Jetpack Compose: ein Blick auf Androids neue UI-Technik

Gekonnt komponiert

Thomas Künneth

Mit Jetpack Compose springt Google für Android auf den Zug der modernen UI-Tools auf. Der deklarative Ansatz soll das Programmieren von Benutzeroberflächen vereinfachen.

Seit Ende Februar 2021 ist das neue Android-Toolkit Jetpack Compose Beta und damit stabil genug für eine ausführlichere Betrachtung (siehe ix.de/zunh). Das Tool setzt auf Kotlin und ist deklarativ: Entwicklerinnen und Entwickler beschreiben, wie die Oberfläche einer Android-App aussehen soll. Sie können die im Hintergrund vorhandenen baumartigen Strukturen ignorieren und den Umbau der Bibliothek überlassen.

Activities als Grundbausteine gibt es weiterhin. Compose setzt auf der Ebene von Bedienelementen (und Gruppen von diesen) an, tritt also in Konkurrenz zu allen Klassen im Paket android.view. Zentrales Element sind Composable Functions – mit `@Composable` annotierte Kotlin-Funktionen.

Die Benutzeroberfläche ist das Aushängeschild einer App. Ihre Bedienung muss flüssig von der Hand gehen: Alle Funktionen sollten zügig und ohne Umwege erreichbar sein. Nutzerinnen und Nutzer erwarten unmittelbar eine Rückmeldung, Verzögerungen kommen nicht gut an. Eine wichtige Rolle spielen heutzutage Animationen. Sie sollen Aktionen unaufdringlich, aber effektvoll in Szene setzen. Wie leicht sich all dies umsetzen lässt, hängt von der verwendeten (Klassen-)Bibliothek ab.

Seit der ersten Android-Version hat Google diese konsequent aktualisiert und erweitert. Und doch wirkt Androids View-basierter Ansatz angestaubt, denn die Architektur findet sich in den meisten UI-Toolkits der letzten 30 Jahre in ähnlicher Form wieder. Bedienoberflächen entsprechen hierbei Komponentenbäumen. Jede Komponente (View) ist über spezifische Eigenschaften konfigurierbar. Änderungen an den Eigenschaften manipulieren somit die Oberfläche und jede noch so kleine Modifikation ist auszuprogrammieren. Man nennt solche Frameworks deshalb imperativ.

Besser deklarative Frameworks nutzen

Das funktioniert im Kleinen sehr gut. Größere Anwendungen werden aber erfahrungsgemäß im Laufe der Zeit schwer wartbar. Zwar lassen sich mit etablierten Entwurfsmustern (Model-View-Controller, Model-View-ViewModel, Model-View-Presenter) Apps recht gut strukturieren, das anstrengende Synchronhalten von Daten und Oberfläche aber bleibt. Hier setzen deklarative UI-Frameworks an. Dabei beschreiben Entwicklerinnen und Entwickler den aktuellen Zustand der Benutzeroberfläche. Ändern sich Daten, aktualisiert das Framework betroffene Komponenten oder Strukturen im Hintergrund. Das Web war mit React Vorreiter. Apple und Google haben mit SwiftUI und Flutter nachgezogen. Android-Entwickler mussten am längsten warten.

Zentrales Element sind Composable Functions. Sie beginnen untypischerweise mit einem Großbuchstaben, weil sie Bedienelemente oder Teile davon konstruieren und an Datentypen erinnern. Eine einfache Compose-App zeigt Listing 1 (das App-Projekt zum Download unter ix.de/zunh).

iX-TRACT

- Mit dem UI-Toolkit Jetpack Compose modernisiert Google den Bau von Benutzeroberflächen für Android-Apps und folgt damit React, Flutter und SwiftUI.
- Google verlässt damit bei Android den klassischen imperativen Ansatz, Oberflächen zu programmieren, und wendet sich dem deklarativen zu. View-basierte Komponentenbäume gelten allmählich als veraltet.
- Kern von Jetpack Compose sind Composable Functions, mit denen sich die Oberfläche der App programmgesteuert definieren lässt. Dazu beschreibt man Form und Datenabhängigkeiten der Oberfläche, anstatt diese selbst zu erstellen.

`setContent {}` zeigt die Benutzeroberfläche der Activity an. Das Composable `Greeting()` bildet die Wurzel. Es ruft `androidx.compose.material.Text()` auf und bekommt den anzuzeigenden Text übergeben. Das unspektakuläre Ergebnis ist in Abbildung 1 zu sehen.

