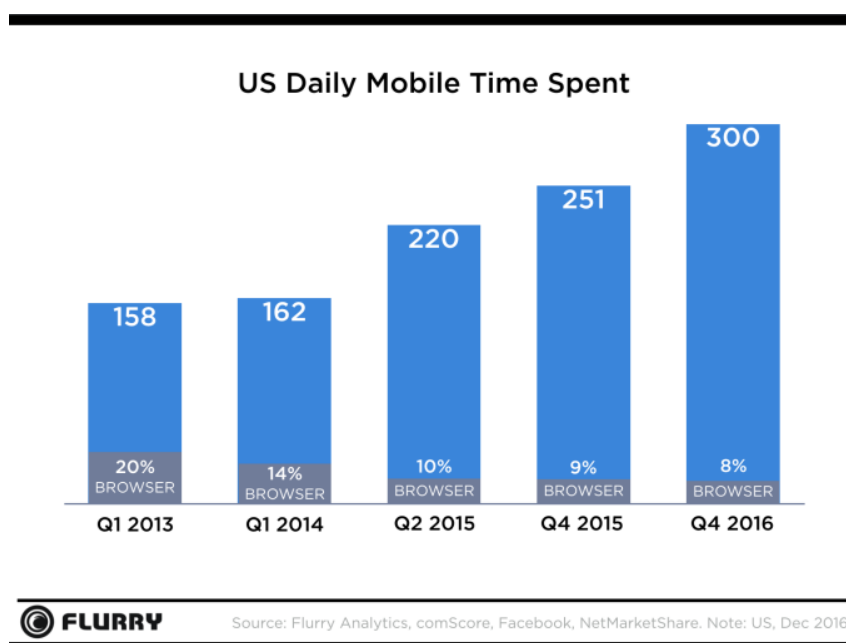


Figure 4: US Daily Mobile Time Spent [64]

Smartphones are ever-present and provide any type of information, entertainment or communication the user desires. [40] In order to keep application states, user data, fetched content or any other type of data persistently stored on the device after closing an application, a specialized, local storage system is required. Furthermore, if the smartphone loses its connection to the internet, a modern application should continue working with the data that was already loaded, or even provide pre-fetched data to stay useful.

In order to store data locally on the device, multiple techniques are available, from simple *Key-Value storage* or file-based storage to NoSQL- or SQL-based databases. [15] [61] [51] Each of

these different technologies provides advantages, but comes with its own drawbacks, which have to be considered, depending on the use case. Moreover, the implementation of the data storage technique in an Android Application should be easy for developers, as their main focus should be on creating high level applications, and not on low level data storage technologies.

2.1 Types of data

Data items as simple as a single flag encoding the state of a user preference toggle, e.g., up to images or files with megabytes or even gigabytes of data, may be needed to be stored on a device. This chapter summarizes the classes of data which will be considered in this thesis. The presented items were selected according to the possible data types to be stored in the default Key-Value-Storage of Android (see Section 3.1) as well as datatypes which may be stored in SQLite databases (see Section 3.3).

2.1.1 Flags

In simple cases, storing yes or no *flag* is sufficient to encode an application state value or in order to configure how to present the application output. In high level programming languages used to develop Android applications, such as Java or Kotlin, this is usually represented by the data type `boolean` [56] [48], whereas in low level programming languages like C flags are often stored as single bits in *integers*, where a single 32-bit `integer` may encode up to 32 single flags. [53] [56]

2.1.2 Primitive data types

If a single yes/no value is not enough to encode the data, any of the primitive data types defined in the used programming languages may need to be stored. In this context, the following data types are considered to be *primitive* and therefore will be discussed in this section. Please note that this list only contains the basic implementation of the data type and not all subtype definitions (e.g. different signedness, different bit length) are discussed in detail.

- **Integer:** An Integer represents a single integral number with limited range. Depending on the programming language and the compile or runtime environment, its range may differ. Furthermore, most languages support an unsigned version of this type. In Java, the default `int` data type can store a signed 32 bit value, which ranges from -2^{31} to $2^{31} - 1$. Normally additionally integral number types with different range are available (e.g. `byte`, `short` or `long`) [56].
- **Float:** A Float, or floating point number, representing a non integral numeric data type capable of representing decimal values. The range and precision is dependent on the programming language, compiler, runtime environment and the selected precision format

(single or double). In Java, *double* is the floating point data type with the highest precision, and can store a double precision floating point value as defined in IEEE 754 [56].

- **Char:** The char data type represents a single character, which — depending on the programming language, compiler and runtime environment — may only include ASCII characters or may be encoded in a more sophisticated format, such as for example UTF-8 or UTF-16. In Java, the default `char` data type can store a single 16 bit Unicode character [56].

2.1.3 Strings

The String data type can hold an array of characters. The type of each single element is dependent on the basic character type used by the programming language (e.g. ASCII, UTF-8, ...) [57]. In many programming languages the basic string is indeed implemented as a basic array of characters. As Strings are used quite frequently, modern languages, however, contain a specialized and optimized String implementation. [56] [58]

2.1.4 Collection of primitive types

Additionally to single primitive values, a collection of primitive values can be used by programs to maintain their internal data. There are many ways to organize these data collections, the most frequently used variants are lists, arrays and maps. [55]

2.1.5 Lists of primitive types

A list is a mutable or immutable collection of an unknown number of primitive elements of the same kind. [59]

2.1.6 Array of primitive types

In contrast to lists, *arrays* may only contain a predefined, defined number of elements. [52] As Android programs are based on Java (or more generic a Java Virtual Machine (JVM) implementation), only *mutable* arrays are accessible in the default Array implementation in Android development (although immutable arrays are possible using the `Collections` class). [52] [54] Therefore only mutable arrays are considered.

2.1.7 Structured data types

Structured data types have a predefined, fixed structure of primitive or other structured data types. Most of the data used in mobile applications, or higher order programming languages in

general, heavily use structured data types, as data that needs to be stored is too complex to fit into a simple primitive type.

In Android, such a structure may be defined in a PDS (*Passive data structure*, which is a Java object only containing fields (and getter/setter methods), without any internal logic [36]). In case of Kotlin, the specialized `data classes` may be used [47].

As many web-services use the structured JSON as their preferred data exchange format [39], structured data must often be handled when caching service data on the device. Furthermore, data stored on a relational database also falls into this category of data, as these databases can store relations to other tables which need to be mapped to complex data structures.

2.1.8 Binary Data

In some cases, there might also be the need to store a blob of binary data [16], e.g. in case of caching an image locally. This type of data will be categorized as *binary data*, as the contents of the binary blob is irrelevant for the storage system. [70]

3 Data storage techniques

After discussing the most important types of application data, we will now present different options to persist this data locally on a device. There are several techniques known to store. The storage options which will be discussed in this thesis, are:

- Key-Value storage
- Relational databases
- NoSQL databases
- File-based storage
- Cloud-based storage

In this chapter, each approach will be discussed in detail and its data storage and retrieval options will be presented. An analysis on which of the above listed storage options is best suited for which type of data (from Section 2.1) will be performed. For each of the storage options, possible Android implementations will be presented.

3.1 Key-Value storage

The most basic data storage technique is based on key-value pairs. For each unique key, a corresponding value is set. The key is used to retrieve the value again. Therefore, from the point of view of an app developer, this concept is similar to a normal variable, but the value is accessed by calling a method with the given key instead of only using the variable name.

As the technique and implementation of this storage systems is rather simple compared to other ones, it is often the simplest and fastest form of storage. [68]

3.1.1 Suitability

As stated above, this technique is most suitable for single variables and supports fast data storage and retrieval. Implementations often only provide methods to store primitive data types (as defined in Section 2.1.2), sometimes also sets or arrays of simple types. [31] To store complex data structures two possibilities exist. The first option is to separate the complex data structure into multiple primitive fields, which can be stored using keys. The second option is to serialize the given data structure into a string and then store it as a single String value (in Android applications e.g. using the `Serializable` class [33] or the `GSON` library [41]). However, storing complex structures may not be the best option for this type of storage, as either serializing (and de-serializing) or storing lots of single key-value pairs may not be very performant any may be failure prone.

