Mobile Programming

`I.BA_MOBPRO.F25` — Zusammenfassung

Author(s) Dominic, Elias, Laura, Lukas
Date 16. Mar. 2026
Pages 70

Inhaltsverzeichnis

1. Entwicklungsansätze

1.1. Optionen zur Mobile-App-Enwticklung

Web HTML, CSS, JS im Browser

Hybrid Webapp in nativen Wrapper verpackt, connector Plugins für native funktionen, z. B. Cordova

Cross-compiled In anderer Sprache geschrieben, die nach Swift/Kotlin oder direkt binär kompiliert, z. B. Xamarin, Flutter

JIT-Compiled / VM In Javascript geschribene App, die auf der JS-Engine des Zielsystems läuft und Just-in-time kompiliert wird, beispielsweise NativeScript oder Flutter

Native App Spezifisch für die Zielplattform entwickelt, Kotlin für Android, Swift für IOS

1.1.1. WebApp

- Eine Web-Anwendung für mehrere/alle Plattformen
- Voraussetzungen pro Plattform: Web-Browser
- Verwenden meist JavaScript-Web-Frameworks, z.B. Ionic, jQuery Mobile, Titanium, Sencha Touch, Angular UI

```diff
+ Eine Code-Basis für mehrere/alle Plattformen
+ Kein App-Store, Download oder Installieren notwendig
+ App kann jederzeit veröffentlicht/verändert werden
+ Vorhandene Web-Seite kann zu mobiler App erweitert werden (Stichwort „Responsive Web Design“) bzw. "Progressive Web App"
- Läuft im Browser („Rahmenapplikation“)
- Kein Zugriff auf alle Funktionen (Kontakte, ext. HW, ...)
- App nicht im Store, Benutzer müssen sie finden (Marketing!)
```

1.1.2. Hybride Apps

- Eine Web-Anwendung für mehrere/alle Plattformen, verpackt in einen nativen Container
- Verwenden typischerweise Web-Frameworks
- Verpackte Webapp, die in einer WebView läuft und durch einen nativen Container installierbar wird
- Zugriff auf native Features durch Plugins
- Voraussetzungen pro Plattform: Nativer Container, z. B. Apache Cordova

```diff
+ Eine Code-Basis für alle Plattformen (grundsätzlich)
+ App-Stores: App findbar, gewisse App-Qualität + Sicherheit
+ Zugriff auf viele (alle?) Geräte-Funktionen durch native APIs
~ Nativer Look&Feel? (Je nach Technologie!..)
- Kein garantierter Zugriff auf alle Geräte-Funktionen
- Unterstützung versch. Plattformen kann aufwändig sein, ggf. spezifischer Code / Anpassungen notwendig
```

1.1.3. Progressive Web App

- „Installierbare Web-App“
- Zwischending: (native) App - Web-App
- Technologisch: Service-Workes, Web Storage, offline-fähig, deklariert in Web-App-Manifest
- Standard? Unterstützung (iOS, Firefox, ...)?

1.1.4. Cross-Compile, JIT-Compile

- Ansatz = „Cross-Plattform“
- Eine Codebasis für alle Plattformen mittels Cross-Compile
- Keine/wenig Kenntnis der nativen Plattform nötig
- Abstraktion, eigenes App-Modell
- Grösster gemeinsamer Nenner
- Installierbar und im Store

- Nativer Look
- Cross-Compile = Nach Programmierung Umwandlung in nativen Code oder Binary
- JIT-Compile = Läuft auf JS-VM auf Zielplattform, just-in-time Kompilierung (JS-VM ist heute auf jeder Platform vorhanden und i.d.R. up-to-date/modern/schnell via Plattform-Updates)

```diff
+ Eine Codebasis (Funktionalität nur 1x implementieren)
+ Vorhandenes Entwicklerknowhow (Web, C#, …) wiederverwenden, keine Einstiegsbarriere
+ Geringere Entwicklungskosten
+ Sieht aus wie native und ist gleich schnell (kein Geflicker)
~ Eigenes Applikationsmodell != native Modell
- Eingeschränktes Programmiermodell: "Grösstes gemeinsames Vielfaches"
- Native Features uneingeschränkt nutzbar, aber aufwändig
- Ggf. schwierig zu Debuggen
```

1.1.5. Native App

- Anwendung wird in von der Plattform vorgegebener Programmiersprache (inkl. vorgegebene APIs) entwickelt
- pro mobiler Plattform (Android, iOS, …) wird grundsätzlich eine eigenständige App entwickelt
- Voraussetzungen pro Plattform: Natives API + SDK

```diff
+ Zugriff auf alle nativen Möglichkeiten
+ Natives Look'n'Feel
+ Typischerweise schnellste Lösung
+ App-Stores: App auffindbar, gew. App-Qualität + Sicherheit
- Grosser Entwicklungsaufwand (1 App pro Plattform), d.h. Unterstützung mehrere
Plattformen ist teuer, gleiche Funktionalität 2x oder mehr entwickeln, kein Code-Sharing
- Benutzer können verschiedene Versionen haben: ggf. aufwändig zu unterstützen
- App-Store-Zulassung kann Veröffentlichung verlangsamen oder verhindern (im vergl. zu
reinen Web-Apps)
```

2. Kotlin Intro 🚀

- Sprache aus dem Hause JetBrains
- relativ „junge“ Sprache
  - 2016: v1.0
  - 2017: Offizielle Android-Sprache
  - 2019: Offizielle Haupt-Android-Sprache
- Es läuft wie Java in der JVM
- Compilation nach *JavaScript* oder *Java* möglich
- 100 % Interoperabilität mit Java
- Open-Source unter Apache 2.0

2.1. Java Ähnlichkeiten

- Selbe Basisoperatoren (`+, -, *, <, =`)
- Ähnliche Kontrollstrukturen (if-else, for-in, while, **when**)
  - `when` ersetzt das `switch` in Java
  - Ausdrücke haben einen Rückgabe-Wert, so wie z. B. die Switch-Expression in Java
- OOP-Keywords (`class` , `interface` , access-modifiers, `final` , ...)
- Lambda-Expressions (`val sum: (Int, Int) -> Int = { x, y -> x + y }`)
- Exception-Handling mit `try ... catch`
- `Any` als Wurzeltyp, nicht `Object` wie bei Java

2.2. Java Unterschiede

- Veränderbarkeit einer Variable wird durch das Keyword `val` oder `var` festgelegt
  `val` Ein *statischer* Wert, welcher nach der Initialisierung nicht mehr verändert werden kann
  `var` Ein *variabler* Wert, welcher angepasst werden kann
- Es gibt **keine** primitiven Datentypen (`int` , `double`), sondern nur Klassentypen (`Int` , `Double`)
  - Kotlin daher „reinere“ Objektorientierte Sprache als Java
  - Klassen wie `IntArray` werden jedoch auf der JVM trotzdem als `int[]` dargestellt, um von der gesteigerten Effizienz profitieren zu können
- Es gibt **keine** checked exceptions
- Es werden **Companion Objects** statt statischen Methoden und Attributen verwendet
  - Definition von Funktionen und Properties auf Klassen-Ebene
  - Verwendung des `companion` -Keywords
  - Siehe auch Kotlin Docs zu Companion Objects
- Generics ohne Wildcard-Typen, sondern mit „Declaration Site Variance“ und „Type Projection“
  - Ein Typenparameter eines generischen Interfaces kann beispielsweise mit dem Keyword `out` annotiert werden, um zu signalisieren, dass dieser Typ *nur* als Rückgabe-Wert verwendet wird
  - Das `in` Keyword signalisiert dasselbe, jedoch einfach darauf bezogen, welche Typen als Parameter an eine Methode dieses Interfaces übergeben werden können
  - Siehe auch Kotlin Docs zu Generics

2.3. Attraktive Spracheigenschaften

- Kein Strichpunkt am Ende der Zeile / des Ausdruckes
- Properties und Fields bieten kompakte Syntax für Java-Instanzvariablen und Access-Modifiert → weniger Boilerplate 🥳
- Per Default keine Vererbung möglich, explizite Erlaubnis durch `open` Keyword
- DataClasses erlauben komplette Klassendeklarationen mit automatischen `equals` , `hashCode` und `toString` Methoden → ähnlich wie Java Records
- Object Expressions und Object Declaration erlauben direktes anlegen von Objekten und Singletons (ähnlich zu Anonymen Java Klassen)
- Delegation & Delegated Properties als Alternative zur Vererbung von Methoden bzw. Properties
- LINQ-artiger (Language integrated Query) Code wie in C# möglich
  - `strings.filter { it.length == 5 }.sortedBy { it }.map { it.toUpperCase() }`

- Type-Safe-Builders wie Jetpack Compose als vom Compiler geprüfte Builders für eigene DSLs
- Destructing Declarations
  - Dekonstruktion von Objekten: `val (name, age) = person`
- Coroutinen als Möglichkeit zur asynchronen Programmierung

2.4. Ausgewählte Spracheigenschaften

2.4.1. Nullability

Eine Variable kann per Default nicht `null` sein. Möchte man dies erlauben, so muss der Typ explizit mit dem `?` Operator entsprechend annotiert werden.

```kt
var some: String = "Test"
some = null                 // Compile-Time-Exception

var other: String? = "Test"
other = null                // OK
```

Dabei gibt es folgende 4 Möglichkeiten, auf nullable-Variablen zuzugreifen

```kt
// if
val len = if (nullable != null) nullable.length else -1 // Wert oder -1

// Safe Call Operator ?
print(nullable?.length)            // Gibt wert oder sonst "null" zurück

// Not-null Assertion Operator !!
val len = nullable!!.length        // Konvertierung in non-nullable type
                                   // falls null -> Nullpointer Exception

// Elvis Operator ?:
val len = nullable?.length ?: -1   // Wie fall 1 einfach kürzer
```

2.4.2. Benannte Funktionen und Default Werte

Parameter einer Funktion können in Kotlin benannt und mit Default-Werten versehen werden.

```kt
fun join(s1: String, s2: String, joiner: String = " "): String {
  return s1 + joiner + s2
}

// Aufruf mit 3 benannten Params
join(s1 = "HSLU", s2 = "I", joiner = "-")

// 2 benannte Params, Default-Wert vom 3. Param.
join(s1 = "I", s2 = "HSLU")

// Kompilier-Fehler: s2 hat keinen Default-Wert
join(s1 = "HSLU")

// Bel. angeordnete benannte Params
join(s2 = "HSLU", joiner = "-", s1 = "I")

// Kompilier-Fehler: Mischen von benannten
// und positionalen Parametern geht nicht
join(s2 = "HSLU", "-", s1 = "I")
```

2.4.3. Klassen und Konstruktoren

```kt
// Klassendefinition inkl. constructor annotation
class ConstrDemo constructor (val s: String, i: Int) {
  private var d: Double
  private val iTwice = 2 * i

  // Initializer Block für den default constructor
  init { d = 4.2 }

  // Secondary Constructor, delegation mit 'this' zum Primären
  constructor(d: Double) : this("hu", 2) {
    this.d = d
  }

  fun run() {
    val cd1 = ConstrDemo("hi", 42)
    val cd2 = ConstrDemo(2.4)
  }
}
```

2.4.4. Erweiterung ohne Vererbung

In Kotlin ist es möglich, eine bestehende Klasse durch eine neue Methode zu erweitern.

```kt
fun Int.prettyPrint() {
  println("The number is $this")
}

42.prettyPrint() // Gibt 'The number is 42' aus
```

- Erweiterung auch auf „fremden“ Klassen wie Companion Objects möglich
- Extensions auf Funktionen und Properties möglich
- Statische Typauflösung zur Compile-Time

2.5. Kotlin und Android

Native Android-Apps können seit einiger Zeit neben Java auch in Kotlin entwickelt werden. Seit einigen Jahren ist Kotlin dabei die empfohlene Sprache für die Androidentwicklung.

2.5.1. Gründe für Kotlin

- Kompakt und ausdrucksstark
- schnellere Entwicklung
- Weniger code (ca. 40% nach Google)
- Typ- und Null-safety
- Interoperabilität mit Java
- Automatische Code-Konvertierung von Java zu Kotlin
- **Jetpack** als Möglichkeit zum Erstellen von Android-Apps
- Android KTX als Sammlung von Kotlin Extensions, welche in Android Jetpack enthalten sind und im Package `androidx.*` enthalten sind

3. Android Grundlagen 🧰

3.1. Grundlagen einer Android Applikation

- Apps sollen grundsätzlich gegen das aktuellste API entwickelt werden
- Mindestanforderung ans SDK kann im Build-Skript gesetzt werden: minSDK
- Sollte tiefer sein als targetSdk, Orientierung an der Verbreitung

3.1.1. App - Komponenten (AK)

Android besteht aus vier Komponenten (heut nicht mehr alles gleich relevant).

- **Activities**: Bilden die UI und interagieren mit dem Nutzer
- **Services**: Hintergrundprozesse ohne UI
- **Broadcast Receivers**: Empfangen System-Events
- **Content Providers**: Teilen Daten zwischen Apps

3.1.1.1. AK -Activity

- App besteht meist aus einer Main-Activity
- entspricht einer main Methode
- eine Activity entspricht einem Screen
- Eine aufgerufene Activity kann Resultate zur vorangegangen Activity zurückliefern.
- Basisklasse: `android.app.Activity`

3.1.1.2. Manifest

- File namens `AndroidManifest.xml`
- Definiert Komponenten, Rechte (Permissions) und Intent-Filter (Einschränkungen für Aufruf).
  - alle Komponenten einer Applikation müssen dem System bekannt gegeben werden
- Enthält Metadaten zur App
- Infos werden bei der App-Installation im System registriert
- Zusätzliche Informationen (Version, ID, etc.) befinden sich im Build-Skript

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="DemoApplication"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.DemoApplication"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:theme="@style/Theme.DemoApplication">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Abbildung 1: Beispiel `AndroidManifest.xml`

- `application tag` : Beschreibt die Applikation
- `permission` : wären zwischen manifest und application
- `intent filter` : sagt dem system welche app gestartet werden muss, wenn sie geöffnet wird (wird im Projekt generiert)

3.2. Lifecycle

- Das System kann eine Applikation ohne Vorwarnung terminieren, wenn der Speicher knapp wird
  - Nur Activities im Hintergrund
  - Von User unbemerkt, bei Zurücknavigation wird Applikation wiederhergestellt

D.h. eine Applikation kann ihren Lebenszyklus nicht kontrollieren und muss daher u.a. in der Lage sein, ihren Zustand zu speichern und wieder zu laden.

3.2.1. Lifecycle-Activity

- Eine Aktivität ist eine einzelne, fokussierte Sache, die der Benutzer tun kann.
- Aktivitäten werden im activity stack verwaltet (last in, first out (LIFO))
  - Siehe auch Tasks and Back-Stack (Android Docs)

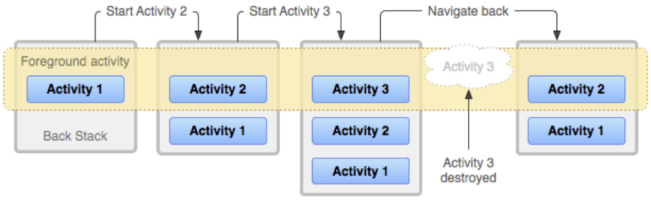

Abbildung 2: activity stack

- eine Aktivität hat im Vordergrund vier states
  - `active` / `running` : ist im Vordergrund eines Screens
  - `visible` : Aktivität hat den Fokus verloren, ist aber immer noch sichtbar für den user
  - `stopped` / `hidden` : Aktivität wird vollständig von einer anderen Aktivität verdeckt.
  - `destroyed` : System kann die Aktivität beenden

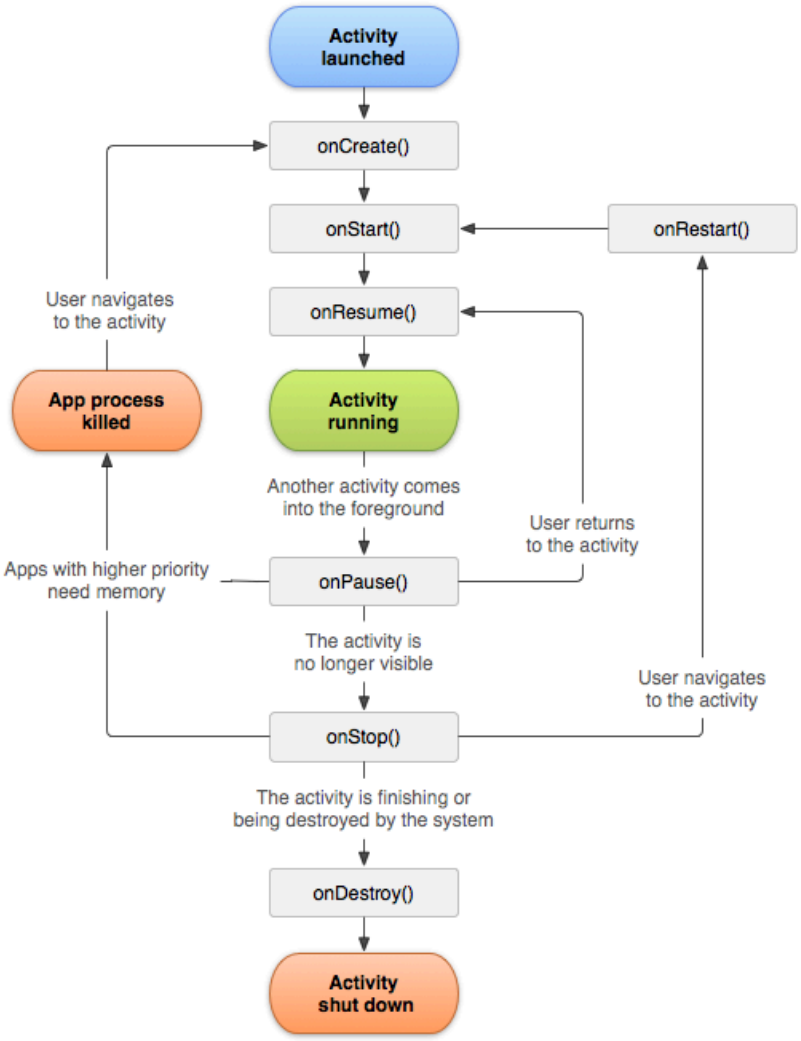

Abbildung 3: Lifecycle Activity

- Durch Überschreiben entsprechender Methoden kann in den Lebenszyklus eingeklinkt werden.
- Immer `super()` aufrufen (Exception)
- müssen nur überschrieben werden, falls default nicht ok ist

Allgemein: Da mit Compose programmiert wird, ist der Lifecycle nicht mehr so relevant (mit xml wichtiger).

