

MOBILE SOFTWARE ENGINEERING

AM BEISPIEL DER ANDROID PLATTFORM

WORUM GEHT ES HIER EIGENTLICH? DIE LERNINHALTE

- Grundlagen der mobilen Software-Entwicklung
- Android-Systemarchitektur
- Entwicklungsumgebung
- Kernkomponenten einer Android-App
- Interaktion zwischen Anwendungskomponenten
- Fortgeschrittene Techniken

WORUM GEHT ES HIER EIGENTLICH?

DIE QUALIFIKATIONSZIELE

Nach erfolgreichem Abschluss sind die Studierenden in der Lage:

- die Unterschiede und Besonderheiten der SW-Entwicklung für mobile Systeme zu erkennen und diese zu erläutern.
- verschiedene Aktivitäten, Rollen und Risiken bei Erstellung, Betrieb und Wartung von mobilen Software-Systemen zu unterscheiden.
- Architektur und technische Eigenschaften der Android Plattform zu erläutern und zu unterscheiden.
- selbstständig mobile Software-Systeme zur Lösung von konkreten Problemen für die Plattform „Android“ zu erstellen.

WIE WIRD GEPRÜFT

Typ: Klausur

Inhalt: Theorie und Praxis, MC und Freitextaufgaben

Datum: Siehe MyCampus

AGENDA FÜR DIESES THEMA

Grundlagen der mobilen Softwarearchitektur

Android-Systemarchitektur

Entwicklungsumgebung

Kernkomponenten der Android-App

Interaktion zwischen Anwendungskomponenten

Fortgeschrittene Techniken

1

2

3

4

5

6

01

GRUNDLAGEN DER MOBILEN SOFTWARE-ENTWICKLUNG

WORUM GEHT ES IN DIESEM KAPITEL?

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- welche Besonderheiten mobile Endgeräte besitzen.
- welche Auswirkungen diese Besonderheiten auf die Software-Entwicklung für die Zielgruppe mobile Anwendungen haben.
- in welche Kategorien mobile Endgeräte unterteilt werden.
- was die Android-Plattform ist.

01

GRUNDLAGEN DER MOBILEN SOFTWARE-ENTWICKLUNG

BESONDERHEITEN VON MOBILEN ENDGERÄTEN

Verlagerung auf mobile Endgeräte:

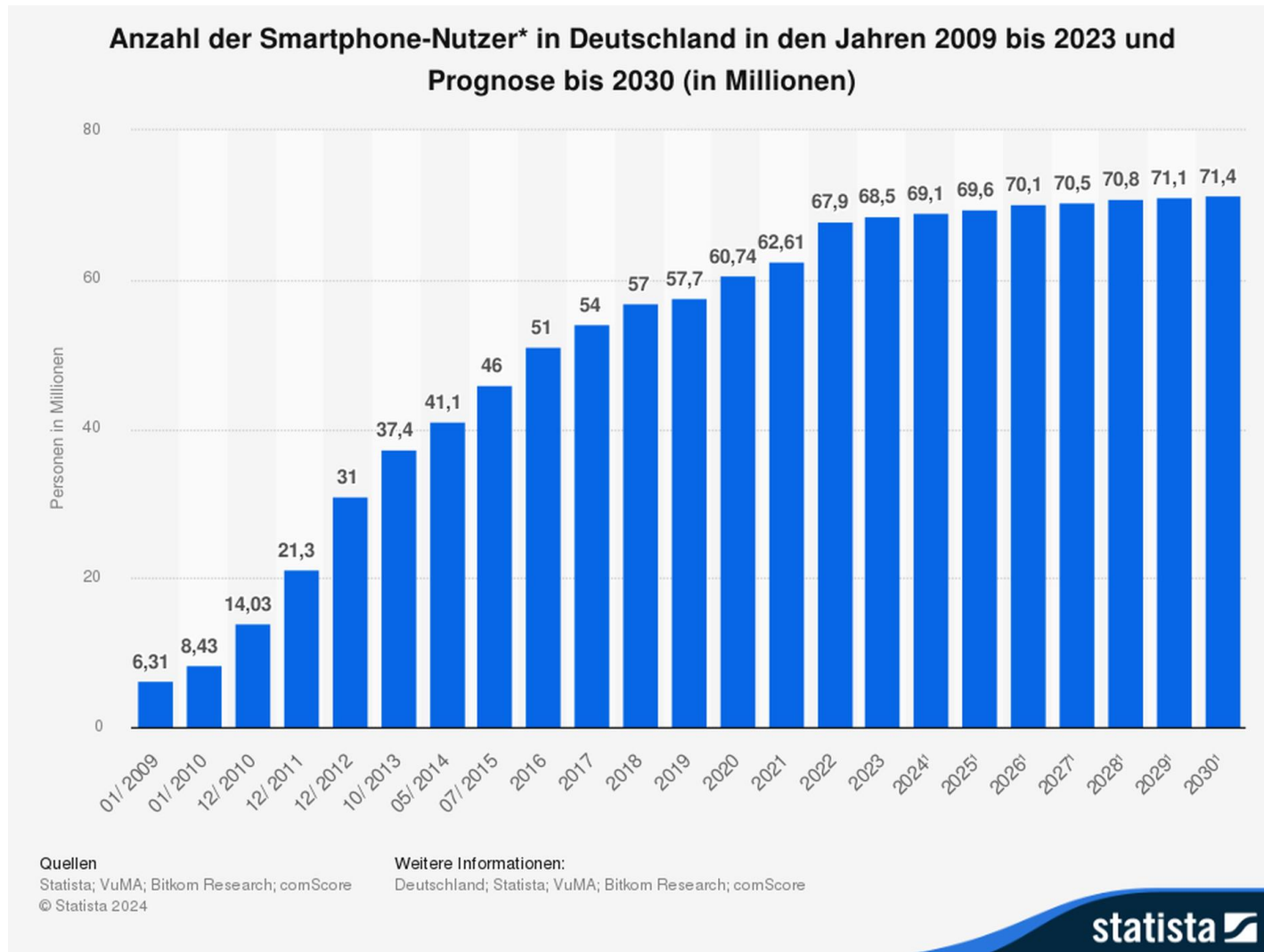
- Rückgang der Verkaufszahlen stationärer PCs, steigende Verkäufe mobiler Geräte.
- Mobile Endgeräte dominieren die Internetnutzung.

