Linca Răzvan Cosmin

Group 234/2

04/06/2021

**Framework Design**

**Final Project**

**Mobile application - proof of concept - using Jetpack Compose**

1. Introduction

For the final project, I chose to present the main concepts of a new Android framework for the development of mobile applications in Kotlin, called **Jetpack Compose**.

The application presents some basic concepts of this framework, through a simple questionnaire, which aims to use several graphic and design elements, in order to highlight the ease with which these are made.

Thus, the application contains elements such as: text inputs, single or multiple selection questionnaires, validators, date picker, event for using the camera and taking the picture.

The application, together with the documentation, can be accessed at this link:

<https://github.com/cosminlinca/fd2021-poc-jetpack-compose-quiz>

1. About Jetpack Compose

*“Jetpack Compose is Android’s modern toolkit for building native UI. It simplifies and accelerates UI development on Android. Quickly bring your app to life with less code, powerful tools, and intuitive Kotlin APIs.”* is the official definition.

In terms of how to create and create interfaces, the UI architecture of Android applications (developed in Java / Kotlin) is well known, which is based on hierarchies of graphical elements made in XML files by programmers. However, this approach **is no longer relevant**.

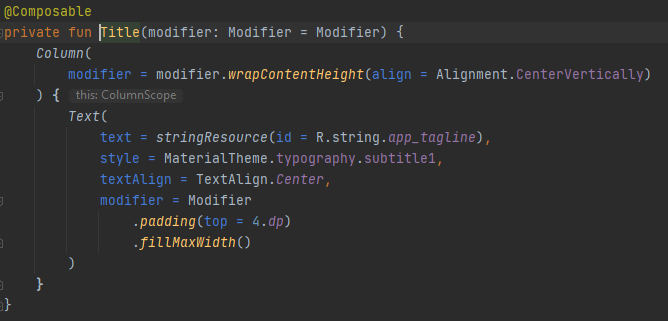
In recent years, the entire industry has begun to move to a **declarative interface model**, which greatly simplifies the architecture of construction and realization, but also the rendering of graphical components.

The technique works by conceptually regenerating the entire screen from scratch, then applying only the necessary changes. This approach avoids the complexity of manually updating a status view hierarchy. Compose is a **declarative** interface framework.

* 1. Core concepts
     1. Composable functions

Jetpack Compose is built around **composable** functions.

These features allow you to define the interface elements of the application, describing the **shape and dependencies of the data**, rather than focusing on the process of building the UI. To create a composable function, just add the **@Composable** annotation to the function name.



* + 1. Layouts

UI elements are hierarchical, with elements contained in other elements (by call). Using Jetpack composites, create a UI hierarchy by **calling composable functions from other composable functions**, so that layouts are created.



* + 1. Recomposition

The concept of *"Recomposition"* is one of the most important with which the Jetpack composite framework comes. In the old architecture, to change a widget, it was necessary to call a setter to change the internal state. Now, using Compose, the @Composable function **is called again**, with new data, the widget being recomposed.

The Compose framework can intelligently recompose **only** the components that changed. Also, compose functions can optimize recomposition by running composable functions in parallel.

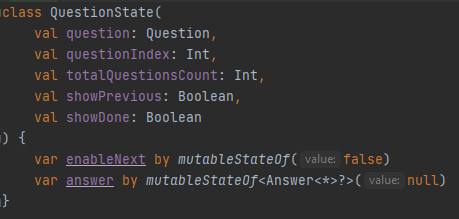
* + 1. Managing state

Composable functions can save a single object in their own memory, using **remember** composable. A value computed by remember is stored in the composition during initial composition, and the stored value is returned during recomposition.

To create an object of type **MutableState** <T> the phrase **mutableStateOf** is used, and it is an observable type integrated with compose runtime.

An example of declaring a MutableState object is:

val mutableState = remember { mutableStateOf(default) }



Example: mutableStateOf<Answer<\*>?>

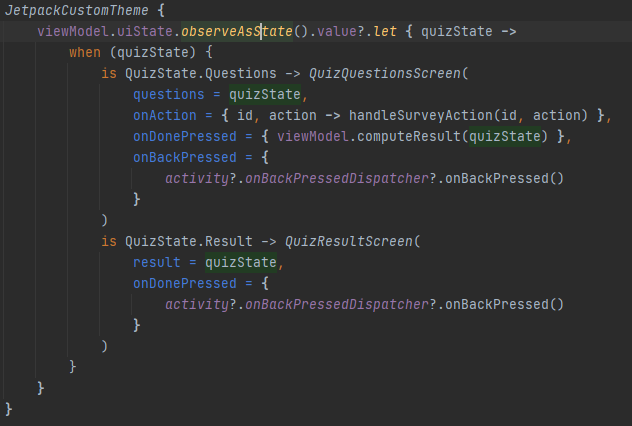
Jetpack Compose also support other observable types. Before reading another observable type in the Jetpack, it is necessary to make a conversion to **State** <T>, so that the framework knows how to automatically recompose the component when changing/updating the state.