3.1.2 Implementation in Android (SharedPreferences)

Simple, persistent Key-Value Storage is provided by the Android framework out of the box. The corresponding mechanism is called `SharedPreferences`. [31] A *shared preferences* instance is a set of key-value pairs, which is stored in a file that is either accessible privately or shared between applications, depending on the selected access mode (see Section *Mode* for more details). This means that the `SharedPreferences` can be used as simple, private datastorage within an application, as well as a shared storage between multiple applications. [31] Please note, that since Android Marshmallow the access to a shared preferences object from multiple applications is no longer supported. [11]

Accessing SharedPreferences

To work with `SharedPreferences`, a `Context` is needed. A `Context` is an object provided by the Android framework which represents the current state of the application, activity or service. Furthermore, a `Context` can be used to resolve resourced and retrieve services or information about the application environment. [10]

Any `Context` will provide the following methods to manage shared preference containers [11]:

- `deleteSharedPreferences(String name)`
Deletes the given shared preferences container file.
- `getSharedPreferences(String name, int mode)`
Retrieves the contents of a shared preference container with the given name, which can later on be used to retrieve, edit or remove values. If no shared preference file with the given name exists, a new one will be created when the first value is written (and committed) (see Section *Modifying data*).
- `moveSharedPreferencesFrom(Context sourceContext, String name)`
Move the contents of the given shared preference container from another `Context` to this one. This is usually only required, if stored data should be migrated to protected storage on the device [34] [11].

Mode

Shared Preferences may be accessed in different modes, which define the accessibility of the preferences to other activities or applications. The mode can be defined by using one of the following predefined flags with the `getSharedPreferences` method: [11]

- `MODE_PRIVATE`: The preferences-file may only be accessed by the calling application or applications which share the same user-ID.
- `MODE_WORLD_READABLE`: The preferences may be accessed in read-mode by any application on the device. Note that this option has been deprecated and will cause a `SecurityException` since Android *Nougat* (API-Level 24).
- `MODE_WORLD_WRITEABLE` The preferences may be accessed (both read and write-access) by any application on the device. Note that this option has been deprecated for the same reasons as `MODE_WORLD_READABLE`.
- `MODE_MULTI_PROCESS`: The preference-file on the disk will be checked for modification, even if the instance is already loaded in the calling process. This is needed, if more than one process is accessing the preferences (this was default behaviour prior Android *Gingerbread*, API-Level 9). Note that this flag is deprecated and applications should no longer use it.

PreferenceManager

To simplify the access to the default `SharedPreferences` instance of the application, Android introduced the `PreferenceManager` [20]. This class provides the following static methods to access the default shared preferences:

- `getDefaultSharedPreferences(Context context)`
Returns an implementation of `SharedPreferences` pointing at the default shared preferences file (whose location is returned by the following method)
- `getDefaultSharedPreferencesName(Context context)`
Returns the name of the default preferences, (default implementations will return a path similar to `/data/data/[applicationpackagename]/shared_prefs/[applicationpackagename]_preferences.xml`, where *[application package name]* is the name of the applications package name)

Accessing data

To access data from an instance of *SharedPreferences*, getter-methods are available for all primitive data types. They require the key of the queried *SharedPreference* and a default value, which will be returned in the case that the given key is not set. [13] The following methods are defined: `getString`, `getInt`, `getLong`, `getFloat`, `getBoolean`. Furthermore, additionally to the primitive types, a set of Strings may be retrieved by calling `getStringSet`. To check whether a given key is set in the preferences or not, the method `contains` can be used. Noteworthy is, that instances of *SharedPreferences* are *Singletons* within a process. [13]

Modifying data

In contrast to querying data, which can be done directly on the *SharedPreferences* instance, editing data is more sophisticated. First a so called *Editor* object must be created by calling `edit()` on the *SharedPreferences* instance.

The *Editor* interface provides the following methods to put key-value-pairs into the preferences: `putString`, `putStringSet`, `putInt`, `putLong`, `putFloat` and `putBoolean`, each with the first parameter being the key as `String` and the second the value of the corresponding type. Furthermore, the method `remove` removes a preference with the given key, and `clear` removes all preferences in this instance. Note that removals (calls to `clear` and `remove`) are always done first when committing or applying an *Editor*, regardless of the order of invocation on the *Editor*-object, whereby `put`-calls will still be done on the preference-file, after clearing or removing values. [13]

It is important to note that changes made to an editor of a *SharedPreferences*-object are not copied back to the file immediately, the changes have to be submitted by one of the following methods:

- `commit`: Commits any changes done in this *Editor* to the underlying file and awaits the completion of this process. It is not recommended to use this method, if the success operation and the timing of the write operation is not crucial (developers should use `apply` instead). [14]

- `apply`: Commits any changes made in the `Editor` to the underlying file asynchronously. This means that errors, which occurred, will not be recognized, but since errors are not likely to happen, it is recommended to use this method. Furthermore, the framework makes sure, that *life-cycle events*, such as shutting down an `Activity`, will be deferred until all `apply`-calls have completed (either with error or success). [14] `commit`-calls, which are invoked after some calls to `apply`, but before the last one has completed, will block until all of them have finished.

Usage Example Listing 1 shows how to retrieve the default, private instance of `SharedPreferences` (line 1), how to retrieve a value and specify a default value (line 2), how to open an `Editor` (line 3), modify a value in the preferences (line 4) and applying the changes (line 5). In this example, calling `getBoolean("showDetails", false)` on the `sharedPreferences`-object between lines 4 and 5 would still return the original value, and not the newly set one (`true`), as the changes in the editor have not been committed yet. Also calling the method right after `apply` will most likely (but not for sure) still return the old value, as applied changes are executed asynchronously on a background thread.

```
1 SharedPreferences sharedPreferences =
    PreferenceManager.getDefaultSharedPreferences(context);
2 sharedPreferences.getBoolean("showDetails", false);
3 SharedPreferences.Editor editor = sharedPreferences.edit();
4 editor.putBoolean("showDetails", true);
5 editor.apply();
```

Listing 1: Example: `SharedPreferences`

3.2 File-based storage

To store more than simple key-value pairs of primitive types, developers can either choose a database (SQL (see Section 3.3) or NoSQL) or classic files. While it is often easier to store and use data — especially well structured data (a list of a structured datatype, e.g.), which can be mapped to a table — within a database, large or unstructured data, pictures, music files or videos, e.g., is most commonly saved as separate files. Furthermore, storing such data as files on the SD card of a smartphone, e.g., enables to sharing files between applications or to transfer them to a personal computer.

3.2.1 Suitability

As files are the most basic and unrestricted way to store data, this storage technique can be used to store any type and any amount of data, as only a stream of bytes is passed to write into the file. [25]

3.2.2 Implementation in Android

As Android is mostly written in `Java` - or languages running in a JVM - the Java-framework within the `java.io` package for basic file operations is inherited. Furthermore, as Android devices run on a Linux kernel, the file system is based on the Linux file system. On top of the basic file system, the Android framework differs the *internal* and *external* storage location:

Internal and external storage

Android basically distinguishes two different sections of file storage in terms of permissions (and former location), the *internal storage* and the *external storage*. [12] These labels come from times, when Android devices contained internal memory as well as a removable storage medium (an SD card, e.g.), which in newer smartphones is becoming less popular. [1] Nowadays, as there are devices without the possibility to insert an SD card, these devices often provide an *internal* and *external* storage visible to the developer or application, but both are being physically stored in the internal memory [75]. Table 2 in Appendix A outlines the basic properties of the two types. [12]

The most important difference between the two storage locations is, that the internal storage is not accessible for other applications, which means that sensitive information may be stored here (except for rooted devices, see Section 4.2.1). Furthermore, the user has to grant applications permission to read from or write on the external storage, as this permission allows full access to all public files stored on the user's device.

Basic API

File locations in Android are represented by `File` objects, which are basically representing the absolute path to a directory or a file. [19] The `File`-class offers basic file or directory related methods, such as create or delete files or directories, check whether a file or directory exists or renaming of a file or directory. [19]

In order to read or write a file, the basic Java *Streaming* methods can be used. [12] Listing 2 shows a sample usage of creating a new directory and creating a file with a given content in it.

```

1 //Get the internal storage directory
2 File filesDirectory = context.getFilesDir();
3 //add a directory to the path (this does not create a new directory)
4 File directory = new File(filesDirectory, "newDirectory");
5 //add the filename to the path (this does not create a file)
6 File file = new File(directory, "test.txt");
7 //mkdirs creates all directories in the given path which are not created yet
8 if(!file.mkdirs()) {
9     throw new RuntimeException("Directories_could_not_be_created");
10 }
11 //Create a new FileOutputStream to the given file (this creates the file)
12 try (FileOutputStream fos = new FileOutputStream(file)) {
13     //write some content in it
14     fos.write("Hello_World!".getBytes());
15     fos.flush();
16     fos.close();
17 } catch (FileNotFoundException e) {
18     e.printStackTrace();
19 } catch (IOException e) {
20     e.printStackTrace();
21 }

```

Listing 2: Example: Create a directory and write to a file

Internal storage

To access the internal storage, the `Context` class provides two methods to get a `File` object to the desired path: [12]

- `getFilesDir()`: Returns a `File` object with the absolute path to the applications internal storage directory. This directory can be seen as the 'root' directory for the applications private storage. As the absolute path to this directory may change (although the contents stay the same), the returned absolute directory should not be persisted [11]. The path will most likely be of the following form: `/data/data/[packagename]/`, where `[packagename]` is the package name of the application.
- `getCacheDir()`: Returns a `File` object with the absolute path to a cache directory private to the calling application, which can be used to cache files with "a reasonable size limit for the amount of memory you use at any given time, such as 1MB" [12]. Files in the cache directory may, however, be deleted at any time without warning, if the system requires memory.