Benutzeroberflächen entstehen durch das Schachteln von Composables. Neben darstellenden Funktionen wie `Text()` gibt es auch solche, die strukturieren oder anordnen. Das LinearLayout in der alten View-basierten Welt ordnet Kindelemente horizontal oder vertikal an. Bei Jetpack Compose gelingt das mit `Row()` und `Column()`: `Column()` positioniert Kinder untereinander, `Row()` nebeneinander (Listing 2, Ausgabe in Abbildung 2). Mit `@Preview` kann man sich ein Composable ansehen, ohne die App installieren und starten zu müssen. Noch dauert das recht lange. Hier muss Android Studio an Performance zulegen, damit dieses Feature praxistauglich wird.

Klassische UI-Frameworks leiten alle Bedienelemente von einer Basisklasse ab. Diese definiert minimale Anforderungen an eine Komponente, zum Beispiel deren Position und Größe. Jetpack Compose geht hier viel pragmatischer vor. Es steuert solche Eigenschaften durch Modifier: `fillMaxSize()` bewirkt, dass ein Composable den größtmöglichen Platz einnimmt, `fillMaxWidth()` wirkt nur auf die Breite. Mit `weight()` lassen sich Kindern Größen in Relation zur Gesamtgröße zuweisen. Das funktioniert ähnlich wie beim alten LinearLayout.

Listing 1: Eine einfache Compose-App

```
package com.thomaskuenneth.ixdemo

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.Text
import androidx.compose.runtime.Composable

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContent {
        Greeting()
      }
    }
}

@Composable
fun Greeting() {
    Text(text = "Hello iX")
}
```

Modifier sind Universalwerkzeuge, denn sie wirken nicht nur auf Größe oder Position, sondern können beispielsweise Hintergründe anzeigen oder helfen, auf Scroll- und Wischgesten zu reagieren. Außerdem lassen sie sich einfach verketten. Um einen einfachen Text anklickbar zu machen und ihm einen Rahmen hinzuzufügen, genügt folgender Code:

```
Text(text = "Hello iX",
    modifier = Modifier
      .clickable { println("Hello iX") }
      .border(1.dp, MaterialTheme.colors.
                                        primary
      ))
```

Eigene Modifier sind schnell geschrieben. Die Funktion `drawOnYellow()` erweitert alle Objekte, die das Interface androidx.compose.ui.Modifier implementieren. Das Beispiel in Listing 3 ruft `drawBehind()` auf, ebenfalls eine Erweiterungsfunktion. Sie zeichnet hinter dem modifizierten Inhalt.

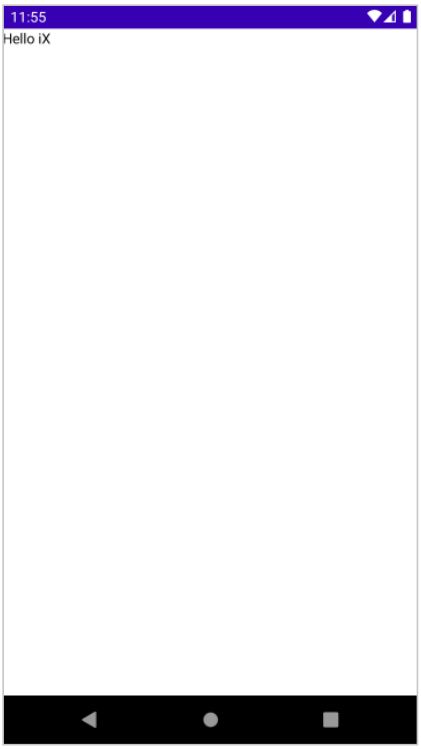
Auf Benutzeraktionen reagieren

Der Modifier `clickable()` erhält einen Lambdaausdruck und wird beim Anklicken ausgeführt. Normale Buttons funktionieren genauso. Allerdings ist der Button im folgenden Code benannt (`onClick =`):

```
Button(onClick = {
    println("clicked")
}) {
    Text(stringResource(id = R.string.
                                  click))
}
```

Um im Quellcode Strings nicht hart codieren zu müssen, kann man mit `stringResource()` auf die in der Datei strings.xml abgelegten Zeichenketten zugreifen. Mit Grafiken funktioniert das ähnlich. Der Buttontext wird hier allerdings nicht als einfacher String, sondern als Composable übergeben. Die Implementierung von `Button()` reicht den Lambdaausdruck als content-Parameter an eine `Row()` weiter (Listing 4). Das ermöglicht unter anderem mehrzeilige Schaltflächen.