3.3. Android Studio

- default build system: Gradle
- Allgemeine Build-Configs in der Regel in `app/build.gradle`

```
android {
    namespace = "ch.hslu.demoapplication"
    compileSdk = 35

    defaultConfig {
        applicationId = "ch.hslu.demoapplication"
        minSdk = 28
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"
```

Abbildung 4: gradle file

- `application ID` : App Identifikation ist einmalig und kann nur geändert werden, wenn eine neue App releaset wird
- `VersionCode` : muss nach einem Update höher sein, wird von PlayStore überprüft

- `VersionName`: wird nur angezeigt, nicht relevant für PlayStore

3.3.1. SDK-Versions

`targetSdkVersion` Für diese Version ist App gebaut, diese API-Version kann voll ausgeschöpft werden

`minSdkVersion` Die tiefste Version, auf welcher diese App lauffähig ist

- Bedingt Einschränkung bei verwendeten APIs oder Verwendung von Support-Library (Backports)
- App wir nicht angezeigt im Store für Geräte mit < Version oder kein update ausgerollt

`compileSdkVersion` Diese Version wird verwendet, um die App (sprich die APK-Datei) zu erstellen. Diese App kann Funktionen von der API nutzen, welche diesem Level oder tiefer entsprechen.

- Entspricht typischerweise der targetSdkVersion

3.4. Jetpack Compose & Navigation

3.4.1. Composables

Abbildung 5: Beispiel

- Nehmen Daten entgegen (aktueller UI-Zustand)
- Keine Rückgabe: „Compose functions that emit UI do not need to return anything, because they describe the desired screen state instead of constructing UI widgets“
- Schnell: werden (potentiell) oft neu gezeichnet
- Idempotent: verhält sich gleich, wenn mehrmals mit denselben Argumenten aufgerufen
- Brauchen keine Instanz um aufgerufen zu werden (nicht teil einer Klasse)
- Keine Seiteneffekte: verändert keinen globalen Zustand o.ä.

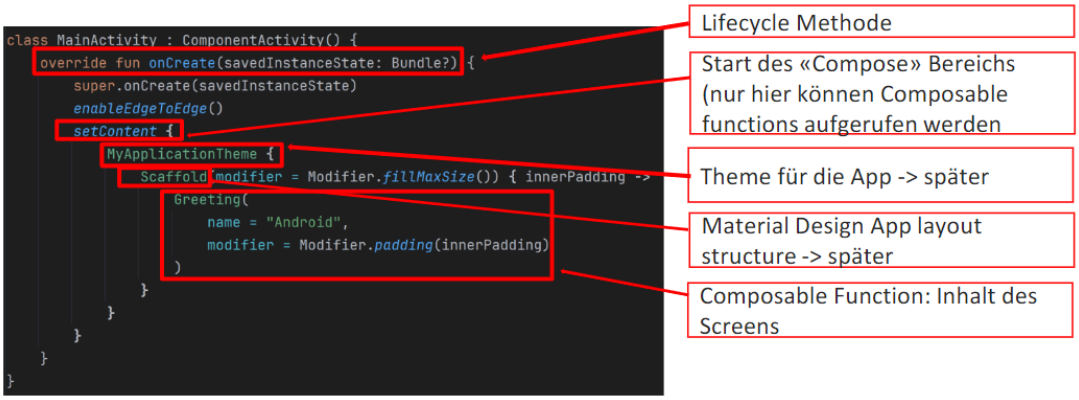

Abbildung 6: Main Activity

3.4.2. Navigation

- Jede Navigation wird auf dem „Back-Stack“ abgelegt
- Beim ausführen der „Zurück“ geste oder betätigen der „Back“-Taste wird der Back-Stack abgearbeitet

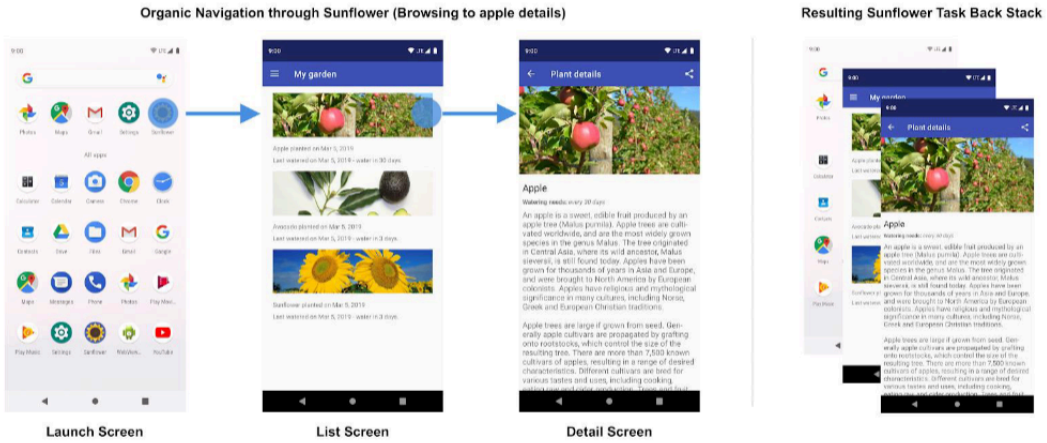

Abbildung 7: activity stack

3.4.2.1. Navigation Komponenten

- `NavHostController`
  - Ein Controller mit dem navigiert werden kann
  - back stack wird verändert
  - kann im Compose Kontext geladen werden `rememberNavController()`
- `NavHost`
  - Definiert Routen und Screens
  - Braucht den `NavHostController`

3.5. Zusammenfassung

- function im `@Composable` können von überall aufgerufen werden, compasable function werden grossgeschrieben
- geschweiftel Klammern sind funktionsimplementierungen
- mit `$` auf Attribute zugreifen
- Ctrl + Alt + L : formatierung
- Ctrl + Alt + O: löscht unbenutzte Sachen

4. Compose UI Styling 🖌️

4.1. Trailing Lambdas

- Kotlin kennt Function Types
- Konsistente und „sinnvolle“ Syntax für Funktionstypen
- Keine Annotation durch `@FunctionalInterface` wie in Java
- Werden oft als Argument für Higher-Order-Functions wie `filter()` oder `map()` verwendet
- **Trailing Lambdas** beschreiben dabei die Eigenschaft, dass das letzte Argument einer Funktion ein Funktionstyp ist, dies nach der Parameterliste in geschwungenen Klammern geschrieben werden kann

```kotlin
// Explizite angabe von In- und Ouput Parametern
val double: (Int) -> Int = { x: Int -> 2 * x }

// Verwenden von Typinferenz
val double = { x: Int -> 2 * x }

// Verwenden des implizierten Parameters 'it'
val double: (Int) -> Int = { 2 * it }

// Verwendung von Trailing Lambdas
val res = listOf<Int>(42, 7, 0)
  .filter {it > 0}
  .map { it * it}
  .fold(0) { a, b -> a + b}
```

4.2. Compose Theme

- Theme legt Style für eine App fest
- Farben, Fonts und Grössen können festgelegt werden
- Theme wird für die App einmal global gesetzt und nimmt dann als letztes Argument `@Composables` entgegen

```kt
AppTheme {
  Scaffold(
    modifier = Modifier.fillMaxSize(),
  ) {
    // Weitere Components
  }
}
```

- Ein Theme hat in der Regel Default-Werte, welche greifen, sofern nichts Weiteres spezifiziert wurde

4.2.1. Farben

- Farben z. B. basieren auf dem System-Wallpaper
- Können überschrieben werden
- Für ältere Android-Versionen muss ein Color Scheme bereitgestellt werden

4.2.2. Typografie

- Wird in `Type.kt` festgelegt
- 15 Standardstile für Text (`display` , `headline` , `title` , `body` , ...)
- Es wird empfohlen, die Default-Werte zu verwenden und nur diejenigen zu überschreiben, für welche man einen speziellen Anwendungsfall hat

4.2.3. Anwendung

- Ein Composable kann über Attribute wie `style` oder `color` in seinem Aussehen beeinflusst werden

```kt
@Composable
fun SomeScreen() {
  Text(
    text = "Lorem",
    style = MaterialTheme.typography.headlineMedium,
    color = MaterialTheme.colorScheme.primary
  )
}
```

4.3. Layouts

- Wird verwendet, um Komponenten anzuordnen und diese in Beziehung zueinander zu setzen
- Im Wesentlichen 3 Typen
  Column Anordnen von Komponenten in einer Spalte
  Row Anordnen von Komponenten in einer Zeile
  Box Anordnen von Komponenten mit teilweiser Überlappung
- Für die Child-Elemente gibt es ein Arrangement und ein Alignment
  Arrangement Gibt an, wie die Kinder an der Hauptsache ausgerichtet werden, z. B. in der Mitte mit (`Arrangement.Center`)
  Alignment Git ab, wie die Elemente auf der Gegenachse ausgerichtet werden

4.3.1. Column

Mit `Column()` werden Komponenten *vertikal* ausgerichtet. Die Haupt-Achse bei einer Column ist **vertikal**.

`verticalArrangement` vertikale Ausrichtung, z. B. `Arragement.SpaceBetween`
`horizontalAlignment` horizontale Orientierung, z. B. `Alignment.End`

4.3.2. Row

Mit `Row()` werden Komponenten *horizontal* ausgerichtet. Die Haupt-Achse bei einer Row ist **horizontal**.

`horizontalArrangement` horizontale Ausrichtung, z. B. `Arrangement.SpaceAround`
`verticalAlignment` vertikale Orientierung, z. B. `Alignment.CenterVertically`

4.3.3. Box

Mit `Box()` können Elemente übereinander gelegt werden.

- Wir z. B. bei Text auf einem Bild verwendet
- In der Regel weniger relevant als `Row()` und `Column()`

4.4. Modifier

- Attribut jeder Komponente
- Wird verwendet, um deren Aussehen zu beeinflussen
- Verarbeiten von Nutzereingaben oder Labels für Barrierefreiheit
- Interaktionen zu Elementen hinzufügen
- Verkettung von Modifikatoren möglich (Fluent API)
- Reihenfolge von Modifikatoren kann eine Rolle spielen
- Beispiel: Wir wollen abstände in einer `Column()` definieren

```kt
Column(
  modifier = Modifier.padding(16.dp)
) {
  // Items der Column
}
```

- Oft verwendete Modifier sind

  Alignment Ausrichtung eines Elements, z. B. nach rechts (`Modifier.align(Alignment.End)`)

  Fill Den gesamten verfügbaren Platz einnehmen, z. B. mit `Modifier.fillMaxSize()`

  Weight Wie viel Platz eine Komponente im Vergleich zu einer anderen Komponente in einem Container in Relation zum gesamten verfügbaren Platz einnehmen darf (`Modifier.weight(1.0f)`)

4.5. Sized und Dimensions

- Auflösung der Geräte in „Pixel per Inch“ unterschiedlich
- Für breite Geräteunterstützung Pixel deshalb ungeeignet
- Unterscheidung zwischen *Density Pixel* (`dp`) und *Scalable Pixel* (`sp`)
  - Density Pixel für
    - Abstände
    - Grössen
  - Scalable Pixel
    - Für Schriftgrössen
- Die Grössen werden dabei in der Regel jeweils in 4er-Schritten gemacht
- Im Allgemeinen sollte jedoch bei der Grössenangabe von Komponenten die Angabe von fixen Grössen verhindert werden und stattdessen auf Möglichkeiten wie `fillMaxWidth()` und dem `weight()`-Modifier gearbeitet werden.

5. Nebenläufigkeit und Kommunikation ⚡

5.1. Compose Preview

- `@Preview` Annotation
- `composable` functions wird in der Designansicht angezeigt
- Konfigurierbar
- Live Updates bei Änderung ohne die App auszuführen
- `@Preview` kann mehrmals der selben Funktion hinzugefügt werden um die Vorschau mit verschiedenen Properties zu sehen
- Es wird eine separate Funktion erstellt mit der `@Preview` Annotation und die Composable Funktion aufgerufen (kann auch direkt im Main erstellt werden).

5.1.1. Vorteile

- bei grossen Projekt dauert es mit dem Emulator zum Teil sehr lange
- kurze Anpassungen vergleichen (z.B. für verschiedenen Sprachen)
- verschiedene Screengrössen vergleichen
- usw.

5.1.2. Konfigurationsoptionen

- `showBackground` (Boolean)
  - zusätzlich `backgroundColor` für Hintergrundfarbe
- `device` : Auswahl eines Geräts
- `widthDp` und `heightDp` : Grösse definieren
- `locale` : Sprache für den Screen

```kt
@Preview
@Composable
fun PreviewHomeScreen() {
    HomeScreen("Android", rememberNavController())
}
```

```kt
@Preview(showBackground = true, device = Devices.PIXEL)
@Preview(widthDp = 500, heightDp = 500, locale = "de-CH")
@Composable
fun PreviewHomeScreen() {
    HomeScreen("Android", rememberNavController())
}
```

5.2. Nebenläufigkeit

5.2.1. Blockierungsproblem

- per default läuft die Applikation in **einem** Thread - im **main-Thread**
  - in diesem main-Thread wird das ganze UI aufgebaut
  - d.h. main-Thread = UI-Thread
  - Konsequenz: falls main-Thread blockiert, friert das UI ein („UI freeze“)
- Es gilt folgendes zu verhindern
  - Langandauernde Operationen (z.B. Netzwerk) auf main-Thread
  - weil diese lange dauern und synchron ablaufen (d.h. sie sind blockierend)
    - das bedeutet, dass keine UI Events mehr verarbeitet werden –> App-freeze

Android lässt gewisse Operation im Main-Thread gar nicht zu, z.B. Aufruf von `URLConnection.connect()` → führt zu `NetworkOnMainThreadException`

Demo: Mit `Thread.sleep(1000)` (oder andere Anzahl von Millisekunden) kann das Blockierungsproblem gezeigt werden.

5.2.1.1. Android-Überwachung: ANR (application not responding)

- Android System überwacht Ansprechbarkeit (responsiveness) von Apps
- Kriterien
  - keine Reaktion auf Input-Event innert 5 Sekunden
  - Broadcast-Receiver nicht fertig innert 10 - 20 Sekunden
- Effekt wenn Kriterien nicht eingehalten werden: ANR-Dialog
  - System-Mechanismus zum Stoppen von „bösen“ Apps

Lösung: Main Thread entlasten

Es ist sehr wichtig die richtige Lösung für eine Situation zu haben.

Die Falsche Handhabung kann dazu führen, dass z.B folgendes passiert:

- ANR
- Batterie wird schnell verbraucht
- usw.