External storage

As described above, the external storage can either not be mounted or the application can not be allowed to read from or write to the external storage, which is why the following states need

to be checked before accessing it: [12]

External storage for write access If the application wants to write on the external storage, the following checks need to be done:

- The state of the external storage needs to be checked, i.e. whether the external storage is mounted or not. This check can be done using the static `getExternalStorageState` method of the `Environment` class, as shown in Listing 3.
- Furthermore, the permission, whether the application may write on the external storage `android.permission.WRITE_EXTERNAL_STORAGE` needs to be checked, and if not granted, requested from the user. Listing 4 shows an example of requesting a permission at runtime. If the permission has not been granted yet, the dialog to request the permission which is prompted to the user by calling the `requestPermissions` method is shown asynchronously, and the result of the request needs to be handled in the `onRequestPermissionsResult` Callback method, which needs to be implemented within the activity, as shown in Listing 5.

```
1 String state = Environment.getExternalStorageState();
2 if (Environment.MEDIA_MOUNTED.equals(state)) {
3     //External storage is mounted
4 }
```

Listing 3: Check if the external storage is mounted [12]

```
1 if (ContextCompat.checkSelfPermission(activity,
    Manifest.permission.WRITE_EXTERNAL_STORAGE) !=
    PackageManager.PERMISSION_GRANTED) {
2     // Request the permission.
3     ActivityCompat.requestPermissions(
4         activity,
5         new String[] { Manifest.permission.WRITE_EXTERNAL_STORAGE },
6         REQUEST_CODE
7     );
8 } else {
9     // Permission already granted
10 }
```

Listing 4: Request a permission at runtime (Minimal working example) [27]

```

1 @Override
2 public void onRequestPermissionsResult(int requestCode, String permissions[],
   int[] grantResults) {
3     switch (requestCode) {
4         case REQUEST_CODE: {
5             if (grantResults.length > 0 && grantResults[0] ==
               PackageManager.PERMISSION_GRANTED) {
6                 // permission was granted
7             } else {
8                 // permission denied
9             }
10        }
11    }
12 }

```

Listing 5: Asynchronous Callback for permission requests at runtime within an Activity (Minimal working example) [27]

External storage for read access If the application wants to read from the external storage, the following checks need to be done: [12]

- The state of the external storage needs to be checked, i.e. whether the external storage is mounted or is mounted in read-only mode. This check can be done using the static `getExternalStorageState` method of the `Environment` class, as shown in Listing 6.
- Furthermore, the permission of the user, whether the application may write to external storage (`android.permission.READ_EXTERNAL_STORAGE`) needs to be checked, and if not granted, requested from the user, as shown above in Listings 4 and 5. If the user has granted permission to write to external storage, the permission to read from external storage has been granted implicitly.

```

1 String state = Environment.getExternalStorageState();
2 if (Environment.MEDIA_MOUNTED.equals(state) ||
3     Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
4     //External storage is mounted (at least for read-access)
5 }

```

Listing 6: Check if external storage is mounted or mounted in read-only mode [12]

Categories of files to write to external storage As the external storage represents the whole public storage of the device, the android framework provides static methods to request the absolute path for predefined folders for different categories of files.

Basically, the files to store in external storage can be split into two categories: [12]

- **Public files:** Any file, that should be available for the user and for other apps, even if the application has been uninstalled, which includes pictures, videos or music files, e.g.
- **Private files:** Any file, which belongs directly to the application and should be removed if the application is uninstalled. These files may be visible to the user, but do not need to be registered by other applications (although they are able to read or alter those files).

To store public files, the android framework provides the static method `getExternalStoragePublicDirectory` in the `Environment` class, which returns a `File` object representing the appropriate path to store a file categorized by the given type (constants defined in the `Environment`-class): [18]

- ***DIRECTORY_MUSIC:*** Audio files which should appear for the user in the list of music
- ***DIRECTORY_PODCASTS:*** Audio files which should appear for the user as podcasts
- ***DIRECTORY_RINGTONES:*** Audio files which can be selected by the user as ring tone
- ***DIRECTORY_ALARMS:*** Audio files which can be selected by the user for alarm sounds
- ***DIRECTORY_NOTIFICATIONS:*** Audio files which can be selected by the user for notification sounds
- ***DIRECTORY_PICTURES:*** Picture files which should be displayed in e.g. the gallery
- ***DIRECTORY_MOVIES:*** Movies, which should be available to the user
- ***DIRECTORY_DOWNLOADS:*** Any files downloaded by the user
- ***DIRECTORY_DCIM:*** Picture or video files which should be found, if the device is mounted as camera
- ***DIRECTORY_DOCUMENTS:*** Any documents created by the user

To store files in external storage which are not intended for any of the above listed use cases, the `getExternalFilesDir` with `null` as parameter will return a directory of a folder in the external storage for this purpose. Although files stored in this directory are intended to not be shared with other applications, they are visible and accessible to them. The advantage of storing files in this folder is that they will be removed if the application is uninstalled.

3.3 SQL-based storage

3.3.1 General

To store more data, or data with a more complex structures, developers often choose SQL-based data storage techniques with relational databases.

Relational databases

Relational databases are a simple and yet powerful way to store and operate on complex data-structures with relations between objects. They were introduced in 1970 to overcome the limitations of file-based storage systems, which was the state of the art way to store data. [76]

SQL

To interact with a powerful and complex data-storage mechanism, like a relational database, a common, standardized language to perform interactions is needed. SQL is the most common — and de facto standard — language to perform operations (like querying, updating or deleting) on different database management systems. [63]

SQL is a standard (former ANSI SQL, now SQL-99, [63]), which defines features which need to be available in all SQL implementations and outlines the basic concepts of commands and their syntax. [63] This enables vendors of database systems to use the SQL language in their product and combine commands for their special feature set with the standard SQL operations in an unified and interoperable way. This leads to a generic language, designed to be simple for small queries, as well as to be able to perform complex instructions not depending on the database management system, as well as using specific database management system features within the SQL language.

SQLite

The Android platform provides built-in support for *SQLite*, which is “an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine”. [69] SQLite provides all standard SQL commands, but as it is very lightweight, lacks some advanced features. *Adam Stroud*, author of *Android Database Best Practices* states:

“It does not include all the features of more robust systems, but it does provide a large enough feature set to handle most use cases that arise in mobile development.” [76]

While the official *SQLite* documentation states 15 different limitations within their database [71], *Stroud* states only two differences between SQLite and other SQL-based systems, which developers are likely to face even in not too complex databases, as may be used in mobile applications [76]:

- SQLite only provides limited implementations for the `JOIN` operator [71], which is used to create queries using multiple tables at the same time connecting their datasets
- SQLite provides only a small set of datatypes: [70]
 - `INTEGER`: A signed integer, with 1, 2, 3, 4, 6 or 8 bytes of data, depending on the value stored (which may not be defined by the user as it is common in other

SQL based storage systems, where different types of integers (e.g. `SMALLINT` or `BIGINT`) may be defined for a field)

- `REAL`: A floating-point value, stored within 8 bytes
- `TEXT`: A text string, stored with the database encoding
- `BLOB`: A blob of data

Stroud summarizes SQLite that to be “a very popular database system primarily because it has a really small footprint and still provides many of the features of much larger database systems”, where the term *footprint* refers to the size of the library, not the functionality. [76]

3.3.2 Suitability

Saving data in an SQLite database is most suitable for any type of relational data with a defined structure. Examples include contact information, user data or general lists of content loaded from a server or entered by the user. As the data in the database may be retrieved or altered easily using SQL with simple or complex queries, it is fast and easy to access the data. Despite the fact that SQLite is quite performant, the data is stored in a single file underneath. Furthermore, it is supported out of the box by the Android framework, and only requires minimal setup [29].

3.3.3 Implementation in Android

The Android framework provides the basic SQLite classes in the `android.database.sqlite` package, in which, for a single database, only the `SQLiteOpenHelper` (see below) needs to be extended. While these APIs are usable for any feature and quite powerful, they are “*fairly low-level and require a great deal of time and effort to use*” [29] because of following reasons: [29]

- The raw SQL queries are not verified on compilation, which might lead to errors at runtime
- A lot of boilerplate-code is needed to convert the data from SQL query results to Java objects

Therefore, Google introduced the *Room* library within the *Android Architecture Components* in Summer 2017. This is a mapping library on top of the basic SQLite classes, to minimize boilerplate code and add compile time SQL checks [30].