Evolution mobiler Technologien:

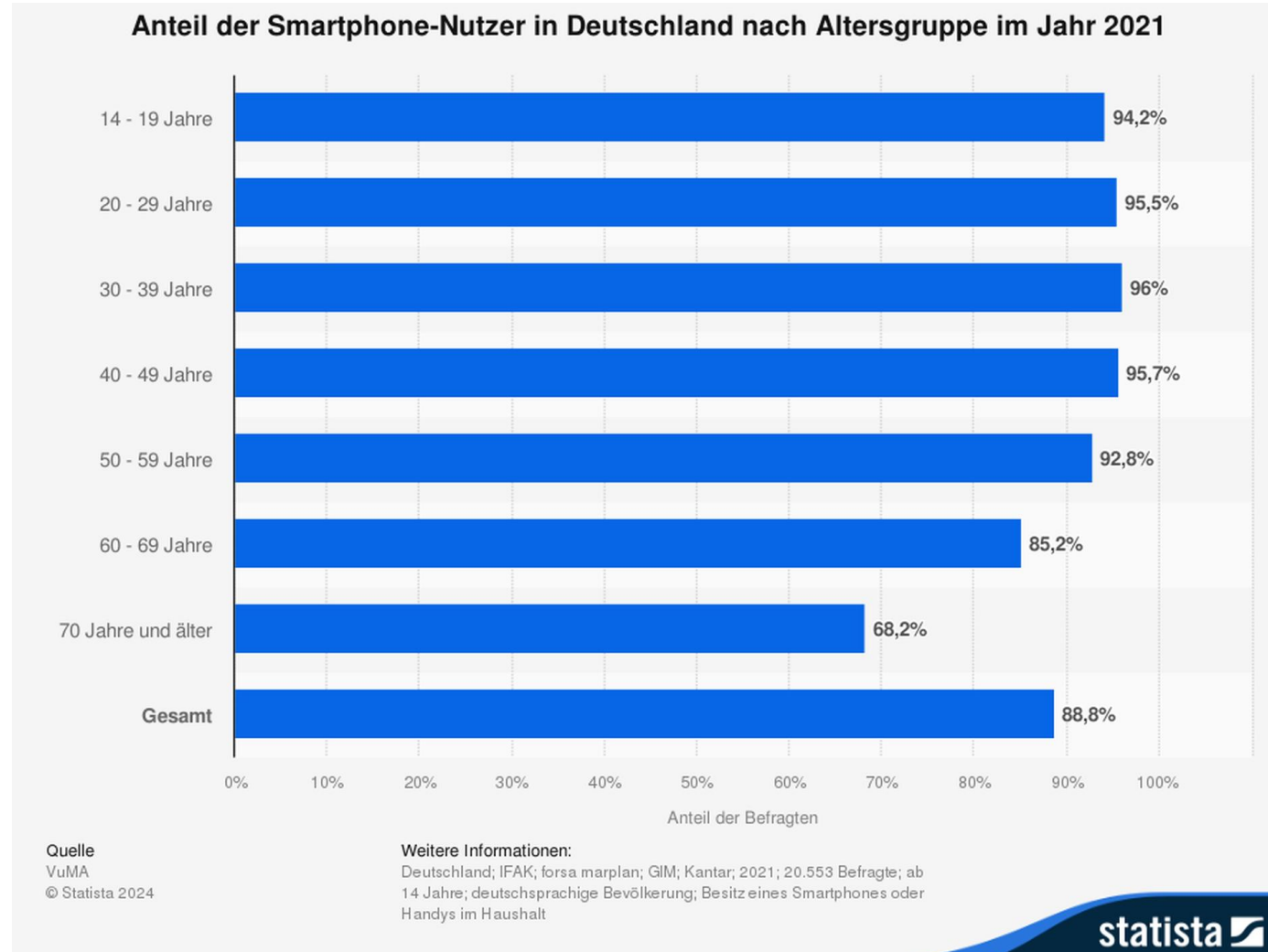
- Frühe mobile Geräte boten nur Browserzugriff auf Desktop-Webseiten.
- Fortschritte durch größere Touchscreens und Touch-Funktionalität.

