Android Jetpack

Tobias Fischer
University of Applied Sciences Upper Austria
Faculty for Informatics, Communications and Media
Degree in Mobile Computing
4232 Hagenberg, Austria
s2010237030@fhooe.at

Michael Zauner
University of Applied Sciences Upper Austria
Faculty for Informatics, Communications and Media
Degree in Mobile Computing
4232 Hagenberg, Austria
s2010237032@fhooe.at

***Abstract*—Android has a huge library landscape, although it is difficult to choose the right one. Jetpack is a collection of libraries designed to make app development easier and decrease boilerplate code. Jetpack libraries use contemporary design approaches to reduce crashes and memory leaks. This paper gives a short insight into Android Jetpack, its major libraries and its benefits and drawbacks.**

I. Introduction

Technologies are constantly changing, and this is true for Android as well. The number of libraries for Android developers is increasing rapidly, and therefore it is very difficult to choose the right ones. Furthermore, it is not an easy task to keep the architecture of an app in a manner that is easily understandable for other people. Uncontrollable growth must be prevented.

In 2018, Google has decided to give Android a new coat of paint with Android Jetpack. Android Jetpack should be a suite of libraries which should reflect the best practices in developing an Android application.

Android Jetpack is constructed like LEGO building blocks – the libraries can be plugged together according to requirements. Splitting a framework into multiple blocks is necessary when it grows into an unmanageable state.

The most important libraries of Android Jetpack are Jetpack Compose, Room, Navigation and Databinding. According to their website [1], Jetpack consists of almost 100 libraries (as of June 2022).

II. Related Work

A. General Android Libraries

Native Libraries

In addition to the Linux kernel layer, the Android platform contains a number of native libraries. Most of the functionality provided through the Android runtime layer is available through these native libraries [2].

According to Cinar [2], the most notable of them are:

- *SQLite*: in-memory, relational SQL database for persisting and accessing the application's data
- *WebKit*: enables HTML, CSS and JavaScript to include web technology
- *OpenGL ES*: rendering functionality (2D and 3D)
- *Open Core*: record and play back audio and video content
- *OpenSSL*: secure communication with SSL/TLS

Android Support Library

While targeting a lower API level will increase the app's audience, it also limits the Android platform features to use. To overcome this trade-off, the Android Support Library was introduced. The Android Support Library Package is a collection of code libraries that provide backward compatible versions of recent Android APIs. An important fact is that the Android Support Library does not cover every new API because sometimes new OS features are required [2].

III. Android Jetpack

A. Jetpack in general

Jetpack is a set of many useful libraries which are used in Android development. It combines old and new libraries, therefore managing and updating gets much easier. As of 2018 when Jetpack got announced at the Google I/O one of the first steps was to rename the old *android.support.** libraries which are used for compatibility across multiple Android versions, to the new namespace *androidx.** [3].

The components used in Jetpack are not bundled and each one must be imported explicitly. The available parts can be categorized into four categories, as shown in figure 1. The categories are architecture, user interface, foundation and behavior. Which components are in which category is shown in the following figure:

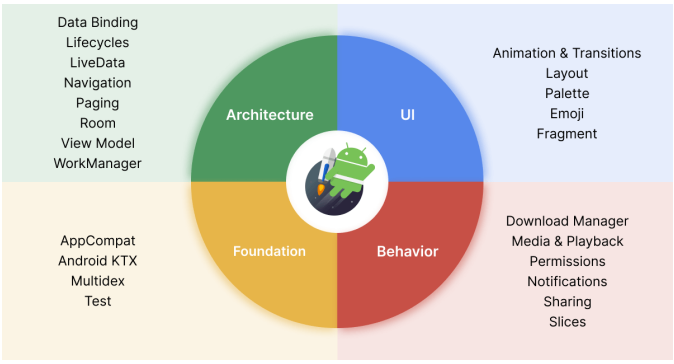

Fig. 1. Jetpack components and its categories [4]

One of the major goals of Jetpack is the integration with the programming language *Kotlin*. At the Google I/O 2019, Google announced the future development of Android will be

written in Kotlin, because you can code more cleaner code than with the boilerplate language *Java*. Kotlin and Java are 100% interoperable, which means you can call Kotlin-code from Java and call Java-code from Kotlin [5].

B. Jetpack Compose

The biggest renewal of the Android framework comes with Jetpack Compose. Compose is a new UI framework for Android. According to the statements of Google [6] the framework speeds up UI development, requires less code, and has many powerful tools.