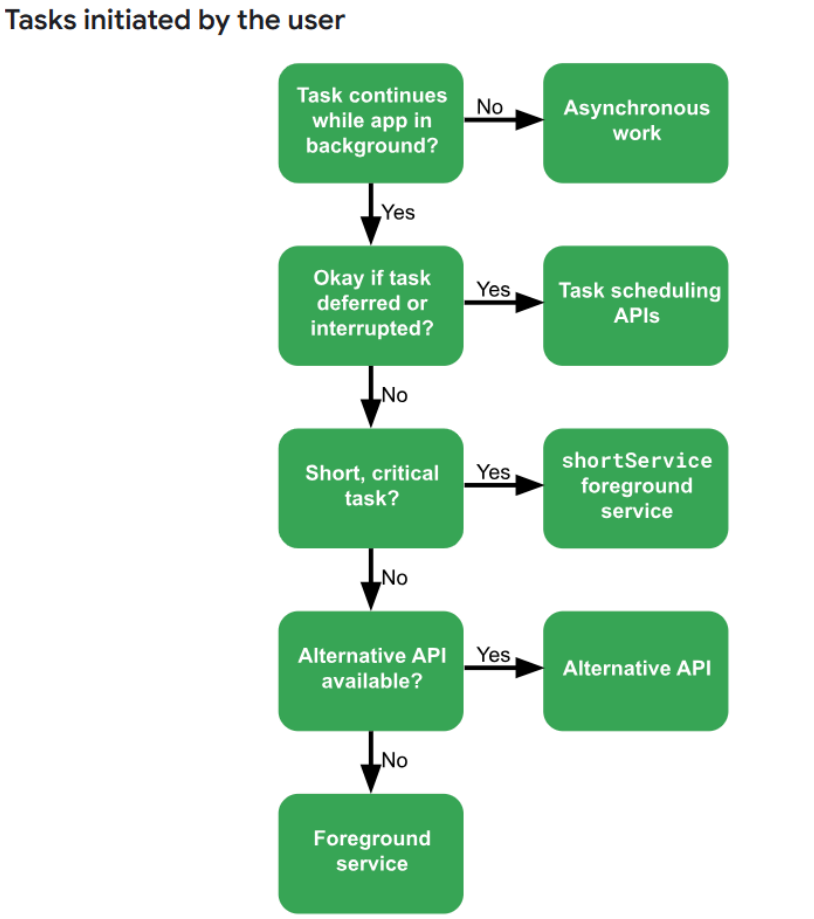

Figure 1: How to choose the right API for running a user-initiated background task.

Abbildung 8: Tasks initiated by the user

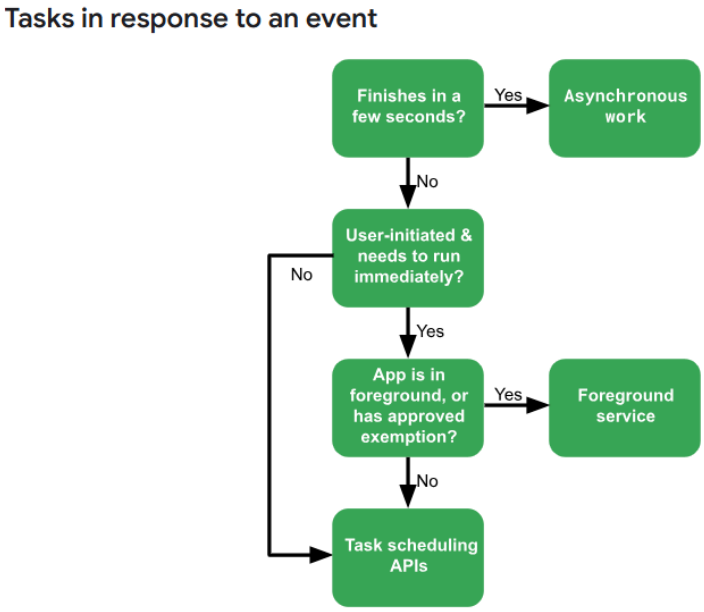

***Figure 2**: How to choose the right API for running an event-triggered background task.*

Abbildung 9: Tasks in response to an event

5.3. Kotlin Corountines

5.3.1. ViewModel

- Stellt der UI Daten zur Verfügung
- Beinhaltet Business-Logik
- Basisklasse `androidx.lifecycle.ViewModel`
- Überlebt Configuration Changes (Daten müssen nicht neu geladen werden, sonder sie werden gecached
- Keine Referenz auf UI (ViewModel muss das UI nicht kennen -> gibt sonst Memory Probleme)
- Läuft so lange wie Owner (Activity)

```kotlin
class BandViewModel: ViewModel() {
```

5.3.2. Basics

- Eine Coroutine ist ein suspendable (aussetzbare) Berechnung
- Läuft gleichzeitig (asynchron) mit dem restlichen Code (ähnlich wie ein Thread)
- Wechsel des auszuführende Thread Innerhalb einer Coroutine
- Neue Coroutines starten, die asynchron laufen

```kotlin
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
```

Hello wird zuerst ausgegeben

5.3.3. Scopes

Mit Coroutine Scopes werden Coroutines gestartet (launched).

- **Global Scope**: Lebt solange wie die Applikation
  - `GlobalScope.launch {}`

- **Lifecycle Scope**: Lebt solange wie der Lifecycle (Activity)
  - `lifecycleScope.launch {}`
- **ViewModel Scope**: Lebt solange wie das ViewModel
  - `viewModelScope.launch {}`

—

- Gibt einen Job zurück: Ge-cancelled sobald die Lebensdauer des entsprechenden Scopes endet
  - Job könnte auch manuell beendet werden (mit launch Funktion)
- für uns am relevantesten: `ViewModel` Scope

5.3.4. Context

- Der Coroutine Context beinhaltet den Job und den Dispatcher, der bestimmt in welchem Kontext die Coroutine ausgeführt werden soll.
  - **Dispatchers.Default**: Für CPU-intensive Arbeit weg vom Main-Thread (z.B. parsing JSON)
  - **Dispatchers.Main**: Coroutine läuft auf Main-Thread (z.B. für UI updates)
  - **Dispatchers.IO**: Für Netzwerk Calls und Interaktion mit dem Speicher (z.B. Room Access, Dateien schreiben/lesen)
- Dispatcher Wechsel mit `withContext(Dispatcher)`
  - Die Coroutine die `withContext` aufruft, wird solange suspended (wartend, aber nicht Thread-blockierend), bis der Code-Block fertig ist.

5.3.5. Suspendable Functions

- Keyword `suspend`
- Funktion nur innerhalb einer `Coroutine` aufrufbar
- z.B. wenn etwas vom Server aufgerufen werden soll

5.3.6. Async Kommunikation (`ViewModel` -> UI)

- Klasse `ViewModel` stellt UI asynchron Daten zur Verfügung (reactive programming)
- lädt die Daten um sie anzeigen zu können
- Daten werden in `Flow` geschrieben
- UI collected die Daten mit der Klasse `FlowCollector`

5.3.6.1. Flow

- ähnlich wie ein Iterator
- ein Stream aus Daten
- der Flow muss der gleiche Datentyp haben wie die Daten selber
- in den Datastream sind drei Entitäten involviert
  - Producer: produziert die Daten für den Stream
  - Intermediaries (optional): kann die Daten modifizieren
  - Consumer: konsumiert die Daten
- Beispiel: Producer ist ein Repository, Consumer ist das UI
- Es gibt unterschiedliche Typen von Flows
  - `SharedFlow` : shares Daten mit allen consumers, hot flow, der nicht beendet wird
  - `MutableSharedFlow` : Flow auf den Daten geschrieben werden können (Erbt von SharedFlow)
  - `StateFlow` : Wie SharedFlow, aber letzter Wert wird gespeichert. Kann über `.value` synchron gelesen werden.
  - `MutableStateFlow` : Flow auf den Daten geschrieben werden können (Erbt von StateFlow; Braucht Default Value)

→ Weitere Infos zum shared und state flow

5.4. Kommunikation über HTTP

- Apps kommunizieren im Hintergrund oft mit einem Server
  - Bsp. Server hält Daten oder Business-Logik die bereitgestellt wird oder eine user authentication
- Kommunikation mit dem Backend: (REST) HTTP-API
- Datenformat: JSON (seltener XML)

5.4.1. Request absetzen

- HTTP-Client-Library verwenden
  - Empfohlen: OkHttpClient (http://square.github.io/okhttp/)

Hinweis: Standard-Klassen `URL` und `URLConnection` erlauben das Absetzen von HTTP-Requests mit Java-Bordmitteln –> mühsam, uraltes API

5.4.2. Cleartext Traffic erlauben

Manchmal aber nicht möglich oder nicht erwünscht! Kann „clear text traffic“ in Manifest explizit erlauben.

```
<application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="DemoApplication"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.DemoApplication"
    android:usesCleartextTraffic="true"
```

Abbildung 10: Manifest

5.4.3. Manifest Berechtigung

- Für Internet-Zugriff (und Netzwerkstatus) muss Berechtigung vorliegen (resp. deklariert sein)
- → Nach Eintrag im Manifest muss ggf. App komplett neu installiert werden!
- Manifest:

```
<?xml version="1.0" encoding="utf-8" ?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
```

Abbildung 11: Manifest

5.5. JSON - Webservices mit Retrofit konsumieren

5.5.1. REST-ful Webservices

- Webservice auf der Basis von HTTP
- Grundidee (in purer Form)
  - Base-URL = Ressourcensammlung oder einer einzelnen Resource
  - HTTP-Methode = Operation auf Daten (GET, PUT, POST, DELETE)
  - Antwort-Datenformat = XML, JSON, …

Resource	GET	PUT	POST	DELETE
Collection URI, such as http://directory.com/contacts/	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URL is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as http://directory.com/contacts/17	**Retrieve** a representation of the addressed collection member, expressed in an appropriate media type.	**Replace** the addressed member of the collection, or if it doesn't exist, **create** it.	Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection.

5.5.2. JSON ⟷ Kotlin

Abbildung 13: JSON vs. Kotlin

5.5.3. Kotlinx Serialization

- JSON-to-Kotlin-Mapper
  - Bildet JSON-Strukturen auf äquivalente Kotlin Klassen ab
- Kotlinx Serialization mit Retrofit
  - konvertiert JSON-Daten in Kotlin-Objekte

5.5.4. Retrofit

- für HTTP Anfragen
- `@Serializable` den data classes hinzufügen
- Unterstütz Coroutine (`suspend`)
- als Interface wird die Function angeben
- `@Path` definieren welche Ressource man möchte

5.5.5. Beispiele

Data Class

```kotlin
@Serializable
data class BandCode(
    val name: String,
    val code: String

)
```

Interface

```kotlin
interface BandsApiService {
    @GET("all.json")  //von der Base URL
    suspend fun getBandNames(): Response<List<BandCode>>

    @GET("info/{code}.json")
    suspend fun getBandInfo(@Path("code") code: String): Response<BandInfo>  //@Path
definiert die Attribute
}
```

Konfiguration und Aufruf Beispiel

- `Retrofit.Builder()` erzeugt die Retrofit-Factory
- `baseURL` : Präfix für alle erzeugten Services

```kt
private val retrofit = Retrofit.Builder()
  .client(OkHttpClient().newBuilder().build())
  .addConverterFactory(Json.asConverterFactory("application/json".toMediaType()))
  .baseUrl("https://wherever.ch/hslu/rock-bands/")
  .build()
```

- Service-Instanz erzeugen:

```kt
private val bandsService = retrofit.create(BandsApiService::class.java)
```

- Service verwenden

```kt
val response = bandsService.getBandNames()
if (response.code() == HttpURLConnection.HTTP_OK) {
  response.body().orEmpty()
}
```

5.6. Coil

- Library
- lädt ein Bild asynchron

```kotlin
AsyncImage(
  modifier = Modifier.padding(8.dp),
  model = currentBand.bestOfCdCoverImageUrl,
  contentDescription = null
)
```

6. Zustand & Persistenz 📦

6.1. Projekt Strukturierung

- Compose functions sind klassenunabhängig
- Zu viele Funktionen in einer Datei sind unübersichtlich
- Klassen können Android-unabhängig sein
- Gute Struktur verbessert
  - Skalierbarkeit
  - Testbarkeit
  - Wiederverwendbarkeit

6.2. Strukturierung

- Vorschlag von Google
- **Schichtenarchitektur**
  - Trennung in UI Layer, Domain Layer (business logic) und Data Layer (Persistenz)
  - Für bessere Wartbarkeit
- **Reaktive Programmierung**
  - Asynchrone, eventbasierte Datenverarbeitung, z. B. mit Kotlin Flow
- **Separation of Concern**
  - Klare Trennung von Verantwortlichkeiten in Modulen oder Klassen
- **Single Source of Truth**
  - Eine zentrale Datenquelle zur Konsistenzsicherung

6.2.1. UI Ebene

- Beinhaltet
  - Compose functions
  - ViewModels
- Kommuniziert mit Domain Layer
  - Falls kein Domain Layer, direkt mit Data Layer

6.2.2. Domain Ebene

- Business Logik
- Abhängigkeit zur Data Ebene
- Keine Abhängigkeit zu Android (reiner Kotlin-Code)

6.2.3. Data Ebene

- Bereitstellen von Daten
  - Datasources
    - Datenbank (Room, SQLite) für persistente Speicherung
    - Datastore (Preferences, Proto) für Key-Value-Speicherung
    - Netzwerkquellen (APIs, Firebase) für externe Daten
  - Repositories
- Keine Abhängigkeit zu anderen Ebenen

6.2.4. Beispiel: Struktur

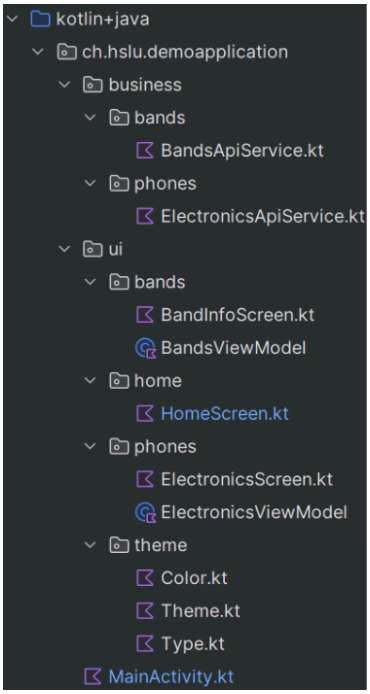

Abbildung 14: Strukturierung Projekt

6.3. Compose Zustand

- Screens haben Zustand, der sich durch Benutzerinteraktionen oder geladene Daten ändern kann.
- **Composable Functions meist stateless**
  - Sie nehmen Zustand als Parameter entgegen und geben UI zurück
  - Änderungen am Zustand müssen ausserhalb der Funktion verwaltet werden
- **Zustand manchmal notwendig**
  - Switch-Button: Ohne gespeicherten Zustand bleibt der Switch unverändert
  - Textfelder, Formulare, Listen mit dynamischem Inhalt
- **State hoisting (Zustandsauslagerung)**
  - Zustand sollte in einem übergeordneten Element gespeichert und in die Composable übergeben werden.

6.3.1. Remember

- `remember` speichert den Zustand über Recompositionen hinweg
- `mutableStateOf` ermöglicht das Aktualisieren des Zustands
- `by` -Delegation erspart das explizite `.value` :
  - **Ohne `by`** : `var name = remember { mutableStateOf("") }` → Zugriff mit `name.value`
  - **Mit `by`** : `var name by remember { mutableStateOf("") }` → Direkt nutzbar

6.3.1.1. Beispiel: Remember

```kotlin
var isActive by remember { mutableStateOf(false) }

Switch(
    checked = isActive,
    onCheckedChange = { isChecked ->
        isActive = isChecked
    }
)
```

- **`var isActive by remember { mutableStateOf(false) }`**
  - Speichert den Zustand (`true` / `false`) und merkt sich Änderungen.
- **`checked = isActive`**

- Bindet den Zustand an das UI.

- `onCheckedChange = { isActive = it }`
  - Aktualisiert den Zustand, wenn der Nutzer umschaltet.