„Mobile First“ hat gesiegt

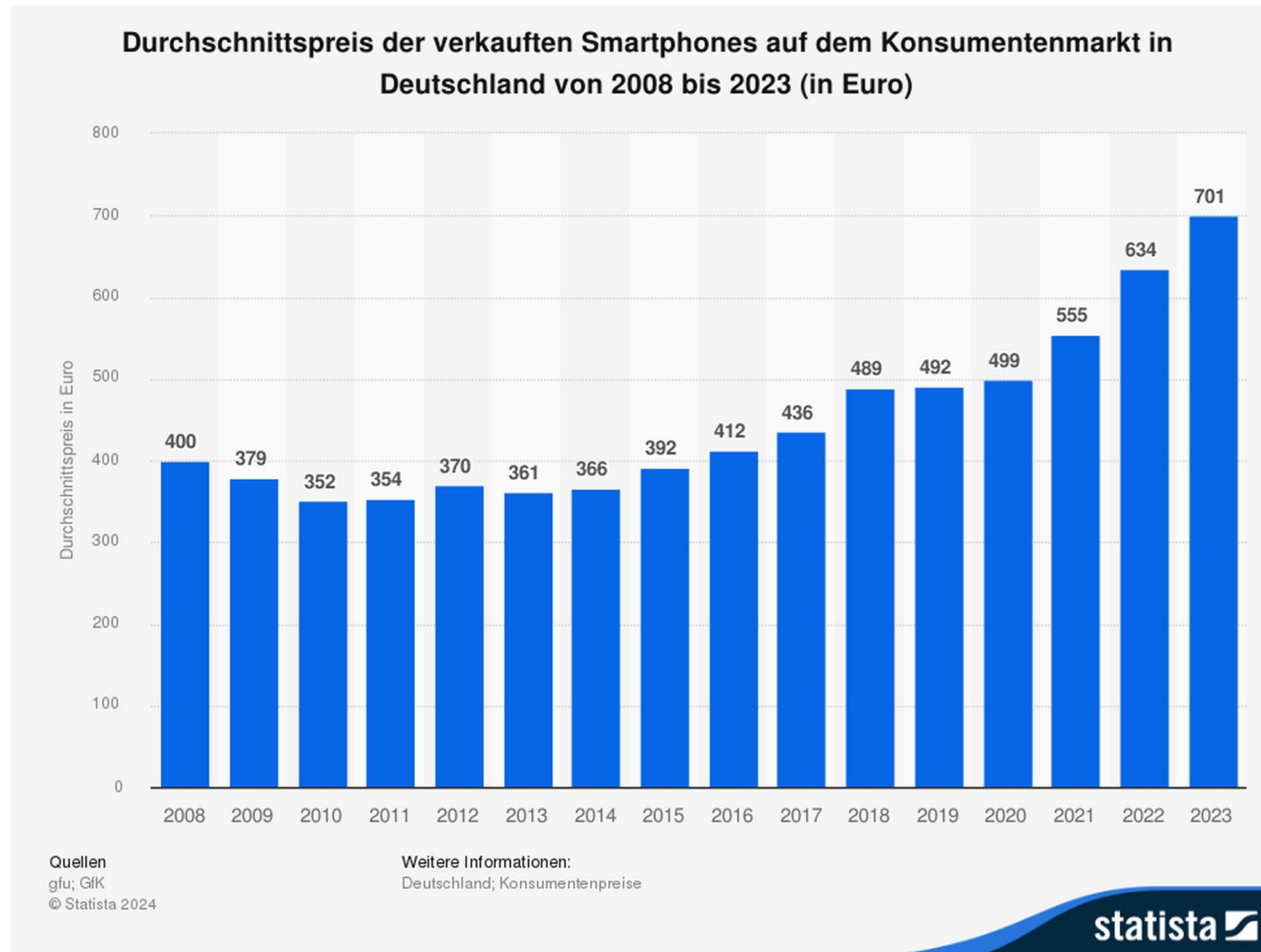
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



