Android-Apps mit Jetpack Compose

Gekonnt migriert

Thomas Künneth

Geht es nach Google, soll man Oberflächen für Android-Apps von nun an mit Jetpack Compose erstellen. Aber wie kombiniert man vorhandene Views und Composable-Funktionen?

Ende Juli 2021 veröffentlichte Google die erste stabile Version von Jetpack Compose, dem neuen deklarativen UI-Framework für Android. Geht es nach dem Willen des Android-Herstellers, schreiben Programmierer neue Apps damit. Aber was ist mit den vielen bestehenden Anwendungen, die sie ebenfalls aktiv pflegen? Müssen Entwickler diese vollständig umstellen oder lassen sie sich gemischt betreiben? Eine schrittweise Migration ist mehr als nur wünschenswert: Das Umkrempeln in einem Rutsch wäre insbesondere für große Anwendungen ein riesiger, wirtschaftlich nicht sinnvoller und vermutlich kaum zu bewältigender Aufwand. Das hat man natürlich auch bei Google verstanden und bietet mehrere Umstiegspfade an. Vor einem Blick darauf muss man sich aber die Architektur von Android-Apps in Erinnerung rufen.

Architektur einer Android-App

Programme für Android bestehen aus mehreren Grundbausteinen, beispielsweise Services, Broadcast-Receivern und Activities. Letztere zeigen die Benutzeroberfläche an und interagieren mit den Anwendern. Insbesondere für die frühen Android-Versionen bedeutete dies: Alles, was inhaltlich zusammengehört und – abgesehen vom Scrollen – auf den Bildschirm passt, bildet eine Activity. Die Navigation zu den verschiedenen Teilen einer App wurde durch das Starten weiterer Activities umgesetzt – jede repräsentierte also einen Aspekt wie „Nachricht erstellen“, „Nachricht anzeigen“, oder „Liste aller Nachrichten anzeigen“.

Die Oberfläche einer Activity beschreibt man traditionell als Komponentenbaum in XML, den die Applikation zur Laufzeit zu einem Objektgeflecht entfaltet und mit `setContentView()` anzeigt. Ein Wiederverwenden von Bedienelementbäumen sahen frühe Android-Versionen nur auf XML-Ebene – `<merge>` und `<include>` – vor. Unterschiedliche Activities konnten zwar dieselbe Layoutdatei nutzen, mussten aber alle Aktionen – zum Beispiel das Reagieren auf einen Button-Klick – selbst implementieren. Das war nicht besonders praktisch. Hier legte Android 3, eine Version speziell für Tablets, nach und führte Fragmente ein: Komponenten mit eigenem Lebenszyklus, die Benutzeroberflächen anzeigen und auf Eingaben reagieren, also UI-Logik enthalten. Auch sie entfalten XML-Dateien. Nimmt man also ein vollständiges Layout als zu migrierende Oberfläche an, befindet man sich je nach Architektur der App auf der Ebene von Fragmenten oder Activities.

Unglücklicherweise hatte Google über viele Jahre hinweg kaum Empfehlungen ausgesprochen, wie man in Fragmenten und Activities den Code zum Steuern des UI und den der Fachlogik strukturiert oder – besser – voneinander trennt. Je älter der Code, desto enger und gewachsener ist üblicherweise die Verzahnung – hier muss man sich überlegen, inwiefern man noch sinnvoll auf Jetpack Compose umsteigen kann. Erst die Android Architecture Components aus dem Jahr 2017 empfahlen bewährte Entwurfsmuster und lieferten entsprechende Bibliotheken. Setzt eine App beispielsweise `Repository`, `LiveData` oder `ViewModel` ein, kann man darüber nachdenken, die Oberfläche komplett auf Jetpack Compose zu migrieren.