6.3.2. Zustand von Flow

- **StateFlow**
  - `collectAsState()` beobachtet und aktualisiert automatisch
- **Flow/SharedFlow**
  - Benötigt `collectAsState(initial = ...)`, da kein gespeicherter Wert vorhanden ist

6.3.3. State hoisting und Single Source of Truth

- **Single Source of Truth**
  - Ein zentraler Zustand wird von mehreren `Composable`-Funktionen genutzt
- **State Hoisting**
  - Zustand wird in den hierarchisch nächsthöheren gemeinsamen Vorgänger verschoben
  - Vorteile: Bessere Wiederverwendbarkeit, einfachere Tests, klarere Trennung von UI und Logik
  - `Children Composables` bleiben stateless

6.3.4. Beispiel: State Hoisting & Single Source of Truth

KOTLIN

```kotlin
@Composable
fun MainScreen() {
    // Single Source of Truth: Der Zustand wird in der obersten Ebene verwaltet
    var name by remember { mutableStateOf("") }

    Column {
        // State Hoisting: Der Zustand wird an die untergeordnete Composable
weitergegeben
        NameInput(name) { name = it }

        // Anzeige des aktuellen Zustands
        Text("Eingegebener Name: $name")
    }
}

@Composable
fun NameInput(name: String, onNameChange: (String) -> Unit) {
    // NameInput ist stateless und erhält den Zustand von MainScreen
    TextField(
        value = name,               // Aktueller Wert des Textfelds
        onValueChange = onNameChange, // Callback zur Aktualisierung des Zustands
        label = { Text("Name") }    // Label für das Textfeld
    )
}
```

6.4. Persistenz

- Sollen über die Laufzeit der App erhalten bleiben
- **DataStore**
  - Für kleine Datenmengen
  - Key-Value mit Preferences
  - Objekte mit Proto
- **Dateisystem**
  - Für binäre Daten, grosse Dateien, Export
- **Datenbank (Room)**
  - SQLite mit Object Relational Mapper (ORM)
  - Für strukturierte Daten & Abfragen

6.4.1. DataStore

- Key-Value-Speicher (`persistente Map`)
- Nur eine Instanz pro Prozess, sonst `IllegalStateException`
- Asynchroner Zugriff:
  - Flows für das Lesen
  - Suspend-Funktionen für das Schreiben

6.4.1.1. Instanzierung

```kotlin
private val Context.dataStore by preferencesDataStore(name = "user_preferences")
```

- Erstellt DataStore-Instanz mit Namen `"user_preferences"`
- Extension Property für direkten Zugriff auf DataStore

6.4.1.2. Lesen

```kotlin
val userName: Flow<String> = context.dataStore.data
    .map { preferences -> preferences[stringPreferencesKey("user_name")] ?: "" }
```

- `context.dataStore.data.map {}` ⟶ Liest die gespeicherten Werte als Flow
- `stringPreferencesKey("user_name")` ⟶ Schlüssel für den gespeicherten Wert

6.4.1.3. Schreiben

```kotlin
val name = "Rolf"
context.dataStore.edit { preferences ->
    preferences[stringPreferencesKey("user_name")] = name
}
```

- `edit {}` ⟶ Führt eine Schreiboperation durch (nur in einer `suspend function` erlaubt)
- `preferences[stringPreferencesKey("user_name")] = name` ⟶ Speichert den Wert unter dem Schlüssel `"user_name"`

6.4.1.4. Auslagern von Keys

```kotlin
private object PreferencesKeys {
    val USER_NAME = stringPreferencesKey("user_name")
    val USER_AGE = intPreferencesKey("user_age")
    val USER_AUTHORIZED = booleanPreferencesKey("user_authorized")
}
```

- Verwaltung der Keys in einem eigenen `object`

6.4.1.5. Benötigte Komponenten

6.4.1.5.1. DataStore Setup

- Schlüssel für Preferences auslagern
  - Zentrale Verwaltung der Keys in einem `object`, um Fehler zu vermeiden
- DataStore als Extension Property für den Context
  - Ermöglicht einfachen Zugriff auf `DataStore` ohne Redundanz

6.4.1.5.2. Repository für Datenzugriff

- `Flow<User>` für reaktive Datenbereitstellung
  - Änderungen im `DataStore` werden automatisch in der UI reflektiert
- `suspend`-Funktionen für schreibende Operationen
  - Schreibvorgänge in `DataStore` erfolgen asynchron innerhalb von Coroutines

6.4.1.5.3. ViewModel

- `StateFlow<User>` zur Kapselung von `Flow<User>`
  - Konvertiert mit `stateIn()` einen `Flow` in `StateFlow` mit Default-Wert
- `updateUser` für Änderungen am Zustand
  - Speichert Änderungen asynchron durch `viewModelScope.launch`

6.4.1.5.4. ViewModelFactory

- Erstellt das `ViewModel` mit `Repository`-Abhängigkeit
  - Ermöglicht Dependency Injection und sorgt für korrekte Instanziierung
  - UI kennt nur das `ViewModel`, nicht das `Repository`

6.4.1.5.5. UI mit ViewModel

- `viewModel.collectAsState()` für reaktive Anzeige der Daten
  - Stellt sicher, dass die UI stets die aktuellsten Daten aus dem `ViewModel` erhält
- `UserInputs` als Composable mit `onUpdateUser`
  - Ermöglicht die Weitergabe von Benutzeränderungen an das `ViewModel`
  - Nutzt `remember(user.name) { mutableStateOf(user.name) }`, um UI-Zustand abzuleiten

6.5. App spezifischer Speicher

- **Interner Speicher**
  - Persistente oder Cache Daten
  - Sicher, verschlüsselt (ab Android 10)
  - Kein Zugriff durch andere Apps
  - Löscht alle Daten bei Deinstallation
  - Begrenzter Speicherplatz
- **Externer Speicher (SD-Karte)**
  - Persistente oder Cache Daten
  - Zugriff durch andere Apps möglich
  - Daten bleiben nach Deinstallation erhalten
  - Grösserer Speicherplatz verfügbar

6.6. Datenbank (Room)

- **SQLite**
  - Relationales DBMS für mobile Geräte
  - Open Source, eine Datei pro Datenbank
  - Low-Level, erfordert manuellen Abstraktionslayer
- **Room (ORM für SQLite in Android)**
  - Abstraktionslayer über SQLite, erleichtert Datenbankzugriff
  - Mapped Klassen (OO-Entities) auf Tabellen
  - DAO-Pattern (Data Access Object) für Datenzugriff
- **Besonderheiten von Room**
  - SQL-Queries als Annotationen in DAOs
  - Manuelle Modellierung von Beziehungen (Performance-Optimierung)
  - Nested Objects: Mehrere POJOs (Plain Old Java Object) in einer Tabelle
  - Einschränkungen für Datenzugriff (nicht im UI-Thread)

6.7. Komponenten

- **Database**
  - Abstrahiert die Verbindung zur SQLite-Datenbank
  - Verwaltet den Zugriff auf die DAOs
- **Entity**
  - Repräsentiert eine Tabelle in der Datenbank
  - Definiert die Struktur der gespeicherten Daten
- **DAO (Data Access Object)**
  - Enthält Methoden für den Datenzugriff
  - Definiert SQL-Operationen wie `Insert`, `Update`, `Delete`, `Query`

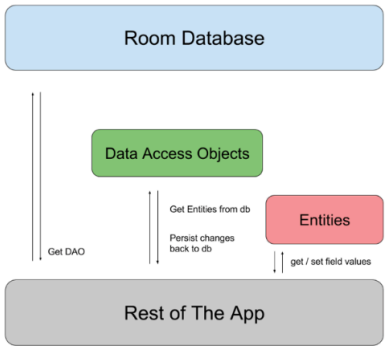

Abbildung 15: Room Komponenten

6.8. Database

6.8.1. Deklaration der Datenbank

- `@Database` Annotation
  - Definiert die Entities (Tabellen) und die Version für Migrationen
- `abstract class DemoDatabase : RoomDatabase()`
  - Muss von `RoomDatabase` erben
  - Enthält eine abstrakte Methode für den Zugriff auf `Dao`

6.8.2. Singleton-Instanz

- `companion object` verwaltet eine statische Instanz
  - `INSTANCE` stellt sicher, dass nur eine Instanz der Datenbank existiert
- `getDatabase(context)`
  - Erstellt die Datenbank beim ersten Zugriff und speichert sie in `INSTANCE`

6.8.3. Erstellung und Konfiguration

- `buildDatabase(context)` nutzt `Room.databaseBuilder`
  - Erstellt eine neue Instanz der Datenbank mit dem angegebenen `DB_NAME`
- Setzt `QueryExecutor` und `TransactionExecutor`
  - Alle Operationen laufen auf dem `IO`-Thread, um die UI nicht zu blockieren

6.8.4. Beispiel

```kotlin
@Database(entities = [UserEntity::class], version = 1)
abstract class DemoDatabase : RoomDatabase() {

    // DAO für den Zugriff auf die Datenbank
    abstract fun userDao(): UserDao

    companion object {
        private const val DB_NAME = "demo-database"
        @Volatile
        private var INSTANCE: DemoDatabase? = null

        // Singleton-Instanz der Datenbank abrufen oder erstellen
        fun getDatabase(context: Context): DemoDatabase {
            return INSTANCE ?: buildDatabase(context).also { INSTANCE = it }
        }

        // Erstellt die Room-Datenbank
        private fun buildDatabase(context: Context): DemoDatabase {
            val ioDispatcherExecutor = Dispatchers.IO.asExecutor()
            return Room.databaseBuilder(
                context,
                DemoDatabase::class.java,
                DB_NAME
            )
                .setQueryExecutor(ioDispatcherExecutor)
                .setTransactionExecutor(ioDispatcherExecutor)
                .build()
        }
    }
}
```

6.9. Entity

- Beschreibt eine Datenbanktabelle in Room.
- Jede Entity ist eine Datenklasse mit einer `@Entity`-Annotation.

6.9.1. Grundstruktur einer Entity

- `@Entity`: Definiert eine Klasse als Datenbanktabelle.
- `@PrimaryKey`: Markiert ein Feld als Primärschlüssel (einzeln oder kombiniert).
- `autoGenerate = true`: Lässt den Primärschlüssel automatisch generieren.
- `@Ignore`: Markiert Felder, die nicht in die Datenbank gespeichert werden.
- `ignoredColumns`: Definiert mehrere ignorierte Felder (z. B. aus Superklassen).

6.9.2. Beispiel

```kotlin
@Entity
class User : Party {
    @PrimaryKey(autoGenerate = true)
    var id: Int =UI mit ViewModel 0
    var firstName: String? = null
    var lastName: String? = null
```

```
    @Ignore
    var picture: Bitmap? = null  // Wird nicht in der Datenbank gespeichert
}
```

6.10. DAO

- Kapselt Datenbankzugriff, getrennt von der Geschäftslogik (Separation of Concerns).
- Interface/abstrakte Klasse ⟶ Room generiert Implementierung.

6.10.1. Convenience Queries

- `@Insert`, `@Update`, `@Delete` definieren Standard-DB-Operationen
- Rückgabewerte:
  - `Insert` ⟶ ID(s) der eingefügten Zeilen.
  - `Update` / `Delete` ⟶ Anzahl der betroffenen Zeilen.

6.10.2. Custom Queries mit `@Query`

- `@Query` für SQL-Abfragen, Kompilierzeit geprüft
- Beliebige Parameter, POJOs mit `@ColumnInfo` sparen Speicher
- Suspend-Funktionen für asynchronen Zugriff mit Coroutines

6.10.3. Beispiel

KOTLIN

```kotlin
@Dao
interface UserDao {

    // Einfügen eines Users
    @Insert
    fun insertUser(user: User): Long

    // Aktualisieren eines Users
    @Update
    fun updateUser(user: User): Int

    // Löschen eines Users
    @Delete
    fun deleteUser(user: User): Int

    // Custom Query: Alle User über 18 abrufen
    @Query("SELECT * FROM user WHERE age > 18")
    fun getAdultUsers(): List<User>
}
```

6.11. Beziehungen modellieren

- Keine direkten Objekt-Referenzen
  - Lösung: Foreign Keys oder Nested Objects
- `@Embedded`: Speichert Felder direkt in der gleichen Tabelle
  - Sinnvoll, wenn das eingebettete Objekt keine eigenständige Existenz benötigt
  - `@Embedded val address: Address`
- Alternative: IDs verknüpfen, per Joins abfragen

6.11.1. 1-1 (One-to-One)

- Foreign Key + `@Relation` lädt verknüpfte Entität
- Beispiel: Ein User hat genau eine Library

```kotlin
@Entity
data class User(@PrimaryKey val userId: Long)

@Entity
data class Library(@PrimaryKey val libraryId: Long, val userOwnerId: Long)

data class UserWithLibrary(
    @Embedded val user: User,
    @Relation(parentColumn = "userId", entityColumn = "userOwnerId")
    val library: Library
)
```

6.11.2. 1-n (One-to-Many)

- `@Relation` für eine Liste verknüpfter Entitäten
- Beispiel: Ein User hat mehrere Playlists

```kotlin
@Entity
data class User(@PrimaryKey val userId: Long)
@Entity
data class Playlist(@PrimaryKey val playlistId: Long, val userCreatorId: Long)

data class UserWithPlaylists(
    @Embedded val user: User,
    @Relation(parentColumn = "userId", entityColumn = "userCreatorId")
    val playlists: List<Playlist>
)
```

6.11.3. n-n (Many-to-Many)

- Zwischentabelle mit `@Entity(primaryKeys = [...])`
- `@Junction` für beidseitige Abfragen (z. B. Songs ↔ Playlists)

```kotlin
@Entity
data class Playlist(@PrimaryKey val playlistId: Long, val name: String)

@Entity
data class Song(@PrimaryKey val songId: Long, val title: String)

@Entity(primaryKeys = ["songId", "playlistId"])
data class SongPlaylistCrossRef(val songId: Long, val playlistId: Long)

data class PlaylistWithSongs(
    @Embedded val playlist: Playlist,
    @Relation(
        parentColumn = "playlistId", entityColumn = "songId",
        associateBy = Junction(SongPlaylistCrossRef::class)
    )
    val songs: List<Song>
)
```

6.12. LazyColumn

- **Column**:
  - Gut für kleine Listen
  - Bei langen Listen Performance-Probleme
- **LazyColumn**:
  - Lädt nur sichtbare Elemente

- Automatisch scrollbar
- Bessere Performance

6.12.1. Beispiel - LazyColumn

```kotlin
val stringList = listOf("Hallo", "Android", "Wie gehts")

LazyColumn {
    items(
        count = stringList.size
    ) { index ->
        Text(stringList[index])
    }
}
```

- `items()` iteriert über Elemente mit Index
- `count` : Anzahl der Elemente
- `itemContent` : Inhalt jedes Eintrags

6.13. Best Practices

- DAO-Methoden immer in Hintergrund-Threads ausführen
- Singleton-Pattern für Room-Datenbank verwenden
- Datenbank-Updates mit Migrationsstrategien absichern
- Flows oder LiveData nutzen, um UI mit DB-Änderungen zu synchronisieren

7. Permissions, Ressources & Services 🔓

7.1. Permissions

Bestimmte App-Funktionen benötigen Benutzer-Permission:

- Kontakte
- Internet
- SD-Karte
- Kamera
- SMS
- Telefonieren

Permissions müssen vom Benutzer genehmigt werden

Deklaration im App-Manifest erforderlich

- App gibt an, welche Berechtigungen benötigt werden

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="ch.hslu.mobpro.persistency">

<uses-feature
  android:name="android.hardware.telephony"
  android:required="false" />
<uses-permission android:name="android.permission.READ_SMS" />
```

Abbildung 16: Permissions im Manifest

7.1.1. Permissons seit API 23

Seit API 23 (Android 6.0): **Keine automatischen ‚dangerous' Permissions** mehr bei Installation

- Nur unkritische (normal) Permissions werden direkt gewährt

Kritische Permissons:

- Müssen **zur Laufzeit angefragt** werden (beim ersten Bedarf)

Benutzer kann Permissions **einzeln ablehnen** oder **nachträglich entziehen**

→ Apps müssen auch ohne alle gewährten Berechtigungen korrekt funktionieren

7.1.1.1. vor API 23

Rechte mussten bei der Installation vom Benutzer gewährt werden (alles oder nichts).

7.1.2. Arten von Permissons

normal

- automatisch bei Installation erlaubt

dangerous

- durch User gesteuert (erlauben / ablehnen)
- Runtime Permissions

signature

- Nur erlaubt, wenn beide Apps vom gleichen Hersteller stammen

7.1.3. Permissions Flowchart

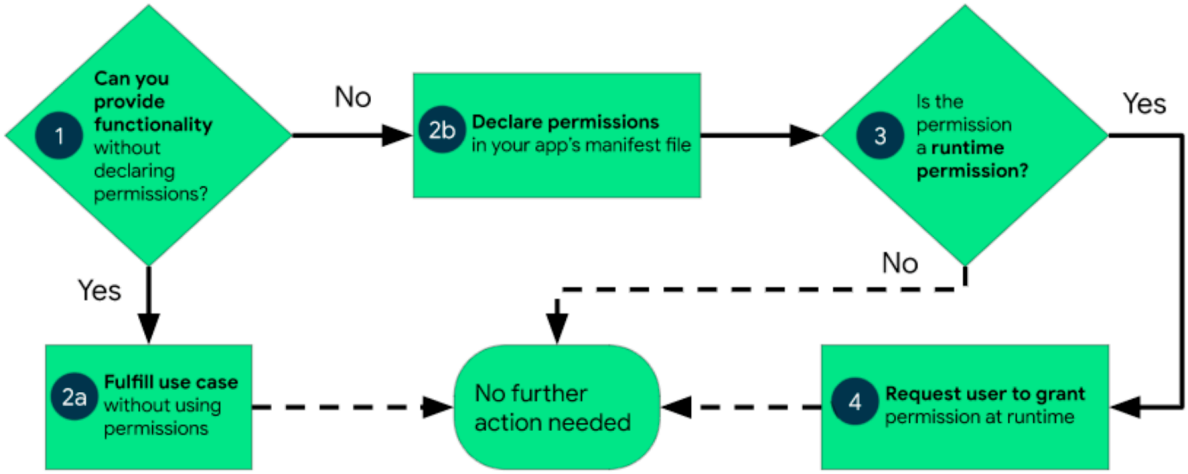

Abbildung 17: Permissions Flowchart

7.1.4. Runtime Permissions

Der Benutzer muss sie **zur Laufzeit explizit erlauben** – wenn die App die Funktion **zum ersten Mal nutzen** will (**dangerous**)