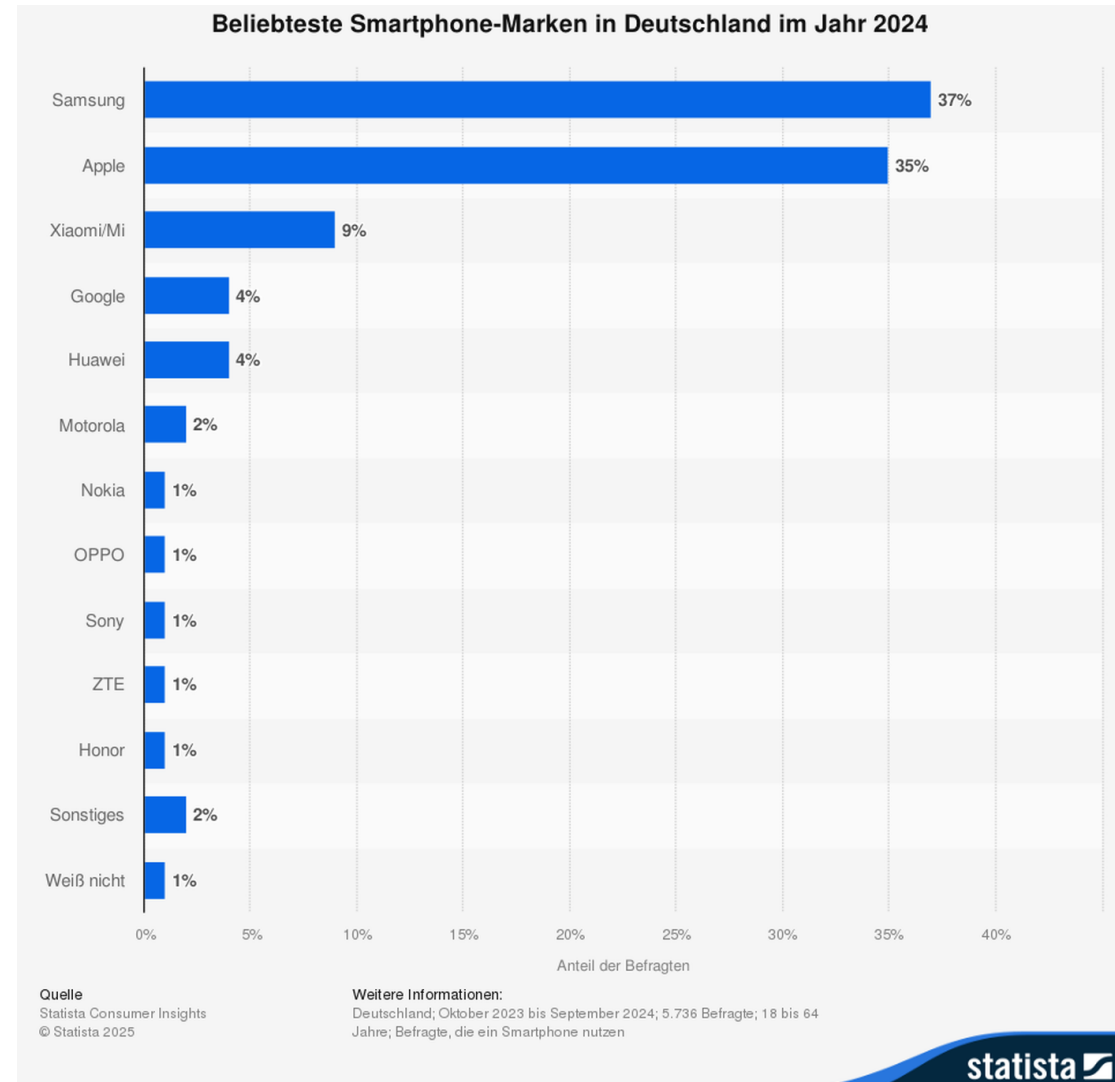
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT

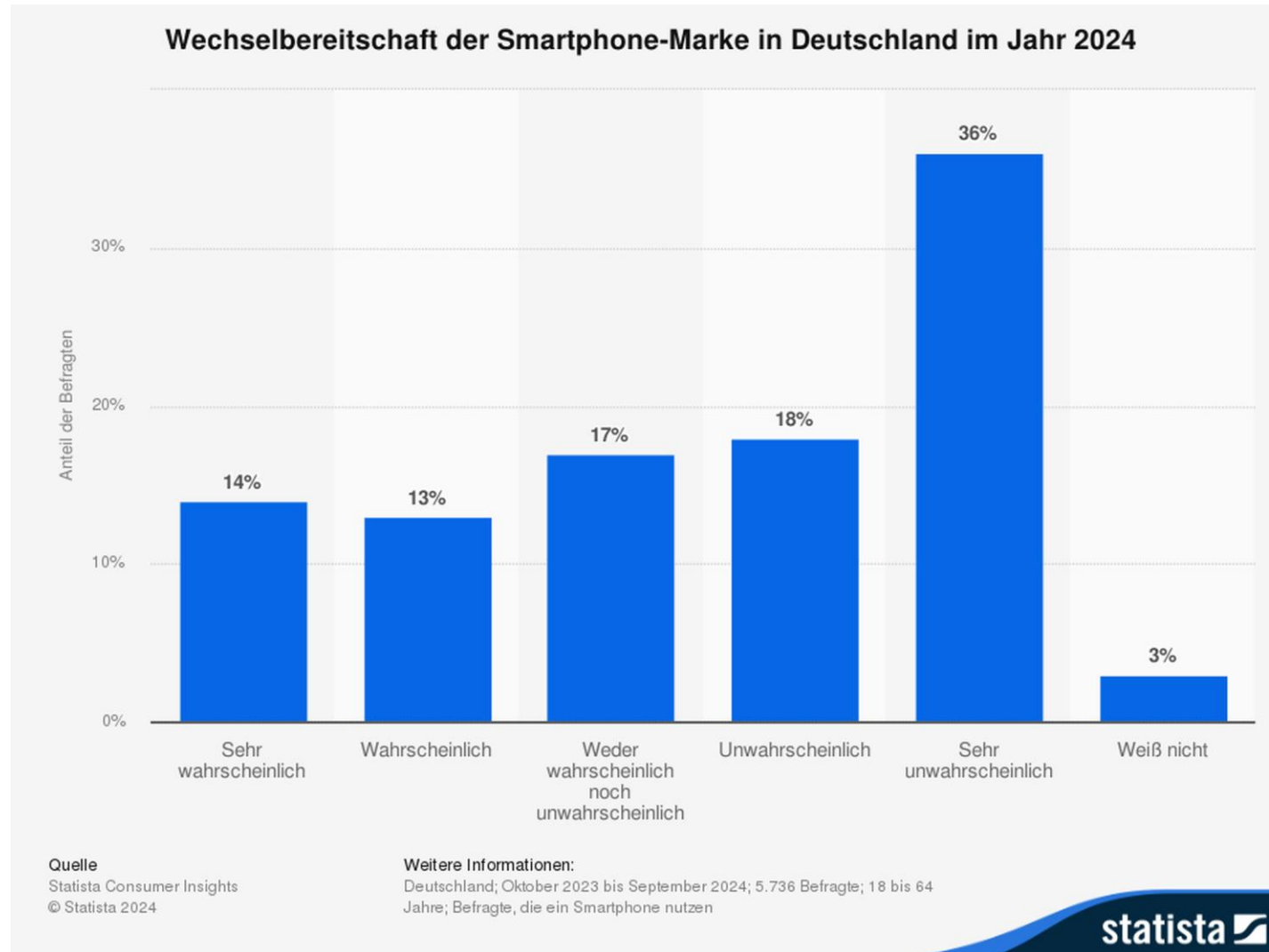


BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT

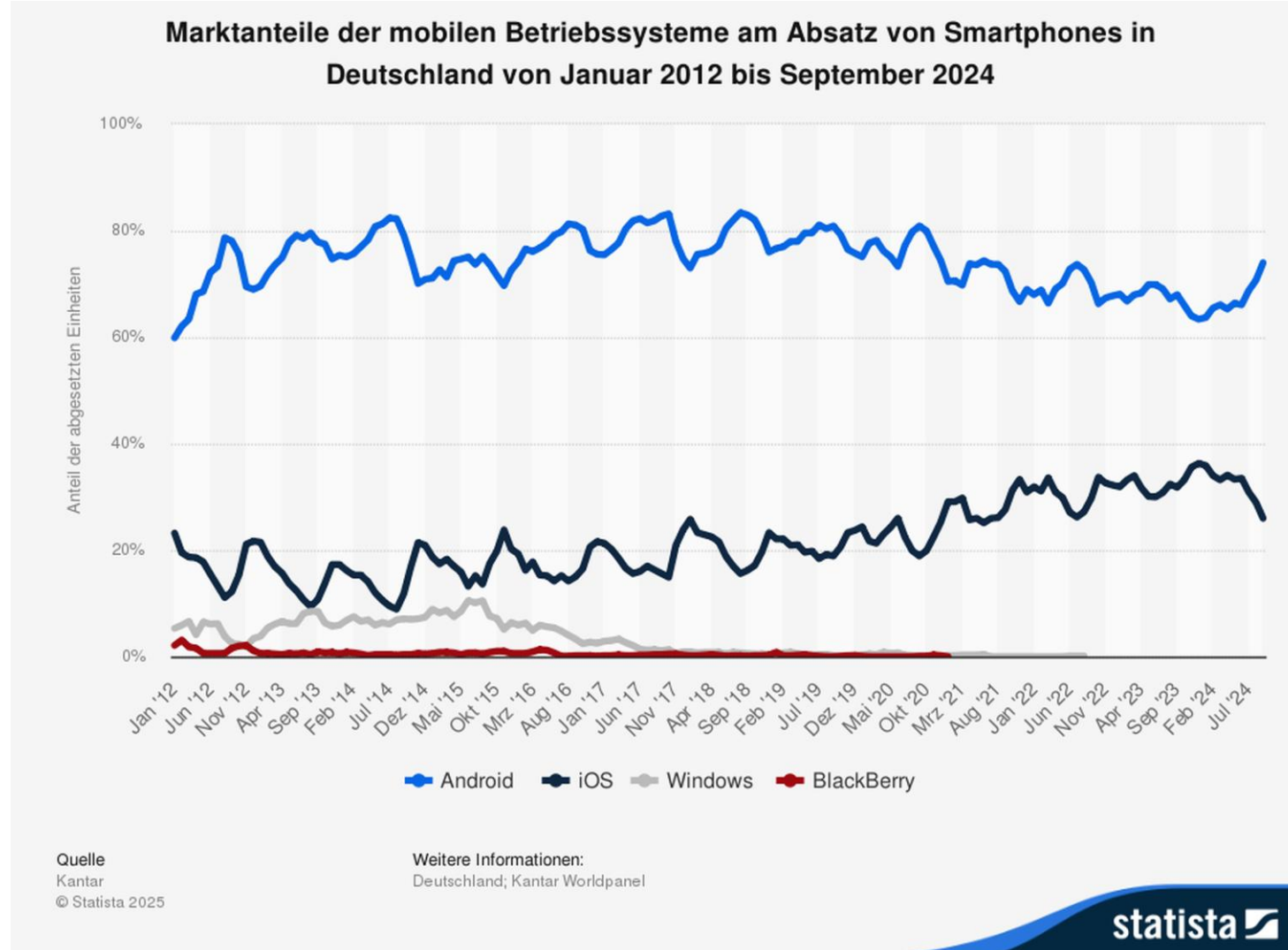


BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT

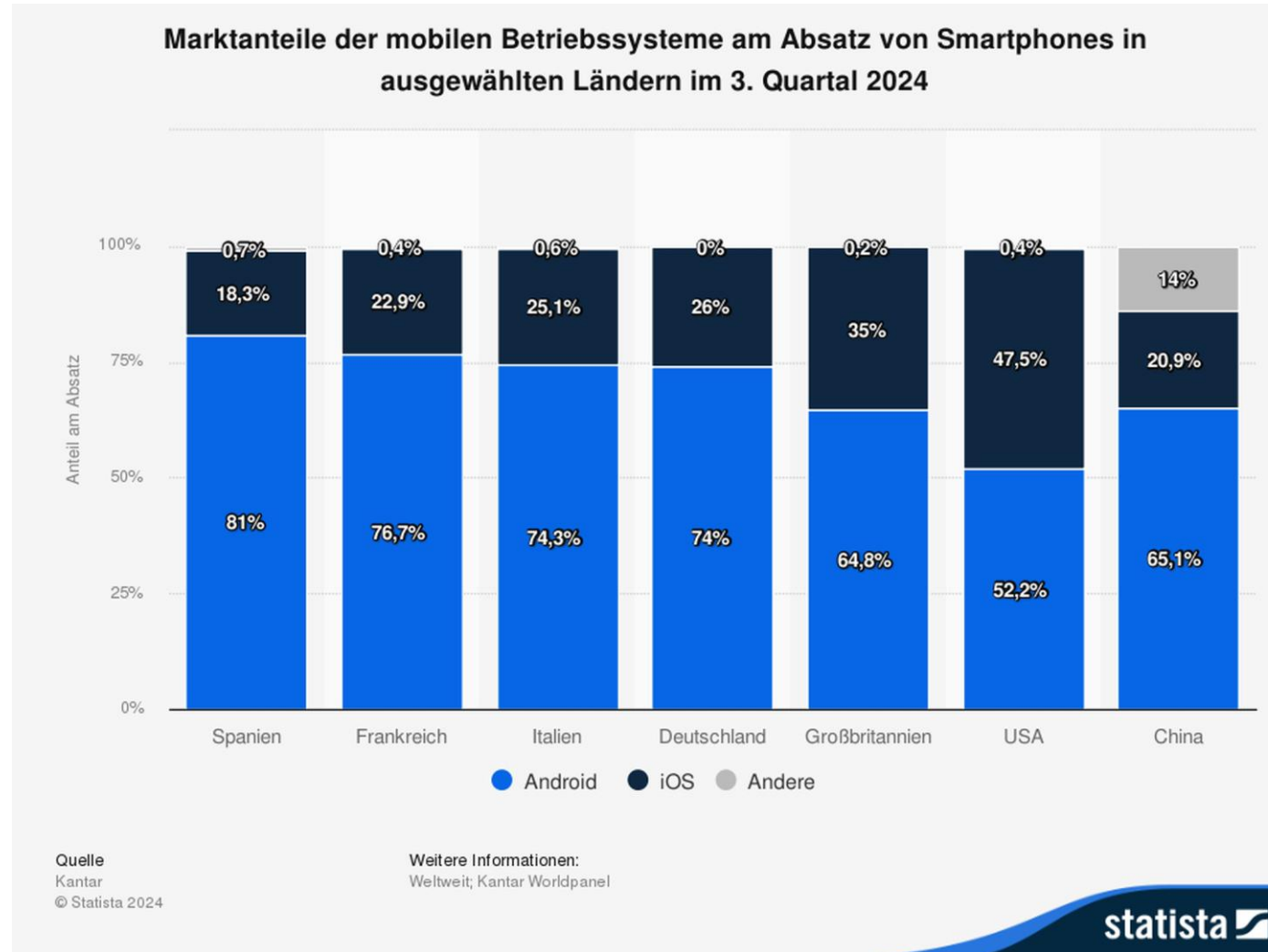