SQLiteOpenHelper

The Android framework provides an abstract class `SQLiteOpenHelper`, which encapsulates the low-level SQLite operations.

As stated above, using this class, SQL statements still need to be written by the developer, and the results need to be converted from cursors to Java objects (if not working with the database cursors itself) but this class eases the use of SQLite in Android significantly.

To create a database using the *SQLiteOpenHelper*, the class needs to implement a constructor and override the `onCreate` and `onUpgrade` methods: [23]

- **Constructor:** As the abstract class only defines one, non-default constructor, this constructor needs to be called in any of the constructors in sub-classes: `SQLiteOpenHelper(Context context, String name, CursorFactory factory, int version)`
 - `context`: A context to open or create the database (most likely passed to the sub-class constructor as well)
 - `name`: The name of the database file
 - `factory`: A *CursorFactory* to use for creating cursors for this database, may be null to use the default cursor factory
 - `version`: The version number of the database schema. Each time the database structure changes, this version number should be increased. Depending on the database version that is currently stored on the disk, and the version passed in the constructor, the `onUpgrade` (or `onDowngrade`) methods will be invoked to alter the database from the version currently stored on the disk to the version given here.
- `onCreate`: Will be invoked when the database is created for the first time, which means that no tables are present in the database. Therefore, this method should be used to execute SQL statements to create desired tables.
- `onUpgrade`: Will be invoked when the database version has changed (the version passed in the constructor is higher than the version currently stored on the disk) with the *oldVersion* and *newVersion* as parameters, to alter the tables to match the schema for the given *newVersion*. The most basic implementation would be i.e. to delete all tables and create the tables again according to the new database schema.

A minimum working example of such a database helper class inheriting from the `SQLiteOpenHelper` class is shown in Listing 7.

The `SQLiteOpenHelper` class provides the methods `getReadableDatabase` and `getWritableDatabase` to retrieve an instance of `SQLiteDatabase`, on which the database queries can be performed (where the readable database supports only read-operations, while the writable database supports both read and write operations).

Inserting values To insert values into a table, the `insert` method with `ContentValues` may be used as shown in Listing 8 (which uses the database created in Listing 7). The `insert`

```

1 public class NewsDbHelper extends SQLiteOpenHelper {
2
3     public static final int DATABASE_VERSION = 1;
4     public static final String DATABASE_NAME = "News.db";
5
6     public NewsDbHelper(Context context) {
7         super(context, DATABASE_NAME, null, DATABASE_VERSION);
8     }
9     public void onCreate(SQLiteDatabase db) {
10         db.execSQL("CREATE_TABLE_news_(id_INTEGER_PRIMARY_KEY,_title_TEXT,_
11             content_TEXT)");
12     }
13     public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
14         db.execSQL("DROP_TABLE_IF_EXISTS_news");
15         onCreate(db);
16     }
17 }

```

Listing 7: Example implementation of the SQLiteOpenHelper class

method requires three parameters: The name of the table to insert into, an optional *nullColumnHack*, which is can any column name of the table, if the content-values are empty (as SQL does not allow to insert an empty row without having at least one column name in the query), and the content-values [22].

```

1 SQLiteDatabase db = newsDbHelper.getWritableDatabase();
2
3 // The keys represent the column names, the values the value to insert
4 ContentValues values = new ContentValues();
5 values.put("title", title);
6 values.put("content", content);
7
8 //returns the primary key of the inserted row
9 long rowId = db.insert("news", null, values);

```

Listing 8: Example insert operation

Retrieving values To read from the database, the `SQLiteDatabase` class provides a generic `query` method, which requires the following parameters: [22]

- **table:** The name of the table to fetch data from.
- **columns:** A string-array of the names of columns actually needed in the result-cursor, as only these columns will be fetched for the result. This value may be null, which indicates to load all columns of the table.

- `selection`: A string representing the *SQL WHERE* clause (to be appended after the *WHERE* keyword in an SQL query) to filter the result. Passing null will return all values. To prevent SQL injections, the parameter value should not be encoded into this String but replaced with a question mark ('?') and the value should be passed in the `selectionArgs` array (see below).
- `selectionArgs`: An array of values that are marked as '?' in the `selection` argument, to be replaced escaped to prevent SQL injections.
- `groupBy`: If the result should be grouped, this argument should contain the *GROUP BY* clause of the SQL query.
- `having`: The SQL *HAVING* clause, if grouping is used and the having-clause is needed.
- `orderBy`: An SQL *ORDER BY* clause, if the result should be ordered.

Listing 9 shows a sample select query (on the database created in listing 7).

Deleting values To delete values from a table, the `SQLiteDatabase` class provides the `delete` method, which takes three parameters: The name of the table, the condition clause for the deletion and the arguments for the deletion clause. Listing 10 shows an example to delete an item (in the database created in Listing 7).

```

1 SQLiteDatabase db = newsDbHelper.getReadableDatabase();
2
3 //define which columns are needed in the result
4 String[] projection = { "id", "title", "content" };
5
6 // Result should be filtered by title 'Android'
7 String selection = "title_=?";
8 String[] selectionArgs = { "Android" };
9
10 // results should be ordered by title descending
11 String sortOrder = "title_DESC";
12
13 Cursor cursor = db.query(
14     "news",
15     projection,
16     selection,
17     selectionArgs,
18     null,    //no grouping
19     null,    //no 'having' as no grouping is done
20     sortOrder
21 );
22
23 List<News> newsList = new ArrayList<News>();
24 while(cursor.moveToNext()) {
25     News newsItem = new News(
26         cursor.getLong(cursor.getColumnIndexOrThrow("id")),
27         cursor.getString(cursor.getColumnIndexOrThrow("title")),
28         cursor.getString(cursor.getColumnIndexOrThrow("content"))
29     );
30     newsList.add(newsItem);
31 }
32 cursor.close();

```

Listing 9: Example retrieve operation

```

1 SQLiteDatabase db = newsDbHelper.getWritableDatabase();
2
3 String selection = "title_LIKE_?";
4 String[] selectionArgs = { "Android" };
5 db.delete("news", selection, selectionArgs);

```

Listing 10: Example delete operation

Android Room

Room is an android framework introduced at the *Google I / O 2017* as part of the *Android Architecture Components* [49], and provides an abstraction layer on top of the Android SQLite framework to ease development of the database access by reducing boilerplate code and adding compile-time SQL code validation.

Room introduces three kinds of components, which are needed to persist data in an SQLite database. Figure 5 shows their relations, and the following list describes its individual components. [30]

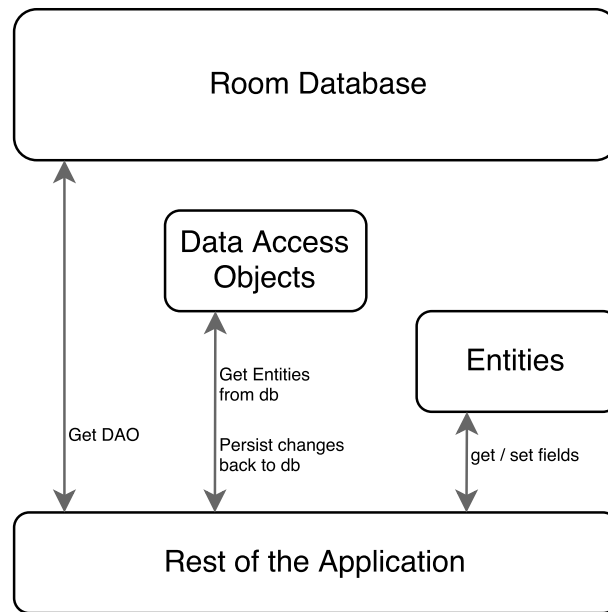


Figure 5: Room architecture diagram [30]

- **Entity:** Any Object which needs to be persisted in the database will be represented by a PDS Java class annotated with `@Entity`. Furthermore, a primary key may be set with the `@PrimaryKey` annotation, and a column name should be set using the `@ColumnInfo` annotation on the field. An example entity is shown in Listing 11.
- **Data Access Object (DAO):** A DAO is an interface defining all methods to interact with the database. The interface needs to be annotated with `@Dao`, and any method in the interface may be annotated with the following annotations:
 - `@Query` [21]: Takes a string as parameter defining the query to execute when invoking the method. The query supports SELECT, UPDATE and DELETE statements.
If the query is a SELECT query, the framework will automatically map the returned fields into the given return-type object of the method by their field-names, or if multiple rows are returned, the return-type of the method may be a `List` or `Array` of objects. If the query contains UPDATE or DELETE statements, the return type of the method may be `void` or `int` (which will be set to the number of affected rows).