```kt
// Compose Runtime Permission Check und Anfrage ist noch experimental
@OptIn(ExperimentalPermissionsApi::class)
@Composable
fun ServiceWidget() {
    // POST_NOTIFICATIONS ist erst seit Android Tiramisu (33) "dangerous"
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        val notificationPermissionState =
            rememberPermissionState(
                android.Manifest.permission.POST_NOTIFICATIONS
            )

        // Runtime-Check: benötigte Permission(s)? sonst Permission(s) anfragen
        if (!notificationPermissionState.status.isGranted) {
            Button(
              // launchPermissionRequest nicht direkt im Composable State ausführen
                onClick = {
                    notificationPermissionState.launchPermissionRequest()
                }
            ) {
                Text("Request permission")
            }
        }
    }
}
```

7.2. Ressourcen

- Nicht-Java/Kotlin-Bestandteile einer App befinden sich im **`res/`-Verzeichnis**
- Enthält ausgelagerte **Konstanten** (z. B. Strings, Styles, Bilder, Menüs)
- Zugriff im Code über automatisch generierte **`R`-Klasse** mit `int`-IDs
- Ressourcen können kontextabhängig sein:
  - Sprache
  - Bildschirmauflösung
  - Gerätetyp
  - Hardwaremodell

7.2.1. Beispiele für Ressourcen

- Texte ⟶ `strings.xml`
- Bilder ⟶ `drawables`
- Farben, Dimensionen, Styles ⟶ `values/`
- Arrays, Menüs ⟶ `arrays.xml`, `menu/`

7.2.2. Spezifische Ressourcen

- Unterschiedliche Varianten für verschiedene Systemkonfigurationen
- Beispiele:
  - Internationalisierung (`values-de/strings.xml`)
  - Gerätespezifische Ressourcen
  - Bildvarianten je nach Auflösung (oder Vektorformate)
- Hinweis: Konfigurationen können zur Laufzeit wechseln (z. B. Sprache!)

7.2.3. Ressourcen-Organisation

7.2.3.1. Standardverzeichnisse (`res/`)

- `drawable/`, `layout/`, `menu/`, `values/`

7.2.3.2. Spezifische Varianten

- Gleiche Struktur, aber mit Konfigurations-Suffix (z. B. `values-de/`)
- Override-Prinzip: Android sucht Ressourcen von spezifisch nach allgemein

7.2.4. Ressourcen in Jetpack Compose

7.2.4.1. Strings

```kotlin
val appName: String = stringResource(R.string.app_name)
```

7.2.4.2. Bilder

```kotlin
val icon: Painter = painterResource(R.drawable.ic_launcher_foreground)
Icon(
    painter = icon,
    contentDescription = null // decorative element
)
```

7.3. Notifications

- Signalisiert dem Benutzer:
  - Etwas passiert gerade
  - Etwas ist passiert
- Kann von einer App oder dem System stammen
- Voraussetzung für einen Foreground Service
- Benötigt User-Permission: `POST_NOTIFICATIONS`

7.3.1. Notification Channels

- Gruppieren Benachrichtigungen in Kategorien
- User kann Notifications pro Channel erlauben oder blockieren
- Pflicht ab Android API Level 26 (Android 8 / Oreo)

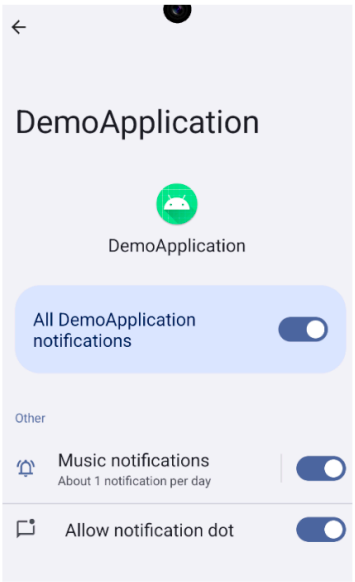

Abbildung 18: Notification Channels

7.3.2. Erstellen und Zuweisen

7.3.2.1. Channel erstellen

```kotlin
val channel = NotificationChannel(
    "ch.hslu.mobpro.demo.channel",         // Channel ID
    "Music notifications",                 // Channel Name (sieht man in den Settings)
    NotificationManager.IMPORTANCE_DEFAULT // Wichtigkeit
)

val notificationManager =
    getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
notificationManager.createNotificationChannel(channel)
```

7.3.2.2. Notification erstellen

```kt
val notification = NotificationCompat.Builder(this, "ch.hslu.mobpro.demo.channel")
    .setContentTitle("HSLU Music Player")
    .setContentText("musicTitle")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    .setOngoing(true)
    .setSmallIcon(android.R.drawable.ic_media_play)
    .setLargeIcon(
      BitmapFactory.decodeResource(resources, android.R.drawable.ic_media_play)
```

```
    )
    .setWhen(System.currentTimeMillis())
    .setCategory(Notification.CATEGORY_SERVICE)
    .build()
```

7.3.2.3. Notification anzeigen

```kt
notificationManager.notify(
    23, // Notification ID
    notification
)
```

7.4. Service

- App-Komponente zur Kapselung von Hintergrundarbeit
- Ursprünglich für Background-Tasks gedacht
- Heute empfohlen für:
  - Foreground Services
  - Export von App-Logik
- Klassische Anwendungsfälle:
  - Musikplayer
  - Navigation (z. B. EasyRide, SBB Mobile)
  - Downloads (z. B. Swisstopo)
- Für andere Fälle: Verwendung von `WorkManager` empfohlen

7.4.1. Konzept – Was bietet ein Service?

- Möglichkeit, dem System mitzuteilen, dass eine Hintergrundarbeit erledigt werden soll
- Möglichkeit, Funktionalität (API) für andere Apps bereitzustellen

7.4.2. Was ist ein Service nicht?

- Kein eigener Thread (läuft standardmässig im Main-Thread)
- Kein eigener Prozess (nur wenn explizit so definiert)

7.4.3. Start und Verbindung

- `startService()` – startet den Service (einmaliger Auftrag)
- `bindService()` – baut Verbindung zum Service auf (für Kommunikation)

7.4.4. Lang andauernde Operationen

- Ein Service sollte einen eigenen Thread starten, um lang andauernde Aufgaben zu erledigen
- z. B. mit `HandlerThread`, `Executor`, `Coroutine`

7.4.5. Lebensarten von Services

7.4.5.1. Foreground

- Es muss eine Notification angezeigt werden
- Läuft weiter, auch wenn die App nicht im Vordergrund ist
- Muss explizit beendet werden
- Beispiel: Musikplayer

7.4.5.2. Background

- Nicht sichtbar für den User
- Wird limitiert, wenn die App nicht im Vordergrund ist
- nicht mehr oft im Einsatz

7.4.5.3. Bound

- Wird mit `bindService()` gebunden

- Bietet eine Client-Server-Schnittstelle
- Existiert nur, solange mindestens ein Client verbunden ist

7.4.6. Lifecycle-Callbacks

7.4.6.1. Bei `startForegroundService(...)`

- `onCreate()` – bei Erstellung
- `onStartCommand()` – wenn ein Auftrag empfangen wird
- `onDestroy()` – bei Beendigung durch Service, App oder System

7.4.6.2. Bei `bindService(...)`

- `onCreate()` – bei Erstellung
- `onBind()` – wenn Verbindung hergestellt wird
- `onUnbind()` – wenn Verbindung beendet wird
- `onDestroy()` – bei Beendigung

7.4.7. Service Lifecycle

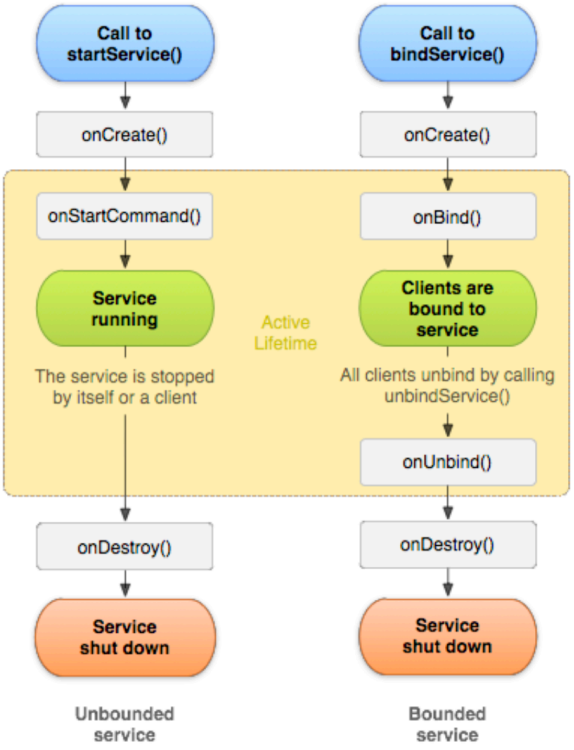

Abbildung 19: Notification Channels

7.4.8. Foreground Service

- Für Hintergrund-Arbeiten, die für Benutzer **wahrnehmbar** sind z. B. Musikplayer
- Zeigt während der Laufzeit eine Notification an
- Benötigt Permission `FOREGROUND_SERVICE` sowie einen Service-Typ, z. B. `FOREGROUND_SERVICE_MEDIA_PLAYBACK`
  - **Normal Permission** → wird automatisch vergeben
- Zusätzlich notwendig: `POST_NOTIFICATIONS`
  - **Dangerous Permission** → muss zur Laufzeit angefragt werden
- Interaktion / Steuerung häufig über Binding (`bindService()`)
- Service wird bei App-Start oder durch ein Event gestartet (`startForegroundService()`)
- Läuft dauerhaft im Vordergrund (sichtbar durch Notification)
- Startet eigenen Worker-Thread oder Thread-Pool (ggf. idle bis zur Nutzung)
- Kommunikation über `bindService()` → synchron möglich

7.4.8.1. Beispiel

Klasse muss von `Service` erben:

```kt
class MusicPlayerService : Service()
```

Implementierung `onStartCommand`

```kt
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    startService() // Wird bei startService() aufgerufen. Achtung: main Thread!
    return super.onStartCommand(intent, flags, startId)
}
```

```kt
private fun startService() {
    ServiceCompat.startForeground(
        service = this,
        NOTIFICATION_ID,
        createNotification(musicTitle = "-- waiting --"),
        // Macht aus diesem Service einen Foreground-Service
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
            // Versions-Check um anzugeben welcher Service Typ
            ServiceInfo.FOREGROUND_SERVICE_TYPE_MEDIA_PLAYBACK
        } else {
            0
        }
    )
}
```

Implementierung `onDestroy()`

```kt
override fun onDestroy() {
    stopForeground(STOP_FOREGROUND_REMOVE) // Wird bei stopService() aufgerufen
    // Macht aus dem Service wieder einen Background-Service (Notification entfernt)
    super.onDestroy()
}
```

7.4.9. Service im Manifest

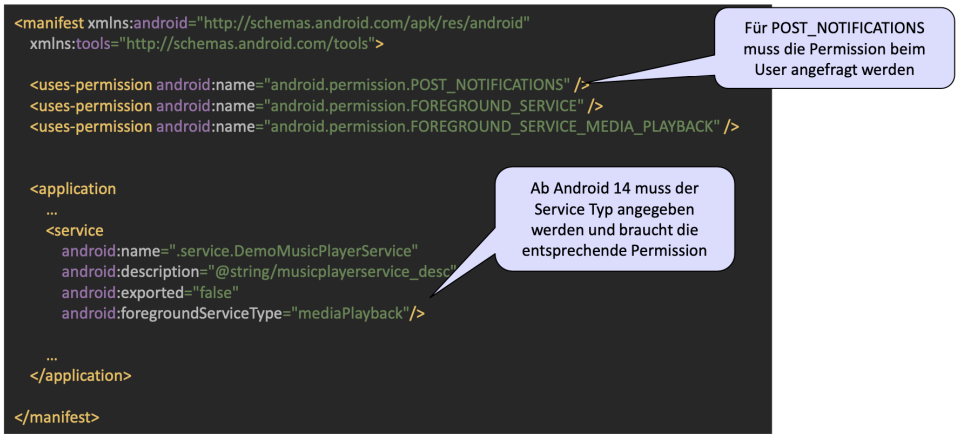

Abbildung 20: Service im Manifest

7.4.10. Service starten

```kt
val context = LocalContext.current
Button(onClick = {
    val intent = Intent(context, MusicPlayerService::class.java)
    context.startForegroundService(intent)
}) {
    Text(text = "Start Service")
}
```

7.5. Service stoppen

```kt
Button(onClick = {
    context.stopService(intent)
    val intent = Intent(context, MusicPlayerService::class.java)
}) {
    Text(text = "Stop Service")
}
```

7.5.1. Intents

Kommunikation zwischen Android-Komponenten Wichtige Bestandteile:

- `action` : Was soll passieren?
- `category` : Zusatzinformationen zur Action
- `type` : MIME-Type
- `component` : Ziel-Komponente
- `extras` : Zusatzdaten als Bundle

7.5.1.1. Intent-Typen

Expliziter Intent:

- Ziel-Komponente direkt angegeben
- Beispiel: `Intent(context, MusicPlayerService::class.java)`

Impliziter Intent:

- Keine Ziel-Komponente angegeben
- System entscheidet anhand von `action` , `category` , `type`
- Verwendung mit Intent-Filter (z. B. für Broadcasts)

7.5.2. Gebundene Services

- Verbindung über `bindService(intent, connection, flag)`
- Kommunikation über `ServiceConnection`
- Exportiert App-Funktionalität z. B. für Remote Services
- Verbindung kann mit `unbindService(connection)` gelöst werden

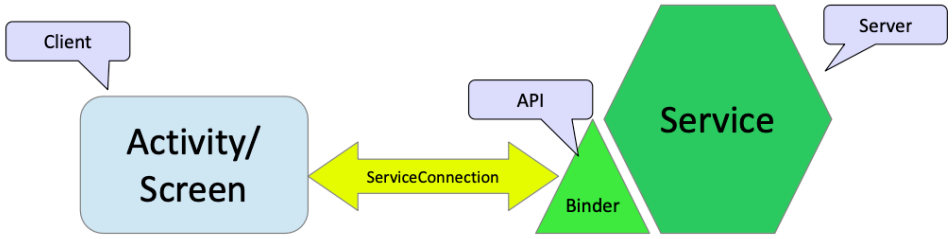

Abbildung 21: Gebundene Services

7.5.2.1. Beteiligte Klassen Beispiel

- `MusicPlayerApi` : **Service-Interface** mit API-Definition
- `MusicPlayerApiImpl` : **Binder-Klasse**, implementiert Interface + `Binder`
- `MusicPlayerService` : **Service**, gibt `Binder` in `onBind()` zurück
- `MusicPlayerConnection` : **Implementiert `ServiceConnection`, erhält API**
- **Client-Komponente** (z. B. Activity) ruft `bindService(...)` auf

7.5.2.2. Ablauf: Bindung & API-Aufruf

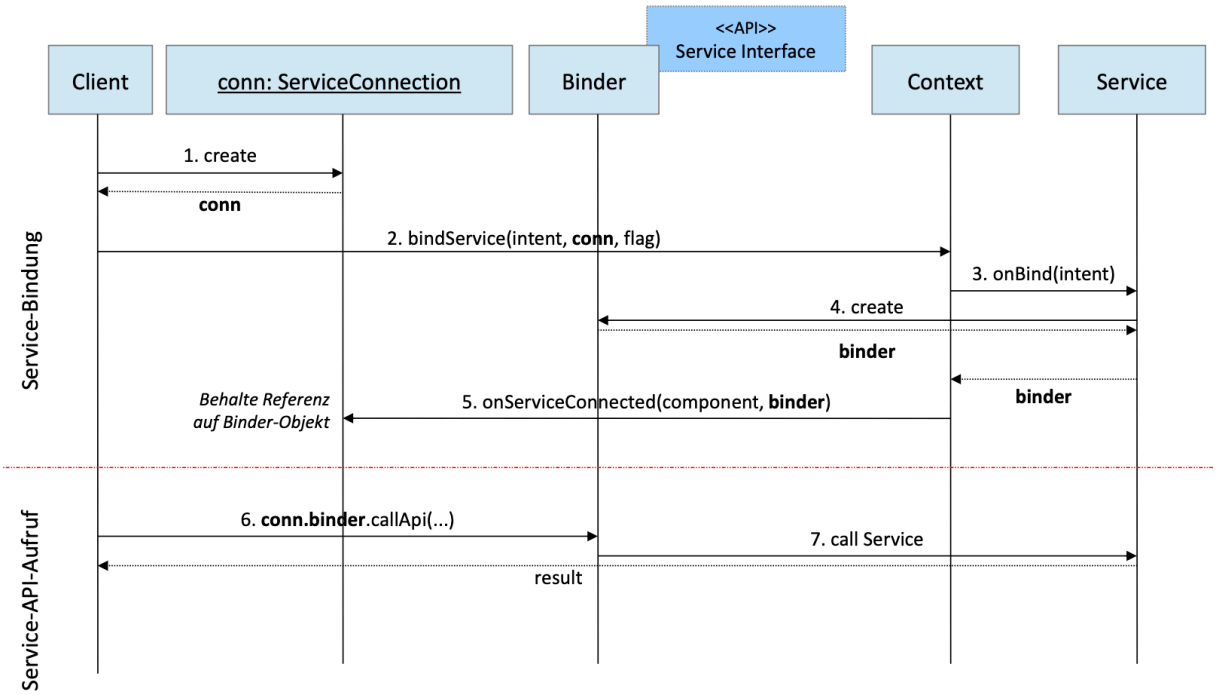

Abbildung 22: Service im Manifest

1. **Erstellung der ServiceConnection** Der Client erstellt ein Objekt vom Typ `ServiceConnection` (hier `conn`), um spätere Callbacks wie `onServiceConnected()` empfangen zu können.
2. **Servicebindung aufbauen** Der Client ruft `bindService(intent, conn, flag)` auf, um die Verbindung zum Service herzustellen.
3. **Service verarbeitet Bind-Anfrage** Die Methode `onBind(intent)` wird im Service aufgerufen. Dabei wird ein Binder-Objekt zurückgegeben.

4. **Binder-Objekt wird erstellt** Das Binder-Objekt wird erzeugt (falls nicht vorhanden) und über den Systemkontext an den Client zurückgegeben.
5. **Callback: onServiceConnected** Der Client erhält im Callback `onServiceConnected(component, binder)` das Binder-Objekt und speichert die Referenz darauf, um später API-Aufrufe tätigen zu können.
6. **API-Aufruf durch den Client** Der Client ruft eine Methode über das Binder-Objekt auf, z. B. `conn.binder.callApi(...)`.
7. **Service verarbeitet den Aufruf** Der Service verarbeitet den API-Aufruf und liefert ein Ergebnis zurück.

7.5.2.3. Beispiel Implementation

7.5.2.3.1. Client