In imperativen UI-Frameworks steuern Attribute wie `enabled`, ob eine Komponente anwählbar (aktiv) ist. Um sie umzuschalten, ändert man die Eigenschaft einer Instanz. Jetpack Compose löst dies über einen Parameter. Um den Zustand anzupassen, ändern Entwicklerinnen und Entwickler den Code so, dass das Composable erneut



Das Composable `Greeting ()` gibt einen Begrüßungstext aus (Abb. 1).

Das Composable strukturiert das Layout mit Zeilen und Spalten (Abb. 2).

aufgerufen wird. Wie das im Zusammenspiel mit anderen Composables abläuft, zeigt Listing 5.

Dabei geschieht eine ganze Menge. Eine `Column()` stellt zwei Composables, ein `TextField()` und einen zentrierten `Button()`, untereinander dar. Wird die Schaltfläche angeklickt, gibt `println()` den Wert `input.text` aus. `input.text.isNotEmpty()` steuert, ob der Button anklickbar ist. Das Objekt `input` ist vom Typ `TextFieldValue`, das man im Callback `onValueChange` des Textfeldes neu setzt. Initialisiert wird es allerdings durch die Zeile

```
var input by remember { mutableStateOf(
                            TextFieldValue()) }
```

`remember` liefert beim ersten Aufruf einen berechneten Wert, nämlich `mutableStateOf(TextFieldValue())`. Alle weiteren Aufrufe produzieren dann immer wieder dasselbe Ergebnis. `mutableStateOf()` liefert ein Objekt des Typs `MutableState`, das mit dem übergebenen Wert – eine `TextFieldValue`-Instanz – initialisiert wurde. Der Variablen `input` wird aber nicht direkt der Rückgabewert des Funktionsaufrufs `remember {}` zugewiesen, sondern – aufgrund des `by` – der Wert der Eigenschaft `value` des erinnerten `MutableState`-Objekts. Die Zeile erzeugt also beim ersten Mal ein `TextFieldValue` und weist es `input` zu. Später erhält `input` den zuletzt geänderten Wert. Alternativ kann man auch `val input = remember {}` schreiben, muss dann aber bei Zugriffen `input.value` verwenden. `by` macht den Code also kompakter.

Um zu verstehen, warum das passiert, muss man ein Grundprinzip von Jetpack Compose kennen: Es erzeugt Oberflächen einmalig (Composition) und aktualisiert sie bei Zustandsänderungen (Recomposition). Solche Zustandsänderungen erfolgen zum Beispiel nach Benutzereingaben. Aber auch das Ergebnis eines Webserviceaufrufs kann zur Neukomposition des UI führen – wenn sich eine von einem Composable erinnerte Variable ändert.

Ein weiteres Grundprinzip besagt: Recompositions können und sollen oft stattfinden. Dazu müssen Composable Functions schnell sein. Langwierige Berechnungen oder gar Datei- und Netzwerkzugriffe sind tabu. Sofern möglich, übergibt man alle benötigten Informationen via Parameter. Das können normale Datentypen oder Zustände sein. Das ist sinnvoll, wenn der Zustand Auswirkung auf mehrere Composables hat. Dann befindet sich der Aufruf `remember {}` im Eltern-Composable.

Im Nu erzeugt: Elemente in einer Liste auswählen

Ein Problem des alten View-Systems ist, dass manche Dinge unnötig kompliziert sind. Hierzu gehört das Anzeigen und Auswählen von Elementen in einer scrollbaren Liste. Tatsächlich lässt sich dies in Compose mit wenigen Zeilen Quelltext lösen (Listing 6). Der wichtigste Baustein ist hierbei das Composable `LazyColumn()`. Es stellt seine Elemente untereinander dar. Nur die sichtbaren Teile sind zu bauen und zu layouten. Die darzustellenden Elemente lassen sich unter anderem mit `itemsIndexed()`

Listing 2: Composables anordnen

```
@Preview
@Composable
fun LayoutDemo() {
    Column(modifier = Modifier.fillMaxSize()) {
      Row(
        modifier = Modifier
          .fillMaxWidth()
          .weight(0.5F)
      ) {
        Text(text = "1", modifier = Modifier.weight(0.3F))
        Text(text = "2", modifier = Modifier.weight(0.7F))
      }
      Row(
        modifier = Modifier
          .fillMaxWidth()
          .weight(0.5F)
      ) {
        Text("3")
        Text("4")
      }
    }
}
```

Listing 3: Mit drawOnYellow ein gelbes Rechteck unter Composables setzen

```
fun Modifier.drawOnYellow() = this.drawBehind {
    drawRect(color = Color.Yellow)
}

Text(
    modifier = Modifier.drawOnYellow(),
    text = "Hello Compose"
)
```