Tücken im Detail lauern aber auch hier. Beispiel Action Bar: Je nach Activity enthält sie das Burger-Menü oder einen Zurück-Pfeil, das Optionsmenü und in vielen Fällen permanent sichtbare Action Items. Die Action Bar lässt sich durch die Activity sowie Fragmente konfigurieren. Jetpack Compose nutzt aber nicht die Action Bar der Activity, sondern eine eigene –

X-TRACT

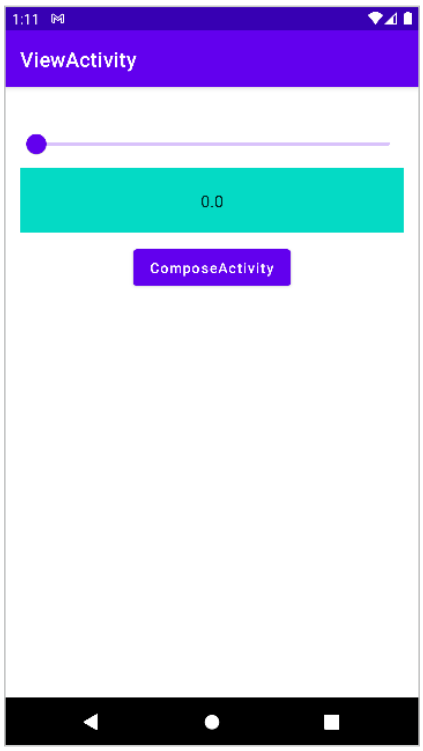
- Generell lassen sich Views und Composable-Funktionen einfach mischen.
- Daten sollten Entwickler zwischen Views und Composable-Funktionen mithilfe eines ViewModels austauschen.
- Vor einer Migration der kompletten App sollte man ihren allgemeinen Zustand bewerten.

genannt `TopAppBar()` – als Bestandteil eines `Scaffold()`.

Composable-Funktionen richtig integrieren

Je nach App stehen dann größere Umbaumaßnahmen an, weil man die alte Logik an Compose anpassen muss. Das betrifft übrigens auch die Navigation: Gerade neuere Apps nutzen gerne die Bibliothek Jetpack Navigation. Sie basiert auf Navigationsgraphen, die Ziele – zum Beispiel Activities oder Fragmente – und zu übergebende Daten enthalten. Zwar gibt es die Komponente auch in einer Version für Compose, allerdings funktioniert diese ganz anders. Navigationsziele sind hier Composable-Funktionen. Beim Umbau kompletter Fragmente oder Activities ist also hoher Aufwand zu erwarten – ob sich das rechnet? Machbarer erscheint vielmehr das Umstellen einzelner Komponenten.

Wie das funktioniert, zeigt die App JetpackComposeMigrationDemo. Sie findet sich auf dem *iX*-Listing-Server (siehe ix.de/z6q5). Das Programm besteht aus den beiden Activities `ViewActivity` (siehe Abbildung 1) und `ComposeActivity` (siehe Abbildung 2). Erstere (Listing 1) integriert eine Composable-Funktion in eine Layoutdatei. Hierzu entfaltet die Anwendung das Layout aus der Datei layout.xml (Listing 2) mit `LayoutBinding.inflate(layoutInflater)` und zeigt es mit `setContentView(binding.root)` an.



`ViewActivity` **integriert eine Composable-Funktion in eine Layoutdatei (Abb. 1).**

`ComposeActivity` **integriert eine Layoutdatei in ein Compose-UI (Abb. 2).**

Das Layout besteht aus einem Slider (`com.google.android.material.slider.Slider`) sowie einem Container für Composable-Funktionen (`androidx.compose.ui.platform.ComposeView`). Bewegt der Anwender den Slider, zeigt `ComposeDemo()` den neuen Wert in einem `Text()` an. Ein Button startet beim Anklicken mit `startActivity(i)` die Activity `ComposeActivity`. Dabei wird der aktuelle Slider-Wert als Extra übergeben.