```kt
// Service API Definition
interface MusicPlayerApi {
    fun playNextSong(): String
}

// Gibt Binder-Instanz zurück (= API Instanz)
private val musicPlayerAPI = MusicPlayerApiImpl()
override fun onBind(intent: Intent): IBinder {
    return musicPlayerAPI
}

// Implementiert API (um Funktionen aufrufen zu können)
// Implementiert Binder, damit die Klasse in der onBind function zurückgegeben werden
kann
inner class MusicPlayerApiImpl : MusicPlayerApi, Binder() {

    // Funktion muss im Service implementiert werden
    override fun playNextSong(): String {
        return this@MusicPlayerService.playNextSong()
    }
}
```

7.5.2.3.2. Server

```kt
val intent = Intent(context, MusicPlayerService::class.java)
// ServiceConnection Instanz erstellen und damit bindService() aufrufen
serviceConnection = MusicPlayerConnection().also { connection ->
    context.bindService(intent, connection, Context.BIND_AUTO_CREATE)
}

// Service "unbinden" mit unbindService() und der ServiceConnection gecastet werden
context.unbindService(serviceConnection as ServiceConnection)

// ServiceConnection implementieren
class MusicPlayerConnection : ServiceConnection {

    private var musicPlayerApi: MusicPlayerApi? = null

    // MusicPlayerApi für das UI hier beziehen
    fun getMusicPlayerApi(): MusicPlayerApi? {
        return musicPlayerApi
    }

    override fun onServiceDisconnected(name: ComponentName) {
```

```
        musicPlayerApi = null
    }

    override fun onServiceConnected(name: ComponentName, service: IBinder) {
        // Weil MusicPlayerApiImpl "Binder" implementiert, kann er in onBind im Service
zurückgegeben werden.
        // Weil MusicPlayerApiImpl "MusicPlayerApi" implementiert, kann service als
MusicPlayerApi gelesen werden.
        musicPlayerApi = service as MusicPlayerApi
    }
}
```

8. Broadcast Receiver, Content Provider & Testing

8.1. Broadcast Receiver

- **Broadcasts = Nachrichten**
- App-interner «Message Bus»
- Alle Komponenten (Activity, Service, Content Provider) können Broadcasts senden und empfangen
- System sendet Broadcasts bei bestimmten Events (z. B. App installiert, Timer)
- Seit API 26 (Android 8, Oreo) stark eingeschränkt:
  - Wegen Performanceproblemen (zuviele Handler, Messages)
  - Nur noch wenige Events werden global verteilt
- **Beispiel**: «OnBootCompleted»

8.1.1. Komponente

- Broadcast Receiver nur während Nachrichtenverarbeitung aktiv
  - Danach inaktiv, wird vom System entfernt
  - Bei Bedarf erzeugt („on demand“)
- **Kein UI**
  - Kommunikation via Notification
  - **Für Hintergrund**: Service starten

8.1.1.1. Do

- Activity oder Service starten
- Notification schicken

8.1.1.2. Don’t

- Keine asynchronen Aufgaben
- Kein Service binden
- Kein Dialog anzeigen

8.1.2. Verschicken

- Broadcasts werden als Intents verschickt ⟶ können Daten mit sich tragen
- **Senden** eines Broadcast:
  - `context.sendBroadcast(intent)`
- **Empfang** von Broadcasts über Receiver (Empfänger)
  - **Dynamische Registrierung** im Code:
    - `ContextCompat.registerReceiver(requireContext(), broadcastReceiver, filter, RECEIVER_NOT_EXPORTED)`
    - Empfohlen wird nur dann geladen
  - **Statische Definition** im Manifest:
    - `<receiver ...>` ⟶ wird auch aufgerufen, wenn die App nicht gestartet ist

8.1.3. Typen

- **Expliziter Broadcast**
  - `package` im Intent definiert (App-Package)
  - Komponente direkt angegeben (wie bei Service)
  - Auflösung über `action` des Intents (Intent Receiver)
- **Impliziter Broadcast**
  - Kein Empfänger direkt definiert
  - Auflösung über `action` des Intents (Intent Receiver)
  - Kann **nicht** von Manifest-Receivern empfangen werden (nur dynamisch registrierte)

8.1.4. Broadcast Receiver im Manifest

- Im Manifest mit `<receiver>`-Tag definieren
  - `android:exported` bestimmt, ob externe Broadcasts empfangen werden dürfen
  - `<intent-filter>` mit passender `action`

```xml
<receiver android:name=".components.MyManifestBroadcastReceiver"
          android:exported="false">
    <intent-filter>
        <action android:name="ch.hslu.demoapplication.MY_ACTION" />
    </intent-filter>
</receiver>
```

- Eigene Klasse leitet von `BroadcastReceiver` ab
  - `onReceive()` überschreiben
  - `intent?.action` prüfen

```kotlin
class MyManifestBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        if (intent?.action == "ch.hslu.demoapplication.MY_ACTION") {
            Log.d("MyManifestBroadcastReceiver", "Broadcast received")
        }
    }
}
```

- Expliziten Broadcast an App senden
  - `Intent("ACTION")` erzeugen
  - `intent.package` setzen für expliziten Broadcast
  - `sendBroadcast(intent)` aufrufen

```kotlin
val intent = Intent("ch.hslu.demoapplication.MY_ACTION")
intent.`package` = "ch.hslu.demoapplication"
context.sendBroadcast(intent)
```

8.1.5. Lokale Broadcasts: «App Message-Bus»

- Verwendung von `LifecycleStartEffect` zur automatischen Registrierung und Deregistrierung
- Registrierung des BroadcastReceivers mit:
  - `IntentFilter(...)` : definiert, auf welche Action reagiert wird
  - `RECEIVER_NOT_EXPORTED` : schützt vor externen Broadcasts
- Cleanup über `onStopOrDispose` : verhindert Memory Leaks
- Empfohlen für app-interne Kommunikation (z. B. zwischen Composables oder Komponenten)

```kotlin
val context = LocalContext.current

// Reagiere auf Lifecycle-Änderungen (z. B. Composable wird sichtbar)
LifecycleStartEffect(Unit) {
    // BroadcastReceiver-Instanz erstellen
    val receiver = MyLocalBroadcastReceiver()

    // Registrierung des Receivers für eine bestimmte Action
    ContextCompat.registerReceiver(
        context,
        receiver,
        IntentFilter(BroadcastActions.LOCAL_ACTION), // Action-Filter
        RECEIVER_NOT_EXPORTED // nur interne Broadcasts erlaubt
    )

    // Beim Verlassen (z. B. Composable wird entfernt) Receiver wieder abmelden
    onStopOrDispose {
        context.unregisterReceiver(receiver)
```

```
    }
}
```

8.2. Content Provider

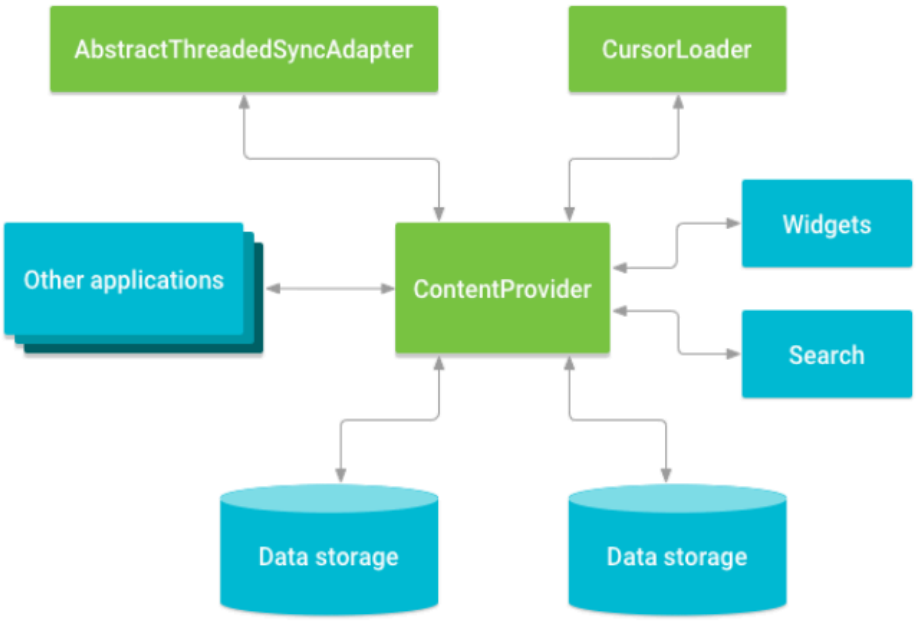

Abbildung 23: Content Provider

- Content Provider stellen für andere Applikationen Daten bereit
  - Daten stammen aus einer gekapselten DB, dem privaten Dateisystem oder werden on-the-fly erzeugt
  - Zugriff auf Daten über URI (Uniform Resource Identifier)
    - z.B. `content://hslu_notes/notes/4`
    - `Scheme` : immer gleich (`content`)
    - `Authority` : sollte ein Fully Qualified Name (FQN) sein (z. B. `hslu_notes`)
    - `Path` : beschreibt, welche Daten (z. B. `notes`)
    - Optional: `Item` : einzelner Datensatz (z. B. `4`)
- Zwei Arten von URIs:
  - **Pfad**: Bezeichnet Datenmenge (vgl. Verzeichnis mit Dateien)
  - **Item**: Einzelnes Datenelement (vgl. einzelne Datei)

8.2.1. Standard Content Providers

- Android stellt fertige Content Providers bereit:
  - **Kontakte**: Namen, Nummern, Emails
  - **SMS/MMS**: Empfangen, gesendet, Entwürfe
  - **Media Store**: Audio, Video, Bilder
  - **Settings**: Geräteeinstellungen
  - **Kalender**: Events, Erinnerungen, Teilnehmer
- Daten liegen meist in mehreren Tabellen

8.2.2. Content Resolver & Provider

- Zugriff auf Content Provider über **Content Resolver**
  - `requireActivity().contentResolver()`
  - **Bietet DB-Methoden**: `insert()` , `query()` , `update()` , `delete()` : `insert()` , `query()` , `update()` , `delete()`
  - **Streams**: openInputStream(uri), openOutputStream(uri)
- **Funktion**:

- URI auflösen ⟶ zuständigen Content Provider finden
- Interprozess-Kommunikation verwalten

- **Achtung**: ggf. Permissions nötig
  - `<uses-permission android:name="android.permission.READ_CALENDAR" />`

8.2.3. Zugriff auf Daten

8.2.3.1. Über Content Resolver + Query

- Datenabfrage erfolgt über den Content Resolver mit der Methode `query(...)`
- **Ablauf**:
  - Eine Activity oder ein Fragment nutzt einen `CursorLoader`
  - Dieser verwendet den `ContentResolver`, um Daten vom `ContentProvider` zu laden
  - Der `ContentProvider` greift auf die zugrunde liegende Datenspeicherung zu (z. B. SQLite)

```kotlin
// Queries the user dictionary and returns results
cursor = contentResolver.query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    projection,                         // The columns to return for each row
    selectionClause,                    // Selection criteria
    selectionArgs.toTypedArray(),       // Selection criteria
    sortOrder                           // The sort order for the returned rows
)
```

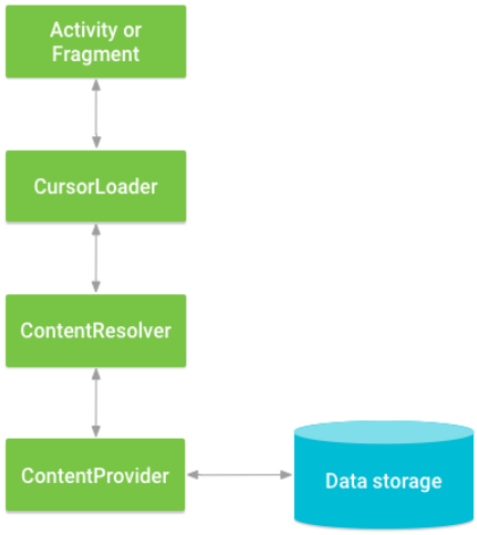

Abbildung 24: Zugriff auf Daten

8.2.3.2. ContentProvider Query vs SQL Query

Content Provider Query	SQL SELECT Query	Notes
`contentUri`	`FROM table_name`	Verweist auf Tabelle `table_name` im Content Provider
`projection`	`Col, col, col,…`	Spalten, die zurückgegeben werden sollen
`selection`	`WHERE col = value`	Auswahlkriterien für Zeilen
`selectionArgs`	(Kein direktes SQL-Äquivalent. Ersetzen Platzhalter `?` in `selection`)	-
`sortOrder`	`ORDER BY col, col,…`	Sortierreihenfolge der Ergebniszeilen

8.2.4. Content Provider verwenden

- **Wo fange ich an?**
  - Jeder Provider hat ein Standard-API, aber Content-URI, Projection etc. musst du kennen
- **Wo finden?**
  - Gute Doku für komplexe Provider wie:
    - Kalender
    - Kontakte
  - Weitere Infos unter `android.provider.*`:
    - Package-Doku
    - Contract-Klassen enthalten URIs und Spaltenkonstanten

8.2.5. Eigener Content Provider

- Eigene Klasse erbt von `ContentProvider`
- Wird beim App-Start geladen, Init in `onCreate()`
- CRUD-Methoden (insert, query, update, delete) ⟶ nicht alle nötig, z. B. für read-only Provider

8.2.6. Beispiel SMS Provider

- SMS des Systems sind über Content-Provider zugänglich
  - `android.provider.Telephony.Sms`
    - Sub-Provider für `Sent`, `Inbox`, `Draft`, etc.
  - Im Package `android.provider.*` befinden sich Contract-Klassen:
    - `Telephony.Sms` mit Hilfsklassen `BaseColumns` und `Telephony.TextBasedSmsColumns`
    - Diese enthalten Content-URIs und Spaltennamen für Projections
- `android.permission.READ_SMS` permission ist nötig
  - Muss beim User angefragt werden

8.2.6.1. Verwendung