If at least one column name of the result of the query matches any field of the return type, the framework will fill the matching field in the object (which does not necessarily need to be an object annotated with `@Entity`, although it is recommended) and will print a `CURSOR_MISMATCH` warning, if a result-field did not match with any of the fields in the object.

To pass parameters into the query, the method defined in the interface may take any number of arguments, which may be mapped into the query using a colon and the name of the parameter in the method. A basic example can be seen at the `findByTitle` method in Listing 12. To create e.g. a SQL `IN` condition, where a list of values needs to be passed onto the query, a method parameter may also be a list or an array. This is shown at the `loadByIds` method in Listing 12.

- `@Update` [24]: To update an entity, a method in the DAO interface annotated with `@Update` with the entity to update as parameter may be defined. To update multiple entities at once, a list of entities may also be passed as arguments. The method does not return any value, so it should be marked as `void`.

The `@Update` annotation further accepts one parameter `onConflict`, which defines what should happen on conflicts while updating the entity (this may be one of the constants defined in the `OnConflictStrategy` class: `ABORT`, `FAIL`, `IGNORE`, `REPLACE`, `ROLLBACK`; and defaults to `ABORT`).

- `@Delete`: To delete entities, a method annotated with `@Delete` and any number of entities (or lists of entities) as parameters may be defined in the DAO interface. The methods should declare the return type `void`. [17]

- **Database:** A database is an abstract class, which will combine the different entities and data access objects into a database. This class needs to be abstract, extend from the `RoomDatabase` class, and annotated with `@Database`. Within the `@Database` annotation, all entities within the database need to be listed, as well as a version number for the database schema needs to be defined. [30] The class then needs to create abstract methods for any data access object, which will access the database. An example database class is shown in Listing 13.

As *Room* is just an abstraction layer on top of the default implementation of SQLite in the Android framework, the framework basically generates an implementation of the *SQLiteOpenHelper* class from the annotations and queries, with bindings to the `Entities`, which can be inspected of the developer in the generated classes folder. [28] Furthermore, if runtime errors occur, the error will be thrown in the generated code within the *SQLiteOpenHelper* implementation, which enables the developer to better understand the error and why it did occur.

```

1@Entity
2public class User {
3
4    @PrimaryKey
5    private int id;
6
7    @ColumnInfo(name = "title")
8    private String title;
9
10   @ColumnInfo(name = "content")
11   private String content;
12
13   //Getters and setters for each field are required by Room, but left out here
14 }

```

Listing 11: Room Example entity class

```

1@Dao
2public interface NewsDao {
3
4    @Query("SELECT_*_FROM_news")
5    List<News> getAll();
6
7    @Query("SELECT_*_FROM_news_WHERE_title_LIKE_:searchInput")
8    News findByTitle(String searchInput);
9
10   @Query("SELECT_*_FROM_news_WHERE_id_IN_( :newsIds)")
11   List<News> loadByIds(int[] newsIds);
12
13   @Insert
14   void insert(News... news);
15
16   @Delete
17   void delete(News news);
18 }

```

Listing 12: Room Example Data Access Object class

```

1@Database(entities = {News.class}, version = 1)
2public abstract class NewsDatabase extends RoomDatabase {
3    public abstract DaoNews daoNews();
4 }

```

Listing 13: Room Example database class

3.4 Third Party data storage libraries

3.4.1 General

As Android provides SQLite out of the box, it provides no other form of relational database in the framework. But as in the Android community a great number of libraries are developed and open sourced, also many third party libraries for relational databases are provided.

In *CodePath's 'Must-Have Libraries'*, which lists libraries which “*are extremely popular and are often used in almost any Android project*” [37], 19 different persistence libraries are mentioned, including *Realm*, *Room* and a large number of Object-relational mapping tools (ORMapper).

3.4.2 Suitability

The different libraries, which implement a custom database, often target a more specific use case, in which they offer a much better performance or ease the implementation of data storage for the developer.

3.4.3 Implementation in Android

Realm

The *Realm Database* is an open source database for iOS and Android, which is not based on SQLite, but implements a custom database engine. *Realm* claims to be “*much faster than an ORM, and often faster than raw SQLite*” [61]. *Realm's* database is implemented using the Android Native Development Kit (*Android NDK*) [62], which allows to run native *C++* code on the device, which is faster and more performant than pure Java code.

ObjectBox

ObjectBox is a an object oriented database which is categorized as NoSQL database. It is “*optimized for performance and designed to save app developers from dealing with SQL*” [51]. While *ObjectBox* is a fully transactional database with all ACID (*Atomicity, Consistency, Isolation and Durability*) characteristics, it uses no SQL queries. In benchmark tests provided by *ObjectBox* itself [50], *ObjectBox* is the fastest library when executing access, update, insert and delete commands compared to *Realm* and *Android Room*, only loading data takes more time than when using *Realm*, as shown in Figure 6.

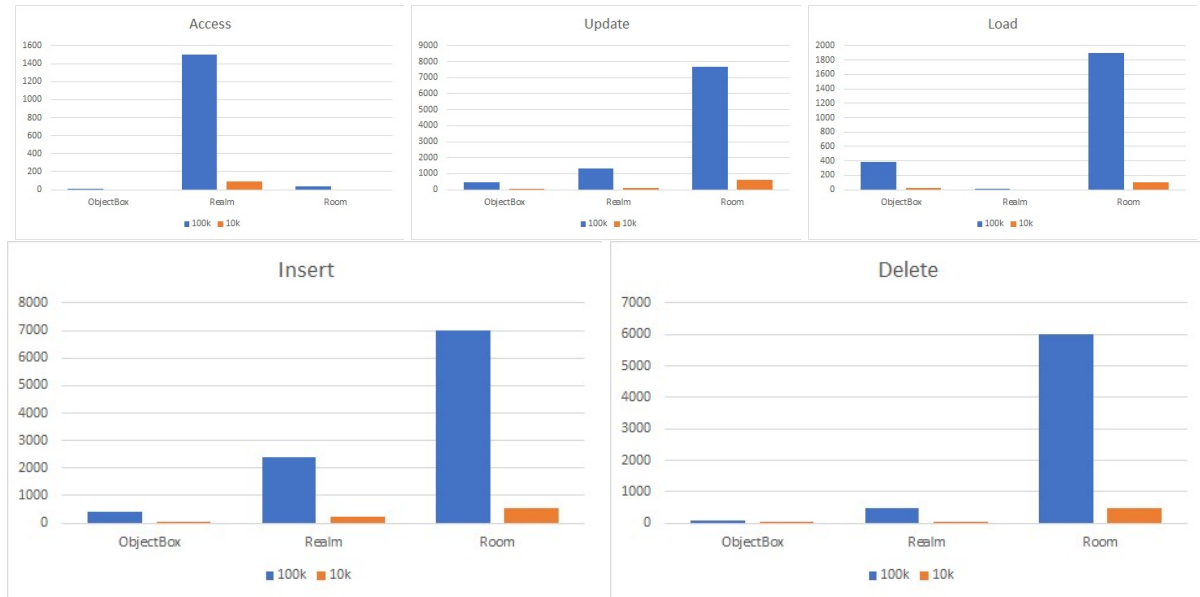


Figure 6: Performance for 100k / 10k elements measured in ms [60]

3.5 Cloud-based database

3.5.1 General

Local data storage is often used to store data retrieved from an online service to access the data faster and even without internet connection. As such online services (often referred as *backends*) also need to be developed, and especially for smaller applications often only reflect online databases, several cloud providers offer solutions called Mobile Backend as a Service (MBaaS). [38] These services provide web services to access custom databases hosted by the cloud provider, but often also provide frameworks for Android or other mobile platforms, which fetch and store the data from the service in the background. Therefore the developer can access the data without caring about the network, connectivity problems or other *edge cases* which are associated with network connections and data synchronization. [42]

3.5.2 Suitability

Such online databases, which automatically get synced with the devices, are generally best suitable for small structures of data to minimize network traffic. Furthermore, as the implementation of such online databases with local persistence is sometimes easier and faster than implementing content fetching and local database caching, this technology may also lead to shorter implementation times. Moreover, the whole synchronization logic, all possible fallbacks and edge-case scenarios are handled by the framework, which eases the implementation for the developer and makes the whole system more reliable. The types of data which can be stored depends on the used service. While nearly all considered frameworks provide the pos-

sibility to store key-value pairs of primitive data types, some of the frameworks are also able to store complete java objects, or such objects converted to maps, e.g. as done in *Firebase*. [44]

3.5.3 Services

MBaaS are often provided by services, which already provide cloud solutions, and run on different — free and paid — plans. [77] Two solutions of large, well known cloud providers are presented. There are numerous other solutions but the two selected ones give a good overview of the storage feature-set provided by the services.