* + 1. ViewModel

ViewModels are the recommended and used state support for composables that are up in the Compose UI. ViewModels survives **configuration/state changes** so you can encapsulate user interface status and events without having to deal with the activity or fragment of the lifecycle that hosts your code.

ViewModels should expose the state in an observable holder, such as **LiveData**.

An important function is **observeAsState**, which observes a LiveData<T> and returns a State<T> object that is updated whenever the LiveData changes. State<T> is an observable type that Jetpack Compose can use directly. observeAsState will observe the LiveData only while it is in the composition.

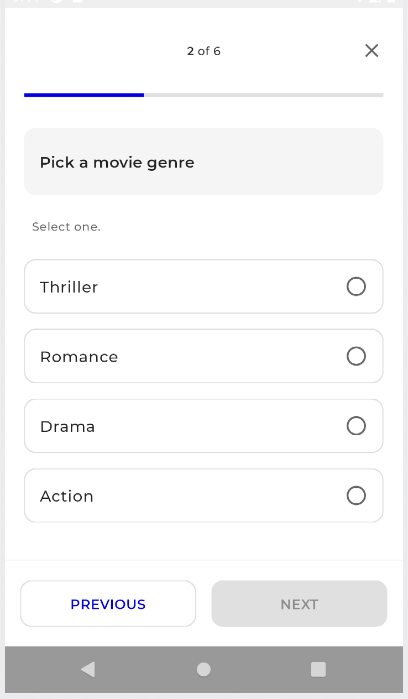


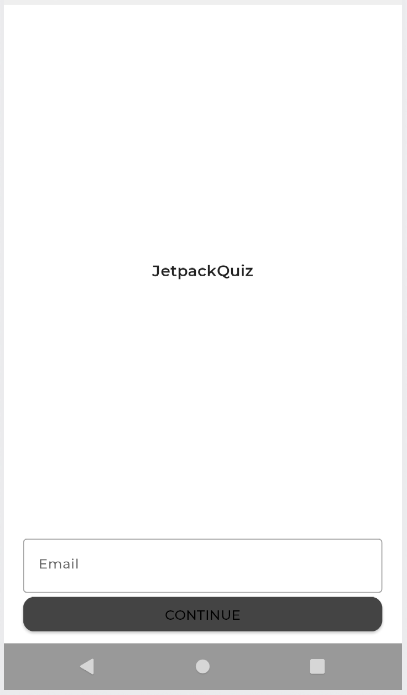
1. Application structure
   1. Setup

To install and run this proof-of-concept app, you need to use the latest **Canary** version of Android Studio, that is still in beta version, so it’s not officially released.

* 1. Features

The application contains two screens: The **welcome** screen, where the user must enter a valid email address (email validator), and the **main screen** with questions related to the quiz, along with other activities.





**MainActivity** is the application's entry point. Each screen is implemented inside a Fragment and MainActivity is the host Activity for all of the Fragments. The navigation between them uses the Navigation library. The screens and the navigation are defined in Navigation.kt.

For the home screen, a simple TextField was used on which an email validation was applied, using a **TextFieldState** and an **EmailState**.

The **quiz main screen** allows the user to complete the questions, showing how to handle complex states. The status of the interface is preserved and restored to redials triggered by various reasons, such as a configuration change or a new question displayed on the screen.

The most complex elements were used in the quiz module, at UI level being used elements such as RadioButton, Checkbox, Slider, Scaffold (for screens with top bar, bottom bar and body), there is also complex management of the states at Quiz model view level - **QuizState**, which contains a list of states for each question – **QuestionState**.

Regarding the data used, a mock repository was used to store the questions – **QuizRepository**.

1. Conclusion

Jetpack Compose offers a modern approach to defining the interface in a declarative way, which allows you to effectively separate concerns. From my point of view, the power of composable functions and the ease with which they are created will make this framework quickly accepted by developers, as the way to create and interact with UI components is much more pleasant and simple.

1. References

* <https://developer.android.com/jetpack/compose>
* <https://developer.android.com/jetpack/compose/mental-model>
* <https://developer.android.com/jetpack/compose/tutorial>
* <https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1-of-2-ca316fe39050>
* <https://developer.android.com/jetpack/compose/state>