Listing 4: Ein zweizeiliger Button

```
Button(onClick = {
    println("clicked")
}) {
    Column {
      Text("Zeile 1")
      Text("Zeile 2")
    }
}
```

Listing 5: Mit onValueChange auf Texteingaben reagieren

```
@Composable
fun TextFieldDemo() {
    var input by remember { mutableStateOf(TextFieldValue()) }
    Column(
      modifier = Modifier.fillMaxWidth(),
      horizontalAlignment = Alignment.CenterHorizontally
    ) {
      TextField(
        modifier = Modifier.fillMaxWidth(),
        value = input,
        onValueChange = {
          input = it
        }
      )
      Button(
        onClick = {
          println(input.text)
        },
        enabled = input.text.isNotEmpty()
      ) {
        Text(stringResource(id = R.string.click))
      }
    }
}
```

Listing 6: Eine scrollbare Liste

```
@Composable
fun ListDemo() {
    val callback = { index: Int -> println("$index selected") }
    val list = arrayListOf("1", "2", "3")
    LazyColumn(
      modifier = Modifier.fillMaxSize(),
      contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp)
    ) {
      itemsIndexed(list) { index, s ->
        Text(
          text = s,
          modifier = Modifier
            .fillMaxSize()
            .clickable { callback(index) },
          style = MaterialTheme.typography.subtitle2
        )
      }
    }
}
```

Listing 7: Animationen einfach umsetzen

```
@ExperimentalAnimationApi
@Composable
fun AnimationDemo() {
    var visible by remember {
      mutableStateOf(false)
    }
    Column(
      modifier = Modifier.fillMaxSize(),
      horizontalAlignment = Alignment.CenterHorizontally
    ) {
      Button(onClick = {
        visible = !visible
      }) {
        Text(stringResource(id = R.string.click))
      }
      AnimatedVisibility(
        visible = visible
      ) {
        Text(
          text = stringResource(id = R.string.app_name),
          style = MaterialTheme.typography.h1
        )
      }
    }
}
```

erzeugen. Das ist praktisch, wenn wie im Beispiel die Position innerhalb der Liste benötigt wird. Listenelemente sind letztlich Composables. Anstelle eines einfachen `Text()` kann man mit `Column()` mehrzeilige Einträge konstruieren.

Jetpack Compose beinhaltet `androidx.compose.material.ListItem()`, es ist aber als experimentell gekennzeichnet. Etwas ungewohnt ist, dass jedes Element mit `clickable {}` auf die Selektion reagiert. Es bietet sich an, jedes Mal denselben Callback zu übergeben.

Animationen spielen in modernen Apps eine wichtige Rolle. Deshalb verwundert es nicht, dass Jetpack Compose Entwicklern auch in diesem Punkt das Leben einfach machen möchte. Das Ein- und Ausblenden von Composables ist schnell erledigt. Wie es geht, zeigt Listing 7 (siehe auch Abbildung 3). Allerdings ist die API noch nicht final. Um beispielsweise `AnimatedVisibility()` verwenden zu können, gilt es, die Annotation `@ExperimentalAnimationApi` bei allen aufrufenden Composables zu setzen.

`AnimatedVisibility()` lässt beim Erscheinen und Verschwinden seines Inhalts (das wird über das Attribut `visible` gesteuert) EnterTransitions und ExitTransitions ablaufen. Hiervon gibt es drei Arten: Fade, Expand/Shrink und Slide. Diese lassen sich mit + kombinieren. Die Reihenfolge spielt dabei keine Rolle, weil die Animationen simultan beginnen.

Fazit

Jetpack Compose löst den View-basierten Teil der Android-Klassenbibliothek ab. Er wird aber nicht verschwinden. Entwicklerinnen und Entwickler müssen also keinesfalls sämtliche Apps umschreiben. Es ist aber damit zu rechnen, dass Google Views und ViewGroups nach und nach für veraltet erklärt. Auch Fragmente dürften relativ schnell an Bedeutung verlieren, da sie einst für die Verwendung mit Layoutdateien und View-Hierarchien entwickelt wurden. Bei neuen Apps sollte man deshalb sehr genau überlegen, ob man noch einmal auf alte Pferde setzt. Google jedenfalls rührt kräftig die Werbetrommel für Jetpack Compose und die anderen Bestandteile von Jetpack. Eines ist bereits klar: Überschaubarer wird die Android-Entwicklung nicht. Entwickler müssen eine weitere Technologie auf dem Schirm haben. (nb@ix.de)

In der Beispiel-App iXDemo lassen sich Animationen einfügen (Abb. 3).

Quellen

App-Projekt und Infos zur Betaversion: ix.de/zunh

Thomas Künneth

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