Firebase Realtime Database

Firebase is a development platform for mobile as well as web applications, which was acquired by Google in 2014. Besides a number of services for mobile applications, it provides the *Firebase Realtime Database* service.

Firebase provides, besides the MBaaS, a number of client frameworks for different platforms to ease the access to the data, including iOS, Android, JavaScript, C++, and Unity. [42] The *Firebase Realtime Database* is basically a Key-Value storage, where the value can either be primitive Java types as well as Java objects, as the values are stored as JavaScript Object Notation (JSON) format, in which java objects may be marshalled easily. [45] The values are cached locally (if configured), and changes can be made while the device is not connected to the internet and are persisted and committed after the device has internet connectivity again [43].

AWS Mobile

Amazon Web Services (AWS) is a cloud service, that provides *AWS Mobile*, which is an MBaaS service, and frameworks for Android and iOS as well as web technologies and React Native [8]. The following three services are provided to save data:

- *NoSQL Database Feature*: With this feature an *AWS DynamoDB* instance can easily be accessed within mobile applications. *DynamoDB* is a *NoSQL* database service of AWS for a fast and flexible cloud database. [6] The data is accessed using a DAO class generated by the web-service. This class must be present at compile time. No local caching option is available for this service. [4]
- *User Data Storage*: This feature combines the two AWS features *Amazon Simple Storage Service (Amazon S3)*, which allows to store files on the web service, and *Amazon Cognito Sync*, which enables to sync key/name pairs between an application, the cloud and mobile devices [5]. Therefore, using this service, the application can use the framework provided by the service to save and retrieve complete files on the server, as well as synchronize simple key-value pairs, which are, however, limited to store strings as values [7].

- *Cloud APIs*: This feature eases the calling of *AWS Lambda* functions within client applications. *AWS Lambda* is a service to run code triggered by events in a serverless computing environment. [2] The client framework requires a class containing the available functions at compile time, which is generated by the *AWS Mobile Hub* console. Additionally it provides request-builder classes to simplify the implementation of a *RESTful* API call of a defined *Lambda function*. [3]

4 Findings

After presenting the main storage technologies available to Android developers, we will now compare them in terms of storage size limitations and implementation effort as well as discuss any security implications from using a given storage technology.

4.1 Comparison of technologies

It is important for an Android developer to understand when to utilize which storage technology and why. This section compares all discussed technologies and derives usage scenarios for each one of them.

4.1.1 Limits of technologies

The introduced technologies offer different limits for storing data.

The Shared Preferences, as they can only store single key-value pairs with primitive types or lists or primitive types, are limited in storing complex data structures. They would need to be split into individual key-value pairs or would need to be serialized as strings to be stored, and afterwards de-serialized to retrieve the content again.

File-based storage has the advantage that it can store large amount of data and imposes no limit on the data format. On the downside, however, the data can only be retrieved on a stream or string level and no out of the box query interface exists.

SQL-based storage offers the possibility to define the data structure in detail and to implement relations between data entities which can be enforced by the database system. Additionally, a powerful data manipulation and query language (SQL) exists, simplifying the data access. The standard Android implementations discussed in this thesis, however, have some limitations and drawbacks in terms of code quality and development support:

- *SQLiteOpenHelper* requires to access the database on a quite low level, which has no type-safety or type-binding and requires the developer to write SQL statements, which are not verified at compile time.
- Android Room introduces type-safety, field bindings and verified SQL-statements at compile time, but lacks automatic fetching of referenced relations in queries, which creates another type of coding overhead for the developer.

The main drawback of Realm is that it is a native external library and therefore bundles numerous shared objects (one for each CPU architecture) with the application, increasing the application package size. Furthermore, Realm does offer Live Data, but lacks the ability to share these live objects across threads, which would be required when running database operations on a background thread.

ObjectBox is considered to be quite fast, but also has limitations on adding relations to the stored objects, and only allows simple reading queries. Moreover, while *ObjectBox* provides automatic migration of entities when updating Java objects (which only requires an additional id attribute provided by the framework), manual migrations are not supported, which leads to problems, if more complex structural changes are performed.

Cloud based storage should be limited to small amounts of data, as the data is synced with the cloud backend, which adds load to the mobile network and may incur costs to the user. Furthermore, as not all services include frameworks which enable local caching, the data may only be accessible if the device has network connectivity.

4.1.2 Usability of the Frameworks

As frameworks should simplify the development of applications, the speed and simplicity to integrate data storage should be high. The following list will summarize some subjective findings of the usability of the framework in terms of simplicity to adopt them and functionality they provide:

- The *SharedPreferences* interface offers a very easy and straightforward interface to store and retrieve data. The methods have simple signatures and the underlying logic is easy to understand.
- The framework to read and write files is inherited from Java, where simple streams for reading or writing files can be easily created. Anything on top of these streams needs to be implemented by the developer. The paths to store specific files are retrieved by helper methods from the *Environment* framework for the most use cases, therefore supporting out-of-the-box security by ensuring that the internal storage of an application is private to it.
- The *SQLiteOpenHelper* is an easy implementable SQLite interface if the developer is accustomed to creating and handling SQL queries. Furthermore, the binding between

Java objects and the returned results needs to be implemented, as well as there are no helpers for creating or updating tables.

- *Android Room* encourages writing SQL queries even more than the *SQLiteOpenHelper*, but also offers binding of results to Java objects and compile-time query validation, which mitigates development errors. It does come with the downside, that relations are not bound automatically, leading to more effort to the developer to, e.g. *lazy load* joined tables.
- While creating entities in *Realm* is easily done, the framework offers live objects which are not thread safe, causing complications, if the database access is not done on the main thread.
- *ObjectBox* is also implemented easily, although ids are only allowed to be numeric, and migrations are only supported automatically, but not manually.
- As framework for *cloud based storage* the *Firebase Realtime Database* has been chosen as it is very popular with Android developers. It provides a powerful and easy to use framework which provides nearly as easy access to data as a simple key-value storage, like *SharedPreferences*, e.g.

4.2 Data Security

An important consideration when choosing a storage technology is the level of required data security and confidentiality. We, therefore, discuss security implications of data storage briefly in this section.

4.2.1 Rooted Devices

Rooted devices bypass the fundamental permission system of Android devices, and therefore weaken the security of stored data. [78] For example, as rooted devices may ignore the fundamental file permission restrictions, the internal storage may be read or write by external applications and thus it is not safe any more to store sensitive or critical data. Moreover, also applications can be altered and reinstalled by other applications, and therefore execute malicious code or extract sensitive data.

Several methods to check whether a device is rooted are available, e.g. checking for *su*-binaries, read/write permissions on internal storages or specific root management applications. [65] However, these checks are not accurate and “*only gives a likely indication of root*” - [65], so they may trigger false positives or false negatives. Such checks are available as libraries for simplicity of implementation, e.g. the *RootBeer* or *RootManager* libraries. [66] [67]

4.2.2 Encryptability

Often it is required to store data in a way that it is not accessible by any unauthorized application or data that is not to be altered by unauthorized applications. This can only be achieved, if the data is securely encrypted. [32]

Out of the frameworks and techniques selected in this thesis, only *Realm* offers encryption with a given key out of the box. SQLite based systems base their security on the fact that the internal storage cannot be accessed by other applications (which is void for rooted devices). Cloud based storage services with the possibility to cache data offline have not been investigated in terms of data encryption in this thesis. However, as the data is synchronized with mostly proprietary backends which are a black-box for the developer (e.g. Firebase), the security of the data stored on the web-server cannot be ensured.

5 Conclusion

As anticipated, there is a great variety of different storage options for android applications, which differ in functionality and suitability. However, no *best* storage option can be defined, as different options may be needed in different use-case scenarios. Furthermore Android Application development is in constant change. So for example Android officially supports a second programming language (*Kotlin*) since Summer 2017, or Android introduced another library for persisting data (*Android Room*) also in Summer 2017. Moreover, the Android application developer community seems to be *Open Source*-minded, so all kinds of modules or libraries are published open source for others to collaborate and use, which enables that great third party libraries building can be built. And as in this thesis discussed, for any type of use case there are several libraries available to use for free.

6 Future Work

In future, next steps to provide a more detailed comparison of different data storage technologies, benchmark tests as well as tests for real applications should be performed. Furthermore, a comparison of the limits in terms of maximum data-sizes or amount of data items for each kind of data storage technology could be researched.