```kotlin
private fun readSms(context: Context): List<String> {
    val smsList = mutableListOf<String>()
    context.contentResolver.query(
        Telephony.Sms.Inbox.CONTENT_URI,
        arrayOf(Telephony.Sms.Inbox._ID, Telephony.Sms.Inbox.BODY), // projection
        null,   // selection
        null,   // selection args
        null    // sort order
    )?.let { cursor ->
        val bodyIndex = cursor.getColumnIndex(Telephony.Sms.BODY)

        while (cursor.moveToNext()) {
            val body = cursor.getString(bodyIndex)
            smsList.add(body)
        }

        cursor.close()
    }
    return smsList
}
```

8.2.6.2. SMS im Emulator verschicken

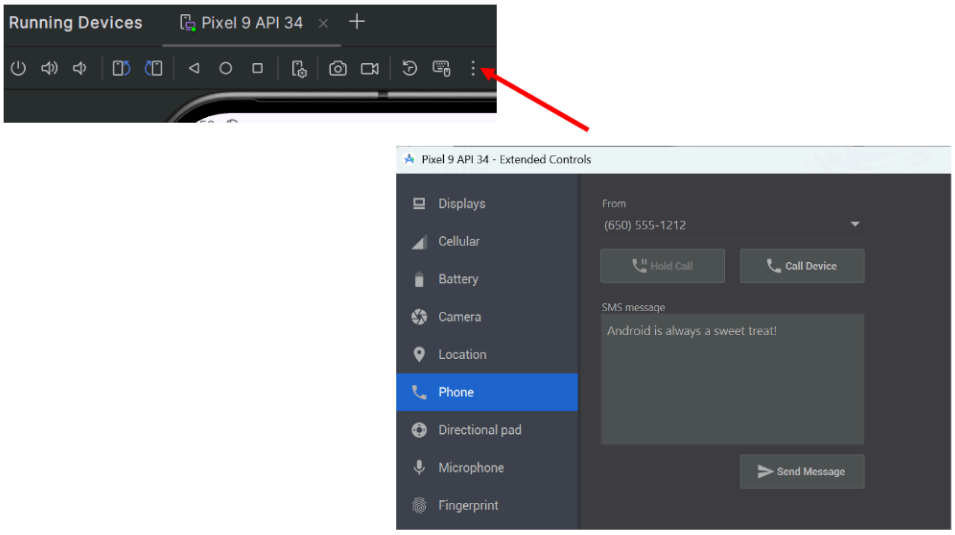

Abbildung 25: SMS im Emulator verschicken

8.3. Testing

8.3.1. Test Before Release

- Tests manuell & automatisch
- Manuell auf mehreren Geräten:
  - Verschiedene Auflösungen, Android-Versionen, Hersteller

➡️ **Fazit**: Viel Aufwand nötig, um breite Gerätekompatibilität zu sichern!

8.3.2. Testing Pyramide

- **E2E/UI-Tests**: wenige, da langsam
- **Integrationstests**: pro Modul, idealerweise pro E2E-Szenario
- **Unit-Tests**: viele, da schnell

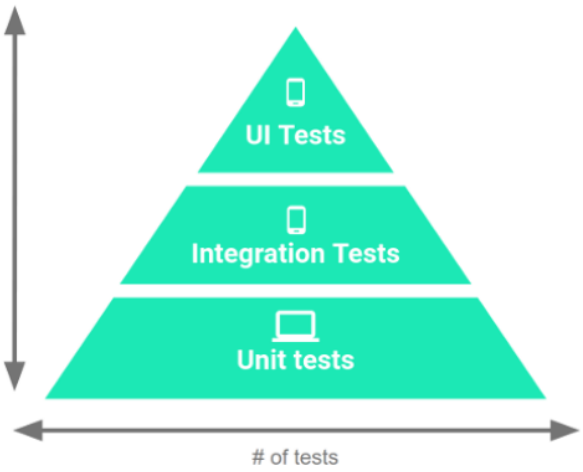

Abbildung 26: Testing Pyramide

- Je höher im Test-Pyramide-Modell ⟶ aufwendiger
- Je tiefer ⟶ mehr Tests, schneller & günstiger

8.3.3. Automatisiertes Testen

- **Gerätetypen**
  - **Real**: zuverlässig, aber langsam

    - **Emulator**: Balance zwischen real und simuliert
    - **Simuliert** (z. B. Robolectric): schnell, aber weniger zuverlässig
- **Testarten**
    - **Klassische Unit Tests**
        - Schnell, laufen lokal (`src/test/java/`)
    - **Robolectric Tests**
        - Simuliert Android-Umgebung
        - Etwas langsamer
        - Lokal auf der Entwickler Maschine
    - **Instrumentation Tests**
        - Langsam, auf Gerät oder Emulator (`src/androidTest/java/`)
- **Weitere Infos**
    - Android Testing Guide
    - Android Code Lab

8.3.3.1. Klassische Unit-Tests

- **Plattformunabhängige**
- Kein Android-Gerät oder Emulator notwendig
- Läuft rein auf der JVM mit JUnit

```kotlin
@Test
fun getActiveAndCompletedStats_noCompleted_returnsHundredZero() {
    val tasks = listOf(
        Task("title", "desc", isCompleted = false)
    )
    // When the list of tasks is computed with an active task
    val result = getActiveAndCompletedStats(tasks)

    // Then the percentages are 100 and 0
    assertThat(result.activeTasksPercent, `is`(100f))
    assertThat(result.completedTasksPercent, `is`(0f))
}
```

8.3.3.2. Unit Tests mit Coroutines

- **kotlinx-coroutines-test**: Ausführung von Coroutines in Tests
- **Mockito-Kotlin**: Mocking von Klassen mit `suspend` -Funktionen
- **Turbine**: Testen von `Coroutine-Flows`

8.3.3.2.1. Test suspend function

```kotlin
@OptIn(ExperimentalCoroutinesApi::class)
class UserViewModelTest {

    @Test
    fun testUpdateUser() = runTest { //runTest um Coroutiens aufzurufen
        // Main-Dispatcher ersetzen, damit viewModelScope sofort ausgeführt
        Dispatchers.setMain(UnconfinedTestDispatcher())

        // Repository mocken für ViewModel
        val userRepositoryMock = mock<UserRepository>()
        val user = User("Hans Muster", age = 35, authorized = false)

        // ViewModel mit Mock
        val viewModel = UserViewModel(userRepositoryMock)

        // Methode aufrufen
```

```
        viewModel.updateUser(user)

        // Aufrufe prüfen
        verify(userRepositoryMock).setUserName(user.name)
        verify(userRepositoryMock).setUserAge(user.age)
        verify(userRepositoryMock).setUserAuthorized(user.authorized)

        // Dispatcher zurücksetzen
        Dispatchers.resetMain()
    }
}
```

8.3.3.2.2. Test von Flow

```kotlin
@Test
fun testAllUsers() = runTest {
    // Beispiel-User
    val user = User("Hans Muster", age = 35, authorized = false)

    // Repository mocken und Flow zurückgeben
    val userRepositoryMock = mock<UserRepository> {
        on { getAllUsers() } doReturn flowOf(listOf(user))
    }

    // ViewModel mit Mock
    val viewModel = UserViewModel(userRepositoryMock)

    // Flow testen mit Turbine
    viewModel.allUsers.test {
        var users = awaitItem() // Erstes Item (z. B. leerer Zustand)
        Assert.assertTrue(users.isEmpty())

        users = awaitItem() // Nächstes Item im Flow
        Assert.assertTrue(users.size == 1)
        Assert.assertEquals(user, users.first())
    }
}
```

8.3.3.3. Robolectric

- Robolectric-Test: **Simuliert Android-Umgebung**
- Macht aus einem UI-Test einen Unit-Test

```kotlin
private lateinit var tasksViewModel: TasksViewModel

@Before
fun setupViewModel() {
    // Android-Kontext nur mit Robolectric möglich
    tasksViewModel = TasksViewModel(ApplicationProvider.getApplicationContext())
}

@Test
fun setFilterAllTasks_tasksAddViewVisible() {
    // Filter setzen
    tasksViewModel.setFiltering(TasksFilterType.ALL_TASKS)

    // Sichtbarkeit prüfen (LiveData-Test mit Hilfsmethode)
```

```
    assertThat(tasksViewModel.tasksAddViewVisible.getOrAwaitValue(), `is`(true))
}
```

8.3.3.4. Android Instrumentation-Tests

- Laufen im Emulator oder auf echtem Gerät
- Verwenden das echte Android-Laufzeitsystem
- Liegen im Source-Root: `src/androidTest/java`

8.3.3.4.1. UI-Tests mit Espresso (Verhaltenstests)

- „White box“ Framework für beobachtbares Verhalten
- Nur **Integrationstests**, Aktionen laufen **synchron**
- Nutzt **Hamcrest-Matcher**
- **Testablauf**:
  1. View finden
  2. Aktion ausführen
  3. Zustand prüfen
- ⚠️ **Funktioniert nicht mit Jetpack Compose!**

8.3.3.4.2. Beispiel

KOTLIN

```kotlin
class RegistrationEspressoTest {

    // Startet RegisterActivity vor jedem Test
    @Rule @JvmField
    val activityRule = ActivityScenarioRule(RegisterActivity::class.java)

    @Test
    fun registerOk_showsWelcomeTextAndRandomBackground() {
        // Szenario durchspielen (Robot)
        Espresso.onView(ViewMatchers.withHint("Username"))
            .perform(ViewActions.typeText("alfredo"))

        Espresso.onView(ViewMatchers.withHint("Password"))
            .perform(ViewActions.typeText("alFr.3d0"))

        Espresso.onView(ViewMatchers.withId(R.id.btn_register))
            .perform(ViewActions.click())

        // Ergebnis testen: Welcome-Text sichtbar
        Espresso.onView(ViewMatchers.withText("Welcome alfredo"))
            .check(ViewAssertions.matches(ViewMatchers.isDisplayed()))

        // Hintergrundbild prüfen (eigener Matcher)
        Espresso.onView(ViewMatchers.withId(R.id.img_avatar))
            .check(ViewAssertions.matches(withAnyDrawable()))
    }
}
```

8.3.3.4.3. Beispiel eigener Matcher

KOTLIN

```kotlin
object MyEspressoMatchers {

    // Factory-Methode für Matcher
    @JvmStatic
    fun withAnyDrawable(): DrawableMatcher {
        return DrawableMatcher(0)
```

```
    }

    // Matcher-Klasse: prüft, ob eine ImageView ein Drawable gesetzt hat
    class DrawableMatcher(private val expId: Int) :
TypeSafeMatcher<View>(View::class.java) {

        private var resourceName: String? = null

        override fun matchesSafely(item: View?): Boolean {
            if (item !is ImageView) return false

            val actualDrawable = item.drawable

            // Kein Drawable erwartet
            if (expId == -1) return actualDrawable == null

            // Irgendein Drawable erwartet
            if (expId == 0) return actualDrawable != null

            // Konkretes Drawable vergleichen
            resourceName = item.context.resources.getResourceEntryName(expId)
            val expectedDrawable = item.context.resources.getDrawable(expId)

            return drawableMatches(actualDrawable, expectedDrawable)
        }

        // → Beschreibung für Fehlerfall kommt (vermutlich) noch separat
    }
}
```

8.3.4. Zusatz

8.3.4.1. Weitere Testmöglichkeiten

- **Android-Testing**: gut dokumentiert in der Android-Doku
- **Weitere Tools**:
  - *UI-Automator*: ab API 18, app-übergreifend
  - *Monkey / Monkey Runner*: zufällige Eingaben
- **Build-Integration**: über Gradle, Emulator oder Gerät auf Build-Server

8.3.4.2. Cloud-Testing

- **Cloud Test Lab (Google)**
  - Tests auf echten Geräten in Google-Datenzentren
  - Integration in Android Studio
  - **Kostenpflichtig**, Google Cloud Account nötig
  - Mehr Infos

8.3.4.3. Browserstack

- Tests auf vielen Geräten & Android-Versionen
- Integration in Android Studio
- Upload eigener APKs möglich
- **Kostenpflichtig**, Account nötig
  - browserstack.com

9. Advanced Widgets 🧠

9.1. Dependency Injection

9.1.1. Zweck

- Übernimmt die Instanziierung von Klassen
- Komplexitätsreduktion
- Lebensdauerdefinition von Aktivitäten
- Reduziert Boilerplate Code
- Realisierung mittels unterschiedlicher Libraries
  - Koin
  - Dagger
  - Hilt

9.1.2. Setup

Es wird eine Applikationsklasse benötigt, welche mit der Annotation `@HiltAndroidApp` versehen ist. Hilt braucht einen Einstiegspunkt, dieser erstellt ein internes Gerüst, ein sogenannter Container. Diese Container ist lebenslang gültige (so lange die App lauft). Die anderen Hilt-Komponenten bekommen Zugriff auf diesen Container, um ihre Abhängigkeiten zu erhalten.

```kt
@HiltAndroidApp
class DemoApplication: Application()
```

Weiter muss die Applikations-Klasse im Manifest angegeben werden:

```xml
<application
  android:name=".components.DemoApplication"
```

9.1.3. Abhängigkeiten in Klassen / Field Injection

- Damit zur Laufzeit eine Android-Komponente DI verwenden kann, muss diese mit `@AndroidEntryPoint` annotiert werden
- Danach ist darin die Injektion mit `@Inject` möglich
- Auch ist dann möglich diese Android-Komponente an einem anderen Ort zu injecten.

```kt
@AndroidEntryPoint
class MainActivity: ComponentActivity() {
  @Inject lateinit var userRepository: UserRepository
}
```

Beachte, dass bei Android-Komponenten **Field-Injecten verwendet werden muss**, da die Klassen nicht von Hilt, sondern von Android instanziert werden.

`@Inject` Bedeutet, dass `userRepository` mittels DI bereitgestellt wird. (Hilt-Abhängigkeit).
`lateinit` Erlaubt, dass diese Variable zur Compile-Time nicht initialisiert ist, da sie ja durch die Dependency-Injection erst zur Laufzeit zugewiesen wird. Ohne `lateinit` würden wir eine Exception erhalten.

9.1.4. Konstruktorabhängigkeit

- **Nicht-Android-Klassen** können Abhängigkeiten **via Konstruktor** (Constructor-Injection) erhalten.

```kt
class UserRepository @Inject constructor (
  @ApplicationContext private val context: Context
) { ... }
```

`@Inject constructor` Definiert, dass ich die entsprechende Klasse / Objekt in den Konstruktor injecten möchte
`Context` Ist ein Beispiel einer Abhängigkeit, welche im Konstruktor von `UserRepository` gebraucht wird und von Hilt injected werden soll
`@ApplicationContext` Wir sagen, wir möchten gerne den Kontext der ganzen Applikation erhalten

9.1.5. ViewModels

- Special-Case, da sie nicht „normal“ instanziiert werden
- Bisher haben wir `ViewModels`, sofern diese Abhängigkeiten besessen haben, mit einer Factory erstellt
- Mittels Hilt können wir diese auch ohne Factory erstellen

```kotlin
@HiltViewModel
class UserViewModel @Inject constructor(
  private val userRepository: UserRepository,
) : ViewModel() {
```

```kotlin
val userViewModel = hiltViewModel<UserViewModel>()
```

erleichtert das leben sehr, da man nicht mehr das UserRepository holen muss

9.1.6. Möglichkeiten

Mit `Hilt` ist es möglich, in folgende Android-Klassen Abhängigkeiten zu Injektieren:

- Application
- View Model
- Activity
- Fragment
- View
- Service
- Broadcast Receiver

9.1.7. Module

- Können für Klassen ohne Konstruktoren verwendet werden (Interfaces oder Builder) wie z. B. Retrofit oder Database
- Klasse mit `@Modul` annotieren
- Komponente mit `@InstallIn()` annotieren, um anzugeben, wer eine Abhängigkeit schlussendlich verwenden darf
  `SingletonComponent::class` Überall dieselbe
  `ActivityComponent::class` Verwendung nur in Activities
  `ViewModelComponent::class` Verwendung nur innerhalb ViewModels
- Siehe auch Android Docs

9.1.7.1. `@Bind`

Mit `Bind` können wir auf einem Interface auf eine konkrete Implementation binden.