Compared to building the UI implicitly in XML files, Jetpack Compose is meant to be declarative. Declarative UI allows to put design and programmatic logic in one single file [6]. Some other popular examples using this approach are React, Flutter, SwiftUI and many more.
The main benefits according to the "Android for Developers" web page [6] are:

- less Code
- intuitive
  - writing code in Kotlin (declarative)
  - Android takes care of the rest
- accelerate Development
- powerful
- performs better than XML
- usage alongside existing code
- test integration

Besides the list of advantages, there are also some drawbacks of using Jetpack Compose:

- a lot of magic happens in the background
- difficult to understand
- the preview is in a early state and leads to some teething troubles

Declarative Programming Paradigm

In the old approach (imperative), the layout is built with a tree of UI widgets. There, the state changes of the application need to be updated manually with methods like `setText(String)` or `setVisibility(Integer)`. This technique is very error prone because a update of those widgets could be missing at some point. Another error cause is that the view was removed from the UI and creates an illegal state [7].

To avoid these errors, the declarative approach has been developed in recent years. This approach describes the desired end result and what it should look like, instead of listing all the execution steps that would lead to a result. On every state change the entire screen gets regenerated from scratch which could be very expensive in terms of time, computing power and battery usage. Therefore Compose tries in the Recomposition process to choose the parts which have to be redrawn and the other ones are skipped. Recomposition is performed only on objects that are intended to change. [7].

Composable Functions

The basic functionality of Compose is relying on composable functions. Composable functions convert data into UI elements, so they may have parameters and they do not return anything. Instead of returning something, the functions are describing the screen state instead of building UI widgets. These functions are annotated with `@Composable` annotation. Another important fact is that it is not possible to set or get the function. It simply can be overwritten by calling the same functions with other arguments [7].

A simple composable function that displays "Hello" with a name specified in the parameter:

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name!")
}
```

In the matter of fact that composable functions are written in Kotlin, it is possible to use loops (for, while, ...), contitions (if, when, ...) or any other logical construct [7].

Some other important facts from the Google documentation [7] are as follows:

- Composable functions can execute in any order and parallel
- Recomposition ignores composable functions and lambdas where possible
- Recomposition is optimistic (restarts recomposition again whenever a new parameter arrives)
- Composable functions should be very fast as they might run quite frequently (can lead to poor performance if this is not followed)

Managing States

The only way to update the UI is by calling the same composable with new arguments [8]. For state changes, functionality is needed that saves a state and calls the current composable again with new arguments to redraw the UI. This basic functionality is implemented as follows:

```
var state by remember {mutableStateOf("")}
```

A composable function can store a single object in memory by usage by of `remember` keyword. During composition the value is stored and the value gets returned while recomposition. Therefore, the value can be used with the variable (for example `state`) during the recomposition process.

The delegated property `mutablestateof` creates a observable which would schedule a recomposition of any composable function which reads the changed value. [8]

An example of saving the state of a text field and using the "Greeting" function shown earlier:

```
@Composable
fun SomeInputText() {
    var name by remember {
        mutableStateOf("World")
    }

    Greeting(name)
```

```
    TextField(
        value = name,
        onValueChange = { name = it }
    )
}
```

Phases

Jetpack Compose renders each frame in three different phases. In the standard Android View system, the three phases are called Measure – Layout – Drawing. Instead of the Measure phases, Compose has a phase called Composition.

The phases mentioned in the Android Developer platform [9] are as follows:

- Composition: Which parts of the user interface should be displayed?
- Layout: Where to place objects on the UI?
- Drawing: How to render the UI?

Architectural Layering

Since Jetpack Compose consists of more than one project, it is assembled from several modules to form a complete stack. The major layers are [10]:

- Material: implementation of Material Design (theming system, designed components, ripple displays, icons, ...)
- Foundation: system independent building blocks (`Row`, `Column`, `LazyColumn`, ...)
- UI: basic UI capabilities (`Modifier`, ...)
- Runtime: provides fundamental functionality (`remember`, `mutableStateOf`, `@Composable`, ...)

Example of the "new" RecyclerView

RecyclerView were used to represent a list of data on the UI. Every old RecyclerView needs an Adapter with a ViewHolder and produces much boiler plate code. Compose introduces the `LazyColumn` and `LazyRow` and therefore no longer requires RecyclerViews.

An example of creating a list with LazyColumn:

```
LazyColumn(Modifier.padding(it)){
    items(someList) { someItem ->
        Text(someItem)
    }
}
```

C. Room

In comparison of using Shared Preferences instead of Room it is possible to persist non-trivial amounts of structured data. The Room persistence library provides an abstraction layer over the SQLite database. It allows easier access to the database, but uses SQLite in the background. The most important benefits are [11]:

- compile-time validation of SQL queries
- minimizes boilerplate code with annotations
- enables migrations

The major components of Android Room are [11]:

- Database Class: main access point for the database connection
- Data Entities: tables of database
- Data Access Objects (DAOs): methods for querying, updating and deleting data in the database

In [12] it is shown how theses components work together. The Room Database get the queries from the DAO and executes them. The result will then be stored in the entities via the getter/setter methods.

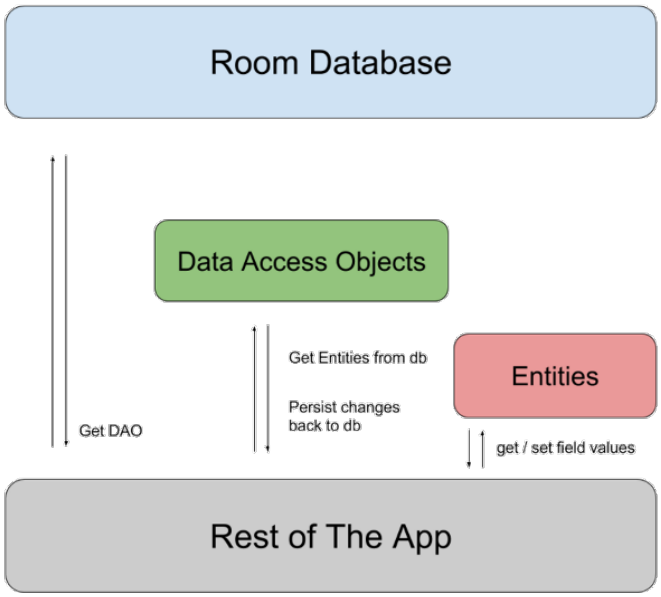

Fig. 2. Architecture of Room [12]

Example of a DAO for some table supporting a select, insert and delete statement:

```
@Dao
interface SomeTableDao {
    /**
     * @return all entries
     */
    @Query("SELECT * FROM some_table")
    fun getAll(): Flow<List<SomeEntry>>

    /**
     * insert a range of entries
     */
    @Insert(onConflict =
        OnConflictStrategy.REPLACE
    )
    fun insertAll(vararg entries: SomeEntry)

    /**
     * delete all entries
     */
    @Query("DELETE FROM some_entry")
    fun deleteAll()
}
```

D. Navigation

The interactions that let users move between, inside, and outside of the various pieces of content in a app are referred

to as navigation. The Navigation component of Android Jetpack assists in implementing navigation, from straightforward button clicks to more intricate patterns like app bars and the navigation drawer. The navigation component also ensures a consistent and predictable user experience by adhering to a set of principles [13].

The main navigation concept conists of three major parts:

- NavController manages the navigation workflow. When a new destination should be visible, the NavController sends the UI elements to the NavHost. It also take care of the Back Stack when interacting with different destinations. When the user wants to go back, the NavController know exactly which destination was the previous one. The controller is read out as follows:
  `val navController = rememberNavController()`
- NavHost is a container which displays the composable functions of the destinations. Here follows an example:

```
val start = "screen1"
NavHost(navController, start) {
    composable("screen1") {
        Screen1()
    }
    composable("screen2") {
        Screen2()
    }
}
```

- Navigation Graph: defines the possible ways to navigate through the application. Before Jetpack Compose this component was declared in a XML file, where the single destinations and the interactions between them can be defined. Like in this example [14] you define a start destination and from there on you have 2 different possible ways to navigate through the application. You can go to the registration or view the leader board.

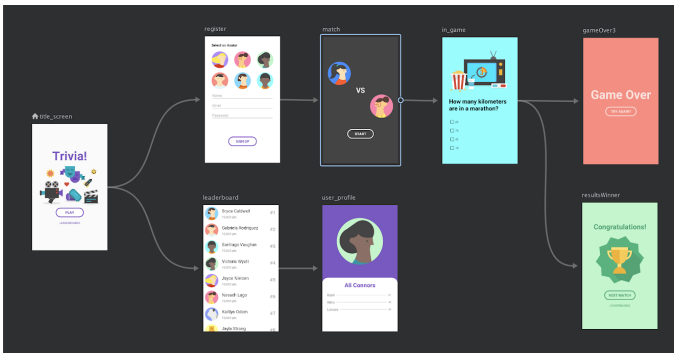

Fig. 3. Example navigation graph [14]

E. Interacting with ViewModels and Kotlin Flows

Asynchronous Data

Flow is a datatype introduced in Kotlin Coroutines and it is used to collect data which changes in the background (e.g.: updating news from a website). Normally non-simultaneous functions return a single value, but what if you need multiple values? This is where Flow comes into action. It generates a continuous data stream where you can attach to it and collect its asynchronous data. A Flow consists of two parts [15]:

- *producer*: provides the data stream (mostly asynchronous due to Coroutines) using *emit()*
- *consumer*: fetches the data and do other processing (e.g: displaying in UI) using *collect()*

ViewModel

A ViewModel is designed to store and handle UI relevant data in an application, taking into account the Android lifecycle. It holds the states and values of its objects (e.g.: like a Flow list of news) regardless of system configuration changes (e.g.: changing screen orientation). ViewModel is a observable holder for data and you can get notified when the data changes. In a modern application architecture a ViewModel transforms the data from a repository (local or remote source) to a Flow/LiveData list. When ever the list content changes it will update the UI.

A `@Composable` function can collect the asynchronous data from a ViewModel as a state. Whenever a change in the data is recognized, the value of the variable will be changed and it will automatically update the UI (if the data needs to be displayed)

IV. Conclusion

The Android development world is growing rapidly. It is hard for developers to stay up-to-date all the time. Libraries and APIs are constantly changing. Google tries to counter this problem with Android Jetpack. They provide a collection of libraries to help developers cope with this problem. With Jetpack, developers have all the tools to create a modern android application.

On the other hand, Jetpack also has a not-so-good side. For example, it is hard for people who learned Android programming the older way, like the XML-style UI, to switch to the declarative UI of Jetpack Compose. Everything they learned about fragments is different now, because fragments just don't exist anymore. There are only Composable functions left to interact with the UI.

Jetpack has a great documentation about all components and also the GitHub account of Google is a great starting point. They provide coding examples to get in touch with the best coding practices in Android development. It just takes some time to get used to a new style of programming. But it is definitely recommended for an Android beginner to start with the old XML-style UI because it necessary to learn about the components a application really has and how these work together. Jetpack Compose does a lot of magic in the background and when someone really wants to learn Android development it is necessary to know what these magically short code snippets really do. Currently another big drawback is, that some features of Jetpack are not yet mature enough (for example the preview of Jetpack Compose).

To sum it up, Android Jetpack is really great. Room, Navigation and Android KTX (Kotlin extensions) make most of the developers happy, it simplifies so many things and saves

the lifes from bad practices. Jetpack Compose will be the future of UI development, but it will take some time to adapt to this new style. Everybody is looking forward for the new handy libraries in Android Jetpack.

REFERENCES

[1] "Android jetpack," accessed: 03.06.2022. [Online]. Available: https://developer.android.com/jetpack

[2] O. Cinar, "Android platform," in *Android Quick APIs Reference*. Apress, Berkeley, USA: Springer, 2015.

[3] G. Allen, "Introducing Android," in *Android for Absolute Beginners*. Apress, Berkeley, USA: Springer, 2021.

[4] "Figure 1." [Online]. Available: https://img.orangesoft.co/media/android-jetpack-components.png

[5] "Android's kotlin-first approach," accessed: 23.05.2022. [Online]. Available: https://developer.android.com/kotlin/first

[6] "Jetpack compose," accessed: 23.05.2022. [Online]. Available: https://developer.android.com/jetpack/compose

[7] "Thinking in compose," accessed: 24.05.2022. [Online]. Available: https://developer.android.com/jetpack/compose/mental-model

[8] "State and jetpack compose," accessed: 24.05.2022. [Online]. Available: https://developer.android.com/jetpack/compose/state

[9] "Jetpack compose phases," accessed: 01.06.2022. [Online]. Available: https://developer.android.com/jetpack/compose/phases

[10] "Jetpack compose architectural layering," accessed: 01.06.2022. [Online]. Available: https://developer.android.com/jetpack/compose/layering

[11] "Save data in a local database using room," accessed: 24.05.2022. [Online]. Available: https://developer.android.com/training/data-storage/room

[12] "Figure 2." [Online]. Available: https://developer.android.com/static/images/training/data-storage/room_architecture.png

[13] "Principles of navigation." [Online]. Available: https://developer.android.com/guide/navigation/navigation-principles

[14] "Figure 3." [Online]. Available: https://miro.medium.com/max/1400/1*TPe93-NWXAili2anuyel4A.png

[15] "Kotlin flows on android," accessed: 07.06.2022. [Online]. Available: https://developer.android.com/kotlin/flow