Bibliography

- [1] ADRIAN KINGSLEY-HUGHES: *Android Reference: PreferenceManager*. <http://www.zdnet.com/article/the-real-reason-why-micro-sd-card-slots-are-disappearing-from-smartphones/>, May 2015. Accessed: 22.01.2018.
- [2] ALEX HANDY: *Amazon introduces Lambda, Containers at AWS re:Invent*. <https://sdtimes.com/amazon/amazon-introduces-lambda-containers/>. Accessed: 26.01.2018.
- [3] AMAZON WEB SERVICES, INC.: *Add Cloud APIs to Your Mobile App - AWS Mobile*. <https://docs.aws.amazon.com/aws-mobile/latest/developerguide/add-aws-mobile-cloud-logic.html>. Accessed: 01.01.2018.
- [4] AMAZON WEB SERVICES, INC.: *Add NoSQL Database to Your App - AWS Mobile*. <https://docs.aws.amazon.com/aws-mobile/latest/developerguide/add-aws-mobile-nosql-database.html>. Accessed: 01.01.2018.
- [5] AMAZON WEB SERVICES, INC.: *Add User Data Storage to Your Mobile App - AWS Mobile*. <https://docs.aws.amazon.com/aws-mobile/latest/developerguide/add-aws-mobile-user-data-storage.html>. Accessed: 01.01.2018.
- [6] AMAZON WEB SERVICES, INC.: *Amazon DynamoDB*. <https://aws.amazon.com/dynamodb/>. Accessed: 26.01.2018.
- [7] AMAZON WEB SERVICES, INC.: *Dataset (AWS SDK for Android - 2.6.12)*. <https://docs.aws.amazon.com/AWSAndroidSDK/latest/javadoc>. Accessed: 01.01.2018.
- [8] AMAZON WEB SERVICES, INC.: *What is AWS Mobile?*. <https://docs.aws.amazon.com/aws-mobile/latest/developerguide/what-is-aws-mobile.html>. Accessed: 01.01.2018.
- [9] AMAZON WEB SERVICES, INC.: *What is NoSQL?*. <https://aws.amazon.com/nosql>. Accessed: 29.12.2017.
- [10] AMIT SHEKHAR: *Understanding Context In Android Application*. <https://blog.mindorks.com/understanding-context-in-android-application-330913e32514>. Accessed: 29.12.2017.
- [11] ANDROID OPEN SOURCE PROJECT: *Android Developers: Context*. <https://developer.android.com/reference/android/content/Context.html>. Accessed: 09.12.2017.
- [12] ANDROID OPEN SOURCE PROJECT: *Android Developers: Saving Files*. <https://developer.android.com/training/data-storage/files.html>. Accessed: 28.12.2017.

- [13] ANDROID OPEN SOURCE PROJECT: *Android Developers: SharedPreferences*. <https://developer.android.com/reference/android/content/SharedPreferences.html>. Accessed: 09.12.2017.
- [14] ANDROID OPEN SOURCE PROJECT: *Android Developers: SharedPreferences.Editor*. <https://developer.android.com/reference/android/content/SharedPreferences.Editor.html>. Accessed: 09.12.2017.
- [15] ANDROID OPEN SOURCE PROJECT: *Android Developers: Storage Options*. <https://developer.android.com/guide/topics/data/data-storage.html>. Accessed: 28.12.2017.
- [16] ANDROID OPEN SOURCE PROJECT: *Android Reference: Blob*. <https://developer.android.com/reference/java/sql/Blob.html>. Accessed: 25.01.2018.
- [17] ANDROID OPEN SOURCE PROJECT: *Android Reference: Delete*. <https://developer.android.com/reference/android/arch/persistence/room/Delete.html>. Accessed: 31.12.2017.
- [18] ANDROID OPEN SOURCE PROJECT: *Android Reference: Environment*. <https://developer.android.com/reference/android/os/Environment.html>. Accessed: 29.12.2017.
- [19] ANDROID OPEN SOURCE PROJECT: *Android Reference: File*. <https://developer.android.com/reference/java/io/File.html>. Accessed: 30.12.2017.
- [20] ANDROID OPEN SOURCE PROJECT: *Android Reference: PreferenceManager*. <https://developer.android.com/reference/android/preference/PreferenceManager.html>. Accessed: 21.01.2018.
- [21] ANDROID OPEN SOURCE PROJECT: *Android Reference: Query*. <https://developer.android.com/reference/android/arch/persistence/room/Query.html>. Accessed: 31.12.2017.
- [22] ANDROID OPEN SOURCE PROJECT: *Android Reference: SQLiteDatabase*. <https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>. Accessed: 30.12.2017.
- [23] ANDROID OPEN SOURCE PROJECT: *Android Reference: SQLiteOpenHelper*. <https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html>. Accessed: 07.01.2018.
- [24] ANDROID OPEN SOURCE PROJECT: *Android Reference: Update*. <https://developer.android.com/reference/android/arch/persistence/room/Update.html>. Accessed: 31.12.2017.
- [25] ANDROID OPEN SOURCE PROJECT: *FileOutputStream*. <https://developer.android.com/reference/java/io/FileOutputStream.html>. Accessed: 26.01.2018.

- [26] ANDROID OPEN SOURCE PROJECT: *Reducing Network Battery Drain*. <https://developer.android.com/topic/performance/power/network/index.html>. Accessed: 25.01.2018.
- [27] ANDROID OPEN SOURCE PROJECT: *Requesting Permissions at Run Time*. <https://developer.android.com/training/permissions/requesting.html>. Accessed: 22.01.2018.
- [28] ANDROID OPEN SOURCE PROJECT: *Room Persistence Library*. <https://developer.android.com/topic/libraries/architecture/room.html>. Accessed: 07.01.2018.
- [29] ANDROID OPEN SOURCE PROJECT: *Saving Data Using SQLite*. <https://developer.android.com/training/data-storage/sqlite.html>. Accessed: 30.12.2017.
- [30] ANDROID OPEN SOURCE PROJECT: *Saving Data Using the Room Persistence Library*. <https://developer.android.com/training/data-storage/room/index.html>. Accessed: 31.12.2017.
- [31] ANDROID OPEN SOURCE PROJECT: *Saving Key-Value Sets*. <https://developer.android.com/training/data-storage/shared-preferences.html>. Accessed: 09.12.2017.
- [32] ANDROID OPEN SOURCE PROJECT: *Security Tips*. <https://developer.android.com/training/articles/security-tips.html>. Accessed: 26.01.2018.
- [33] ANDROID OPEN SOURCE PROJECT: *Serializable*. <https://developer.android.com/reference/java/io/Serializable.html>. Accessed: 25.01.2018.
- [34] ANDROID OPEN SOURCE PROJECT: *Supporting Direct Boot: Migrating Existing Data*. <https://developer.android.com/training/articles/direct-boot.html#migrating>. Accessed: 21.01.2018.
- [35] ARTYOM DOGTIEV: *App Download and Usage Statistics 2017*. <http://www.businessofapps.com/data/app-statistics/>, January 2018. Accessed: 26.01.2018.
- [36] BLACK, P. E. Dictionary of Algorithms and Data Structures, [Online]. Available: <https://www.nist.gov/dads/> [2007, December 3]. Vreda Pieterse and Paul E. Black.
- [37] CODEPATH AND CONTRIBUTORS: *Android Guides: Must Have Libraries*. https://github.com/codepath/android_guides/wiki/Must-Have-Libraries. Accessed: 31.12.2017.
- [38] DAN ROWINSKI: *The Rise of Mobile Cloud Services: BaaS Startups Grow Up*. <http://readwrite.com/2012/04/17/mobile-backend-as-a-service-ec/>, April 2012. Accessed: 25.01.2018.
- [39] ECMA INTERNATIONAL: *The JSON Data Interchange Format*. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>, October 2013. Accessed: 25.01.2018.