Zunächst muss man die Referenz auf die `ComposeView` ermitteln – entweder über `findViewById()` oder, wie im Beispiel, das

Listing 1: ViewActivity.kt

```
package eu.thomaskuenneth.jetpackcomposemigrationdemo

import ...
import eu.thomaskuenneth.jetpackcomposemigrationdemo.databinding.
                                                        LayoutBinding
import eu.thomaskuenneth.jetpackcomposemigrationdemo.ui.theme.
                                                JetpackMigrationDemoTheme

const val KEY = "key"
class ViewActivity : AppCompatActivity() {

    private lateinit var binding: LayoutBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val viewModel: MyViewModel by viewModels()
        viewModel.sliderValue.observe(this) {
            binding.slider.value = it
        }
        viewModel.setSliderValue(intent.getFloatExtra(KEY, 0F))
        binding = LayoutBinding.inflate(layoutInflater)
        setContentView(binding.root)
        binding.slider.addOnChangeListener { _, value, _ ->
viewModel.setSliderValue(value) }
        binding.composeView.run {
            setViewCompositionStrategy(ViewCompositionStrategy.
                                    DisposeOnViewTreeLifecycleDestroyed)
            setContent {
                val sliderValue = viewModel.sliderValue.observeAsState()
                sliderValue.value?.let {
                    ComposeDemo(it) {
                        val i = Intent(context,
                            ComposeActivity::class.java)
                        i.putExtra(KEY, it)
                        startActivity(i)
                    }
                }
            }
        }
    }
}

@Composable
fun ComposeDemo(value: Float, onClick: () -> Unit) {
    JetpackMigrationDemoTheme {
        Column(
            modifier = Modifier.fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Box(
                modifier = Modifier.fillMaxWidth()
                    .background(MaterialTheme.colors.secondary)
                    .height(64.dp),
                contentAlignment = Alignment.Center
            ) {
                Text(text = value.toString())
            }
            Button(
                onClick = onClick,
                modifier = Modifier.padding(top = 16.dp)
            ) {
                Text(text = stringResource(id = R.string.compose_activity))
            }
        }
    }
}
```

Listing 2: layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    tools:context=".ViewActivity">

    <com.google.android.material.slider.Slider
        android:id="@+id/slider"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <androidx.compose.ui.platform.ComposeView
        android:id="@+id/compose_view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/slider" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 3: Ein einfaches ViewModel

```
package eu.thomaskuenneth.jetpackcomposemigrationdemo

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

class MyViewModel : ViewModel() {

    private val _sliderValue: MutableLiveData<Float> =
        MutableLiveData<Float>(0.5f)

    val sliderValue: LiveData<Float>
        get() = _sliderValue

    fun setSliderValue(value: Float) {
        _sliderValue.value = value
    }
}
```

neuere `ViewBinding`. Anschließend kann man mit `setViewCompositionStrategy()` festlegen, wann Jetpack Compose benötigte Objekte – Composition ist ein zentraler Mechanismus – aufräumt. Wird die Methode nicht aufgerufen, greift `ViewCompositionStrategy.DisposeOnDetachedFromWindow`: Die Composition wird freigegeben, wenn der View vom Fenster, in dem die Activity angezeigt wird, entkoppelt wird. `setContent {}` setzt das anzuzeigende Compose-UI.

Für den Datenaustausch zwischen Views und Composable-Funktionen sollte man ViewModels verwenden. Sie sind vom Lebenszyklus von Activities unabhängig und überstehen Konfigurationsänderungen, zum Beispiel wenn man das Smartphone oder Tablet vom Hochkant- in den Quermodus dreht. Die Klasse `MyViewModel` (Listing 3) speichert den aktuellen Wert des Sliders in der privaten Eigenschaft `_sliderValue`. Lesend wird über `sliderValue` zugegriffen, Werte lassen sich mit `setSliderValue()` setzen. Das geschieht zum Beispiel im Callback, den man mit `binding.slider.addOnChangeListener()` setzt.

Listing 4: custom.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.textview.MaterialTextView
        android:id="@+id/textView"
        android:layout_width="0dp"
        android:layout_height="64dp"
        android:background="?colorSecondary"
        android:gravity="center"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <com.google.android.material.button.MaterialButton
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="@string/view_activity"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/textView" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Views und Layouts im Compose-UI

Änderungen am ViewModel überwacht man wie üblich mit `viewModel.sliderValue.observe()`. Im Beispiel wird dann der Slider auf den neuen Wert gesetzt. Wie aber erhält das Composable `Text()` den anzuzeigenden Text? Zunächst erzeugt man mit `viewModel.sliderValue.observeAsState()` eine `State<Float?>`-Instanz, übergibt sie dann an `ComposeDemo()` und verwendet sie dort per `text=value.toString()`. Dadurch findet bei Zustandsänderungen eine Recomposition (Neuzeichnen des UI) statt.