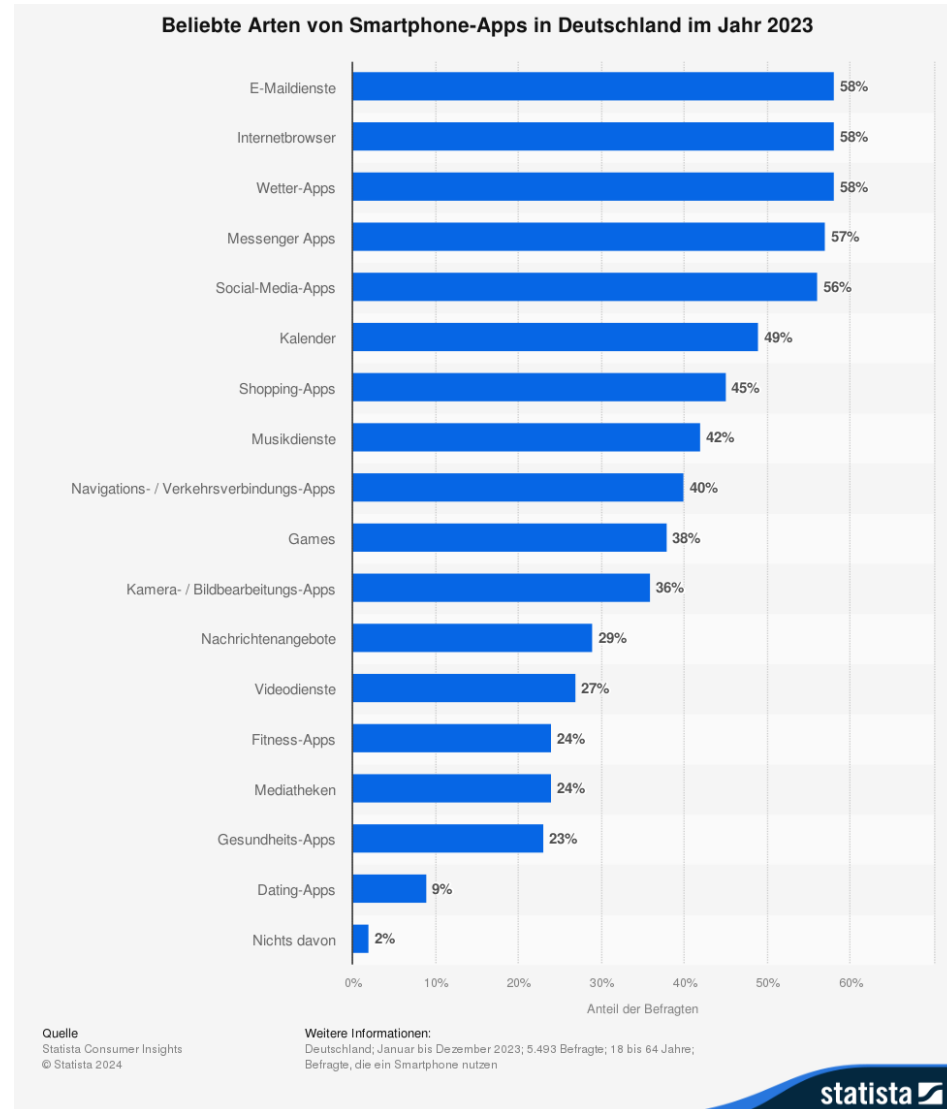
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