- [40] GfK: *Smartphone Users Spend as Much Time on Entertainment as Texting – GfK MRI Study*. <http://www.gfk.com/en-us/insights/press-release/smartphone-users-spend-as-much-time-on-entertainment-as-texting-gfk-mri-study/>, February 2016. Accessed: 25.01.2018.
- [41] GOOGLE AND CONTRIBUTORS: *Google Gson*. <https://github.com/google/gson>. Accessed: 25.01.2018.
- [42] GOOGLE DEVELOPERS: *Firebase Realtime Database*. <https://firebase.google.com/docs/database/>. Accessed: 25.01.2018.
- [43] GOOGLE DEVELOPERS: *Firebase Realtime Database: Enabling Offline Capabilities on Android*. <https://firebase.google.com/docs/database/android/offline-capabilities>. Accessed: 21.01.2018.
- [44] GOOGLE DEVELOPERS: *Firebase Realtime Database: Read and Write Data on Android*. <https://firebase.google.com/docs/database/android/read-and-write>. Accessed: 01.01.2018.
- [45] GOOGLE DEVELOPERS: *Firebase Realtime Database: Structure Your Database*. <https://firebase.google.com/docs/database/android/structure-data>. Accessed: 01.01.2018.
- [46] ILYA GRIGORIK: *High Performance Browser Networking*. <https://hpbn.co/optimizing-for-mobile-networks/>. Accessed: 23.12.2017.
- [47] JEMEROV, D. and S. ISAKOVA: *Kotlin in Action*. Manning Publications, 2017.
- [48] KOTLIN: *Boolean - Kotlin Programming Language*. <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-boolean/index.html>. Accessed: 25.01.2018.
- [49] LUKAS BERGSTROM: *Announcing Architecture Components 1.0 Stable*. <https://android-developers.googleblog.com/2017/11/announcing-architecture-components-10.html>, November 2017. Accessed: 22.01.2018.
- [50] MARKUS JUNGINGER AND CONTRIBUTORS: *Performance Benchmarks*. <https://github.com/objectbox/objectbox-performance>. Accessed: 22.01.2018.
- [51] OBJECTBOX LIMITED: *ObjectBox Mobile Database*. <http://objectbox.io/>, 2017. Accessed: 01.01.2018.
- [52] ORACLE: *Arrays (The JavaTM Tutorials > Learning the Java Language > Language Basics)*. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>. Accessed: 25.01.2018.
- [53] ORACLE: *Bitwise and Bit Shift Operators*. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html/>. Accessed: 25.01.2018.

- [54] ORACLE: *Collections (Java Platform SE 7)*. <https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>. Accessed: 25.01.2018.
- [55] ORACLE: *Lesson: Interfaces (The Java™ Tutorials > Collections)*. <https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html>. Accessed: 25.01.2018.
- [56] ORACLE: *Primitive Data Types (The Java™ Tutorials)*. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>. Accessed: 07.01.2018.
- [57] ORACLE: *String (Java Platform SE 7)*. <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>. Accessed: 25.01.2018.
- [58] ORACLE: *Strings (The Java™ Tutorials)*. <https://docs.oracle.com/javase/tutorial/java/data/strings.html>. Accessed: 07.01.2018.
- [59] ORACLE: *The List Interface (The Java™ Tutorials > Collections > Interfaces)*. <https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html>. Accessed: 25.01.2018.
- [60] RADOSLAV YANKOV: *Realm, ObjectBox or Room. Which one is for you?*. <https://notes.devlabs.bg/realm-objectbox-or-room-which-one-is-for-you-3a552234fd6e>. Accessed: 01.01.2018.
- [61] REALM: *Realm Database*. <https://realm.io/products/realm-database>. Accessed: 01.01.2018.
- [62] REALM JAVA TEAM: *Reducing your Android APK size When Using Native Libraries*. <https://academy.realm.io/posts/reducing-apk-size-native-libraries/>, February 2015. Accessed: 22.01.2018.
- [63] RON PLEW, RYAN STEPHENS: *Welcome to the World of SQL*. <http://www.informit.com/articles/article.aspx?p=29583>, 2002. Accessed: 07.01.2018.
- [64] SARAH PEREZ: *U.S. consumers now spend 5 hours per day on mobile devices*. <https://techcrunch.com/2017/03/03/u-s-consumers-now-spend-5-hours-per-day-on-mobile-devices/>, March 2017. Accessed: 27.12.2017.
- [65] SCOTT ALEXANDER-BOWN: *Detecting Root on Android*. <https://medium.com/@scottyab/detecting-root-on-android-97803474f694>, March 2016. Accessed: 26.01.2018.
- [66] SCOTT ALEXANDER-BOWN: *RootBeer*. <https://github.com/scottyab/rootbeer>, September 2016. Accessed: 26.01.2018.
- [67] SHIQI JIANG: *Root Manager*. <https://github.com/Chrisplus/RootManager>, June 2016. Accessed: 26.01.2018.

- [68] SOLID IT GMBH: *Key-value Stores*. <https://db-engines.com/en/article/Key-value+Stores>. Accessed: 26.11.2017.
- [69] SQLITE CONSORTIUM MEMBERS: *About SQLite*. <https://www.sqlite.org/about.html>. Accessed: 10.12.2017.
- [70] SQLITE CONSORTIUM MEMBERS: *Datatypes in SQLite Version 3*. <https://sqlite.org/datatype3.html>. Accessed: 17.12.2017.
- [71] SQLITE CONSORTIUM MEMBERS: *Implementation Limits For SQLite*. <https://www.sqlite.org/limits.html>. Accessed: 22.12.2017.
- [72] STATISTA: *Number of smartphone users worldwide from 2014 to 2020 (in billions)*. <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>. Accessed: 27.12.2017.
- [73] STATISTA: *Number of available applications in the Google Play Store from December 2009 to September 2017*. <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>, 2017. Accessed: 27.12.2017.
- [74] STATISTA: *Number of available apps in the Apple App Store from July 2008 to January 2017*. <https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/>, 2017. Accessed: 27.12.2017.
- [75] STEVE COPE: *Understanding and Managing Android Storage*. <http://www.stevesandroidguide.com/understand-android-storage/>, February 2017. Accessed: 22.01.2018.
- [76] STROUD, A.: *Android Database Best Practices (Android Deep Dive)*. Addison-Wesley Professional, 2016.
- [77] WARACLE LTD: *The Rise of Mobile Cloud Services: BaaS Startups Grow Up*. <https://waracle.com/how-to-choose-the-right-backend-as-a-service-baas-platform/>, February 2016. Accessed: 25.01.2018.
- [78] WHITSON GORDON: *Top 10 Reasons to Root Your Android Phone*. <https://lifehacker.com/top-10-reasons-to-root-your-android-phone-1079161983>, August 2013. Accessed: 26.01.2018.
- [79] YURA BONDARENKO: *Essential Guide to Making Your App Work Offline*. <https://www.romexsoft.com/blog/make-app-work-offline/>. Accessed: 25.01.2018.

List of Figures

Figure 1	Number of smartphone users worldwide from 2014 to 2020 (in billions) [72] . . .	1
Figure 2	Number of available applications in the Google Play Store from December 2009 to September 2017 [73]	2
Figure 3	Number of available apps in the Apple App Store from July 2008 to January 2017 [74]	2
Figure 4	US Daily Mobile Time Spent [64]	4
Figure 5	Room architecture diagram [30]	23
Figure 6	Performance for 100k / 10k elements measued in ms [60]	27

List of Tables

Table 1	Latency overhead of a single HTTP request [46]	3
Table 2	Summary of facts about internal and external storage. This table has been excerpted from the Android Developer Training website for saving files. [12]	43

List of Listings

Listing 1	Example: SharedPreferences	11
Listing 2	Example: Create a directory and write to a file	13
Listing 3	Check if the external storage is mounted [12]	14
Listing 4	Request a permission at runtime (Minimal working example) [27]	14
Listing 5	Asynchronous Callback for permission requests at runtime within an Activity (Minimal working example) [27]	15
Listing 6	Check if external storage is mounted or mounted in read-only mode [12]	15
Listing 7	Example implementation if the SQLiteOpenHelper class	20
Listing 8	Example insert operation	20
Listing 9	Example retrieve operation	22
Listing 10	Example delete operation	22
Listing 11	Room Example entity class	25
Listing 12	Room Example Data Access Object class	25
Listing 13	Room Example database class	25

List of Abbreviations

Singleton An object having a single instance within the application

4G network fourth generation broadband cellular network

3G network third generation broadband cellular network

ASCII American Standard Code for Information Interchange

PDS Passive Data Structure

JVM Java Virtual Machine

SQL Structured Query Language

NoSQL “A term used to describe high-performance, non-relational databases.” [9]

SD card A non volatile memory card for use in portable devices

DAO Data Access Object

OR-Mapper Object-relational mapping tool

Android NDK Android Native Development Kit

ACID Atomicity, Consistency, Isolation and Durability

MBaaS Mobile Backend as a Service

AWS Amazon Web Services

JSON JavaScript Object Notation

A Comparison of Internal and External storage

Internal storage	External storage
• It's always available. • Files saved here are accessible by only your app. • When the user uninstalls your app, the system removes all your app's files from internal storage.	• It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device. • It's world-readable, so files saved here may be read outside of your control. • When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from <code>getExternalFilesDir()</code>.
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.	External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Table 2: Summary of facts about internal and external storage. This table has been excerpted from the Android Developer Training website for saving files. [12]