```kt
@Module
@InstallIn(SingletonComponent::class)
interface UserModule {
```

```
  @Binds
  fun userRepository(userRepositoryImpl: UserRepositoryImpl): UserRespository
}
```

- Impl = Interface, best practices das so zu benennen
- `@Binds` kann Interface oder abstrakte Klasse sein
- Bsp. für `@Bind`, fürs Testing, um Mock zu erstellen -> das ist ein sehr grosser Vorteil

9.1.7.2. `@Provides`

- Wird bei Abhängigkeiten verwendet, welche nicht per Konstruktor angeboten werden können
- Diese sind „Objects“ (Singleton, darf nur ein einziges mal existieren) oder „Companion Objects“ (static in Java) in anderen Modulen, sofern diese mit `@Bind` kombiniert werden
- Es ist möglich, in einem Interface, welches mit `@Bind` gebunden wird, mit `@Provides` weitere Objekte anzubieten

9.1.7.3. Beispiel

```kt
@Module
@InstallIn(SingletonComponent::class) // Gibt Verfügbarkeit der Komponente an
object RetrofitModule {
  private val contentType = "application/json".toMediaType()

  @Provides // Wir providen einen bandsService unter der
            // Verwendung von Retrofit
  fun bandsService(retrofit: Retrofit): BandsApiService {
    return retrofit.create(BandsApiService::class.java)
  }


  @Provides  // Wir definieren retrofit als Singleton um dies oben zu verwenden
  @Singleton // Scope, gibt an, wie lange das Objekt leben darf
  fun retrofit(): Retrofit {
    return Retrofit.Builder()
      .client(OkHttpClient().newBuilder().build())
      .addConverterFactory(Json.asConverterFactory(contentType))
      .baseUrl("https://whatever.ch/hslu/rock-bands/")
      .build()
  }
}
```

9.1.8. Scopes

- Abhängigkeiten sind per default unscoped
  - Jedes Mal, wenn eine Klasse eine neue Instanz braucht, wird eine neue erstellt
- Einer Abhängigkeit kann ein expliziter Scope mittels Annotation zugewiesen werden
- Beispiele für Scopes sind
  `Singleton` 1 Instanz für die ganze App
  `ViewModelScroped` Alle `ViewModels` derselben `Activity` teilen eine Instanz
- Sparsamer Umgang mit Scopes wird empfohlen

9.2. Bottom Navigation

- Navigationsleiste am unteren Bildschirmrand
- Zusatzinfos können via Badges angezeigt werden
- 2 verschiedene Zustände
  - Selektiert / Aktiv, ausgefülltes Icon mit Hintergrund
  - Selektiert / Inaktiv, Icon nur mit Rahmen
- unterteilt die App in verschiedene (primäre) Bereiche
  - meist 3 - 5 Elemente mit Button und Text
  - weniger: Tab-Layout vielleicht besser
  - mehr: nicht mehr richtig lesbar
- Wird über `Scaffold`-Parameter `bottomBar` gesetzt
- Beim Backstack sollte darauf geachtet werden, dass beim Druck auf ein Icon die Navigation nur ein mal im Back Stack landet

```kotlin
onClick = {
  selectedItem = Index
    navController.navigate(route = item.route) {
      launchSignleTop = true //geliche Screen soll nicht mehrmals hinzugefügt werden
      popUpTo(navController.graph.findStartDestination().id) // History(Backstack) soll zurückgesetzt werden. Nur StartScreen beleibt
    }
},
```

9.2.1. Beispiel

9.2.1.1. Navigationselemente definieren

```kotlin
data class BottomNavigationItem(
  val route: String,
  val title: String,
  val selectedIcon: ImageVector,
  val unselectedIcon: ImageVector,
  val hasNews: Boolean,
  val badgeCount: Int? = null
)

// in main activity

BottomNavigationItem(
  route = DemoApplicationScreen.Home.name,
  title = DemoApplicationScreen.Home.name,
  selectedIcon = Icons.Filled.Home,
  unselectedIcon = Icons.Outlined.Home,
  hasNews = false
),

BottomNavigationItem(
  route = DemoApplicationScreen.Components.name,
  title = DemoApplicationScreen.Components.name,
  selectedIcon = Icons.Filled.Star,
  unselectedIcon = Icons.Outlined.Star,
  hasNews = true,
),

BottomNavigationItem(
  route = DemoApplicationScreen.User.name,
  title = DemoApplicationScreen.User.name,
```

```
  selectedIcon = Icons.Filled.Face,
  unselectedIcon = Icons.Outlined.Face,
  hasNews = false,
  badgeCount = 5
)
```

9.2.1.2. NavigationsBar in Scaffold

```kotlin
Scaffold( // Rahmen
  modifier = Modifier
  .imePadding() //Positionierung über Keybord falsl angezeigt
  .fillMaxSize(),
  bottomBar = { // Parameter für Buttom Bar
    BottomNavigation( //Button Implementierung
      navController = navController,
      items = navigationItems
    )
  }
) { innerPadding ->
  DemoAppNavHost(
    navController = navController,
    modifier = Modifier.padding(innerPadding),
  )
}
```

9.2.1.3. NavigationsBar und Items

```kotlin
@Composable
fun BottomNavigation(
  navController: NavHostController,
  items: List<BottomNavigationItem>,
) {
  var selectedItem by remember { // Zustand welches Item selektiert ist
    mutableIntStateOf(0)
}
  NavigationBar {
    items.forEachIndexed { index, item ->
```

9.2.1.4. Manipulations des Backstacks

```kotlin
onClick = {
  selectedItem = Index
    navController.navigate(route = item.route) {
      launchSignleTop = true //geliche Screen soll nicht mehrmals hinzugefügt werden
      popUpTo(navController.graph.findStartDestination().id) // History(Backstack) soll
zurückgesetzt werden. Nur StartScreen beleibt
    }
  },
```

9.2.1.5. NavigationBarItem mit Badge

```kotlin
icon = {
  BadgedBox(
    badge = {
      when {
```

```
        item.badgeCount != null -> {
          Badge { // Badge mit Counter
            Text(text = item.badgeCount.toString())
          }
        }
        item.hasNews -> {
          Badge() // nur roter Punkt, has news
        }
      }
    }
  ) { Icon(
    imageVector = if (index == selectedItem) {
      item.selectedIcon
    } else {
      item.unselectedIcon
    },
    contentDescription = item.title
    )
  }
}
```

9.2.1.6. Badge dynamisch laden

Zustand für Liste von ButtonNavigationsItem's machen:

KOTLIN

```
val navigationItems = remember {
  mutableStateListOf(
    BottomNavigationItem(...),
    BottomNavigationItem(...),
    BottomNavigationItem(...)
  )
}
```

Zustand nicht nach „unten" geben (state hoisting), Funktion verwenden

KOTLIN

```
@Composable
fun UserScreen(
  onUpdateBadgeCount: (Int) -> Unit
)

val users by userViewModel.allUsers.collectAsState()
if (users.isNotEmpty()) {
  onUpdateBadgeCount.invoke(users.size)
}

@Composable
fun DemoAppNavHost(
  navController: NavHostController,
  modifier: Modifier = Modifier,
  onUpdateUserBadgeCount: (Int) -> Unit
)
```

KOTLIN

```
DemoAppNavHost(
  navController = navController,
  modifier = Modifier.padding(innerPadding),
) { userUpdatedBadgeCount ->
```

```
  val oldItem = navigationItems[2] //fixer Index, da bekannt welche Position
  navigationItems[2] =
    oldItem.copy(badgeCount = userUpdatedBadgeCount) // Copy-Funktion von data Klassen.
Erlaubt Erstellung von neuen Instanzen basierend auf bestehenden Instanzen mit
gewünschten Änderungen, gegeben da Data class
}
```

9.3. Top App Bar

- Oben am Bildschirm
- Zeigt in der Regel den Titel des Bildschirms an
- Aktionsbutton, um zurück zu navigieren
- Zusätzliche Aktionen per Icon
- In unterschiedlichen Grössen verfügbar
- Ebenfalls über `Scaffold` hinzufügbar
- Positionierung und Platzverteilung via Scaffold

9.3.1. Beispiel

KT

```
TopAppBar(
    title = {
        currentDestination?.let { destination ->
            val title = when (destination) {
                Screens.Home.name -> "Home"
                // [...]
                else -> ""
            }
            Text(text = title)
        }
    },
    navigationIcon = {
        if (!topLevelScreens.contains(currentDestination)) {
            IconButton(
                onClick = { navController.popBackStack() }
            ) {
                Icon(
                    imageVector = Icons.AutoMirrored.Filled.ArrowBack,
                    contentDescription = "Go back"
                )
            }
        }
    },
    actions = {
        IconButton(
            onClick = {}
        ) {
            Icon(
                imageVector = Icons.Default.Check,
                contentDescription = "check"
            )
        }
    }
  )
```

10. XML UI 👴

10.1. Aufbau

- **1 Activity** (MainActivity wie in Compose)
- **Fragments** angezeigt (anstelle Compose Functions)
- Activity & Fragments in **XML-Layout**
- mehrere Fragments in einer Activity

imperatives Design

- Schrittweise Anweisungen
- UI Elemente gesucht/gefunden
- UI Elemente verändert

Compose deklarativ

- Beschreibung des Zustands

Jedes Layout in eigener Datei

- Root: `View` oder `ViewGroup`
- Standard- & eigene View-Klassen

Mit `Inflater` **instanziierbar** („aufblasen“)

- Wiederverwendbare Komponenten möglich

10.2. Fragment

- Wiederverwendbarer Teil einer App
- eigenes Layout
- braucht Host (Activity/ Fragment)
- Activity & Fragment eigenen Lifecycle
- Fragment benötigt immer Host Activity

10.3. Lifecycle Activity + Fragment

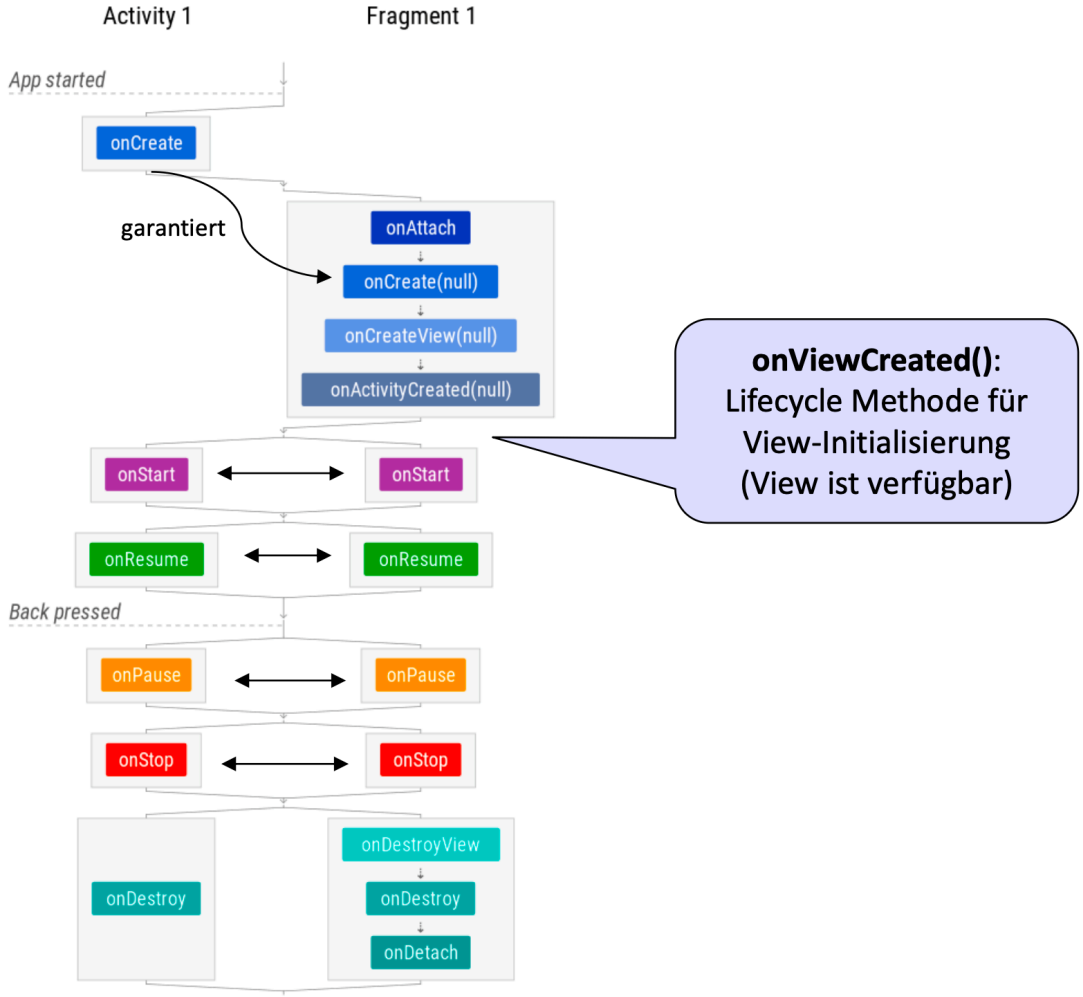

Abbildung 27: Lifecycle Activity + Fragment

10.4. Zusammenhang Activity + Fragment

- Verwende mehrere Fragments in einer Activity
- Activity hat einen eigenen Lifecycle
- Fragment hat einen eigenen Lifecycle
- Fragment benötigt immer eine Host Activity

10.5. View Binding Activity

Hinzufügen im build.gradle (Module :app)

```
android {
  ...
  buildFeatures {
    viewBinding = true
  }
}
```

```kt
import ch.hslu.testapplication.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
```

```
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

10.6. View Binding Fragment

```kt
import ch.hslu.testapplication.databinding.FragmentFirstBinding

class FirstFragment : Fragment() {

    private var _binding: FragmentFirstBinding? = null

    // Diese Property ist nur zwischen onCreateView und onDestroyView gueltig
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentFirstBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

10.7. Fragment Manager

Verwaltung von Fragmenten

- Hinzufügen, Entfernen, Ersetzen und Verstecken von Fragmenten
- Verwaltung des Backstacks (zurück-Navigation)
- Kommunikation zwischen Fragmenten

2 Varianten:

in Acitivty mit `supportFragmentManger`

im Fragment

- Wechsel Fragment auf Activity ⟶ parentFragmentManger
- View im Fragment mit einem Fragment ersetzt wird (ViewPager) ⟶ childFragmentManger

10.8. Fragment in Activity anzeigen

```kt
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        supportFragmentManager
            .beginTransaction()
            .replace(R.id.fragment_container_view, FirstFragment())
            .commit()
    }
}
```

Die `id` muss gesetzt sein, um in der Activity zugreifen zu können.

```XML
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragment_container_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_toTopOf="parent" />
```

10.9. UI Gestaltung

10.9.1. Grundkonzepte

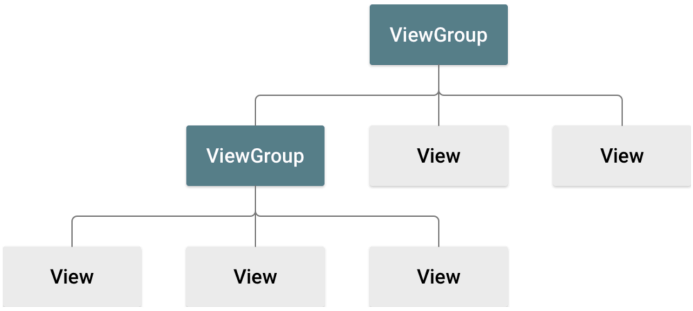

Abbildung 28: Grundkonzepte Android UI - Viewgroup

Android UI…

- ist hierarchisch aufgebaut
- besteht aus
  - ViewGroups
  - Behälter für Views und andere ViewGroups
  - Anordnung durch ein Layout
  - Views (Widgets)
- sollte auf unterschiedlichen Bildschirmgrössen gleich aussehen!

10.9.2. Constraint Layout

- Komplexe Layouts ohne Verschachtelung
- Relative Platzierung mit Bedingungen:
  - zu anderen Elementen
  - zum Parent-Container
  - Element-Chains (spread/pack)
- Layout-Hilfen (Klassen):
  - Hilfslinien

- Barriers

10.9.3. Linear Layout

Reiht Elemente neben-/untereinander auf

- Kann geschachtelt werden, um Zeilen/Spalten zu formen

Eigenschaften

- `orientation`, `gravity`, `weightSum`

Layout-Parameter für Children

- `layout_width`, `layout_height`
- `layout_margin`, ...
- `layout_weight`, `layout_gravity`

10.9.4. Constraint Kette

Views die nebeneinander auf einer Ebene (horizontal/vertikal) beide Constraints angeben bilden eine Kette (**Chain**)

- Verknüpfung wird im Code gemacht

```xml
<RadioButton
    android:id="@+id/radioButton1"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="radio1"
    app:layout_constraintEnd_toStartOf="@+id/radioButton2" <!-- Verknüpfung -->
    app:layout_constraintStart_toStartOf="parent" <!-- Start der Kette -->
    app:layout_constraintTop_toBottomOf="@+id/edit_text" />

<RadioButton
    android:id="@+id/radioButton2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="radio2"
    app:layout_constraintEnd_toStartOf="@+id/radioButton3" <!-- Verknüpfung -->
    app:layout_constraintStart_toEndOf="@+id/radioButton1" <!-- Verknüpfung -->
    app:layout_constraintTop_toBottomOf="@+id/edit_text" />

<RadioButton
    android:id="@+id/radioButton3"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="radio3"
    app:layout_constraintEnd_toEndOf="parent" <!-- Ende der Kette -->
    app:layout_constraintStart_toEndOf="@+id/radioButton2" <!-- Verknüpfung -->
    app:layout_constraintTop_toBottomOf="@+id/edit_text" />
```

10.10. Interaktion mit GUI

Views in Java/Kotlin

- Jedes View-Element hat entsprechende Java-Klasse
- Gilt auch für ViewGroups
- Layout könnte auch programmiert werden

Im Code können Views mittel der Binding-Klasse referenziert werden z.B:

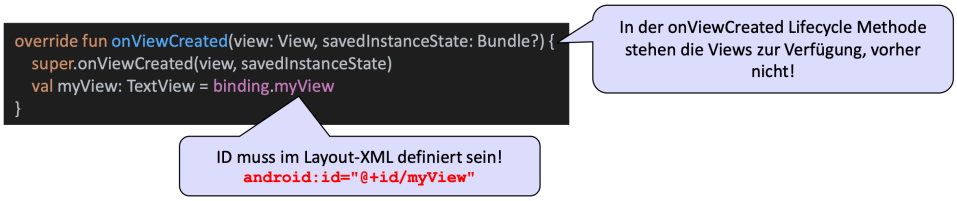

Abbildung 29: Grundkonzepte Android UI - Viewgroup

10.10.1. GUI Events

Entwurfsmuster: Observer / Listener

- Listener für entsprechenden Event bei View registrieren, z.B. bei myButton: Button: `myButton.setOnClickListener(listener)`

Event und Listener-Typen

- `OnClickListener, OnLongClickListener, OnKeyListener,OnTouchListener, OnDragListener, ...`