Der andere interessante Anwendungsfall ist, vorhandene Views in ein Compose-UI zu integrieren. Das bietet sich zum Beispiel bei selbst entwickelten Views an. Aus Gründen der Übersichtlichkeit führt das Beispiel das Vorgehen anhand eines einfachen Layouts vor (Listing 4). Die Klasse `ComposeActivity` zeigt mit `setContent {}` die Composeable Function `ViewIntegration`

`Demo()` an. Es setzt das Anwendungs-Theme und sorgt für einen App-Rahmen (`Scaffold()`) mit App Bar am oberen Rand.

Eine `Column()` enthält einen `Slider()` und einen Container für entfaltete Layouts,

```
AndroidViewBinding().factory = ↵
                    CustomBinding::inflate
```

entfaltet die Layoutdatei custom.xml und zeigt sie an. Den übergebenen Callback ruft die App nach dem Entfalten sowie nach Aktualisierungen auf. Der Compose-Slider setzt beim Verschieben den neuen Wert mit `viewModel.setSliderValue()` im ViewModel. Auf Änderungen in Letzterem reagiert das Programm durch `viewModel.sliderValue.observeAsState()`. `sliderValueState` ist ein Objekt des Typs `State<Float?>`. Compose-üblich führen Änderungen des Zustands zu Recompositions. Konkret wird mit der Zuweisung

```
value = sliderValueState.value ?: 0F
```

der neue Wert des Sliders gesetzt. Hinzu kommt: Auch eine Recomposition von `AndroidViewBinding()` findet statt, weil im Update-Callback mit

```
textView.text = sliderValueState.value.↵
                                toString()
```

der anzuzeigende Text geändert wird. Einzelne Views lassen sich mit `AndroidView()` einem Compose-UI hinzufügen.

Fazit

Google bietet mächtige Werkzeuge an, um bestehende Apps nach Jetpack Compose zu migrieren. Ob ein solcher Wechsel in jedem Fall sinnvoll ist, hängt vom Allgemeinzustand der App ab. Häufige Abstürze, ANRs (Application Not Responding) oder sich nicht richtig aktualisierende Bedienelemente sind Indikatoren, besser die Finger davon zu lassen. Denn selbstverständlich sorgt eine Mischung aus UI-Techniken für zusätzliche Komplexität.

Läuft die App hingegen halbwegs rund und wurden vielleicht sogar die wichtigsten Architekturratschläge der vergangenen Jahre umgesetzt – beispielsweise durch Nutzung der Android Architecture Components –, kann sich ein schrittweiser Umstieg durchaus lohnen. Dabei ist natürlich zu beachten, dass Jetpack Compose auf Kotlin basiert. Das App-Projekt muss also entsprechend konfiguriert sein. Ist das nicht der Fall, sollten Entwickler als Einstieg vermutlich andere Teile der Codebasis nach Kotlin portieren.

Beim Erstellen einer Migrationsstrategie spielt aber auch eine Rolle, dass Jetpack Compose zwar mittlerweile in einer ersten stabilen Version vorliegt, Google es aber trotzdem noch intensiv weiterentwickelt. Also ist Vorsicht angeraten: Programmierer müssen damit rechnen, dass sich an Compose selbst im Laufe der Zeit noch Dinge ändern. Das kann bedeuten, einen Umstieg vielleicht noch etwas aufzuschieben. Grundsätzlich aber gilt: Sofern die beiden UI-Welten zum Beispiel über ViewModels synchronisiert werden, klappt das Zusammenspiel von Jetpack Compose und Views sehr gut. (fo@ix.de)

Quellen

Alle Listings und die Beispiel-App:
ix.de/z6q5

Thomas Künneth

arbeitet als Head of Mobile für die MATHEMA GmbH. Neben zahlreichen Artikeln hat er drei Bücher über Android, Java und Eclipse veröffentlicht.