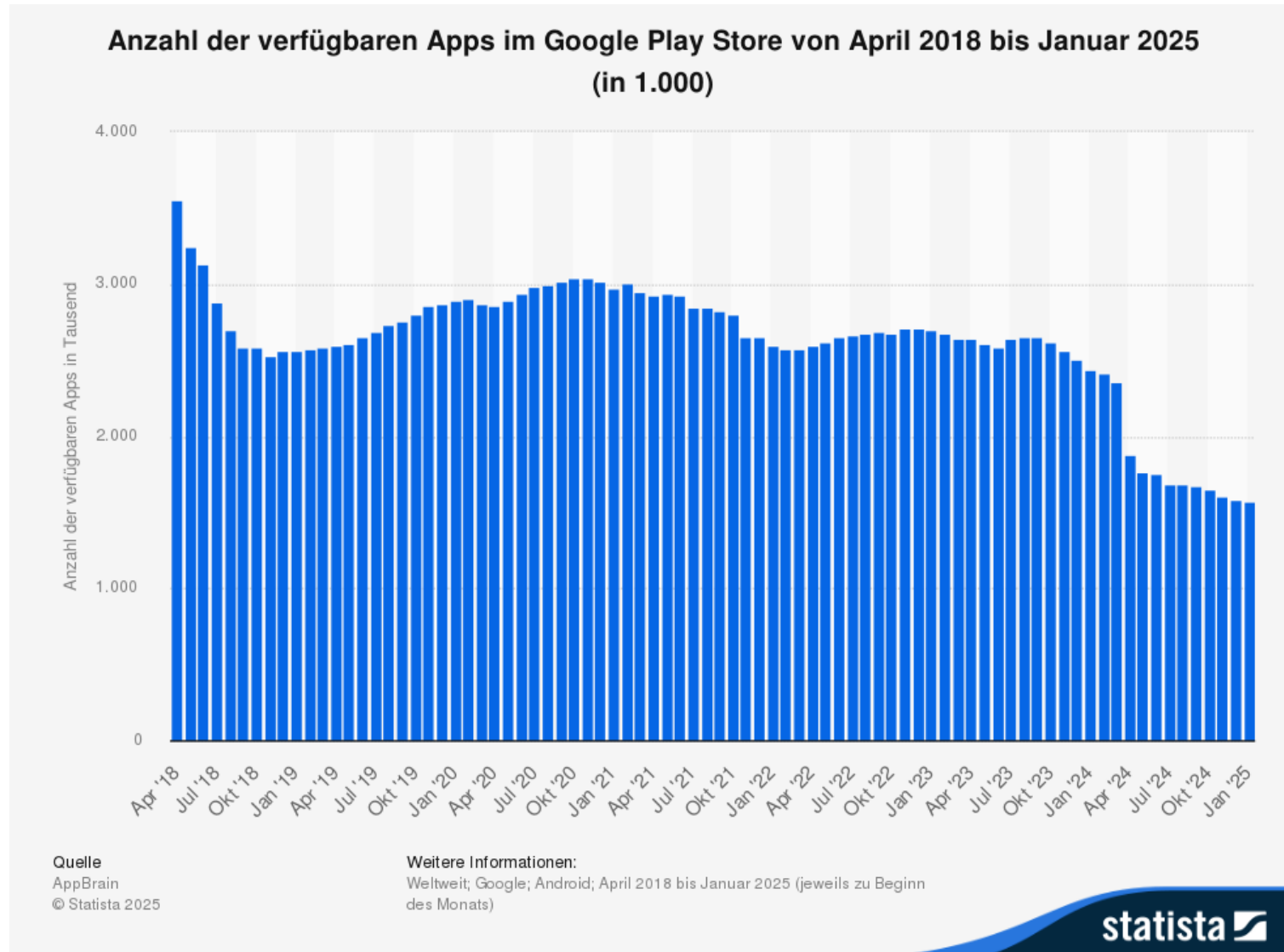
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



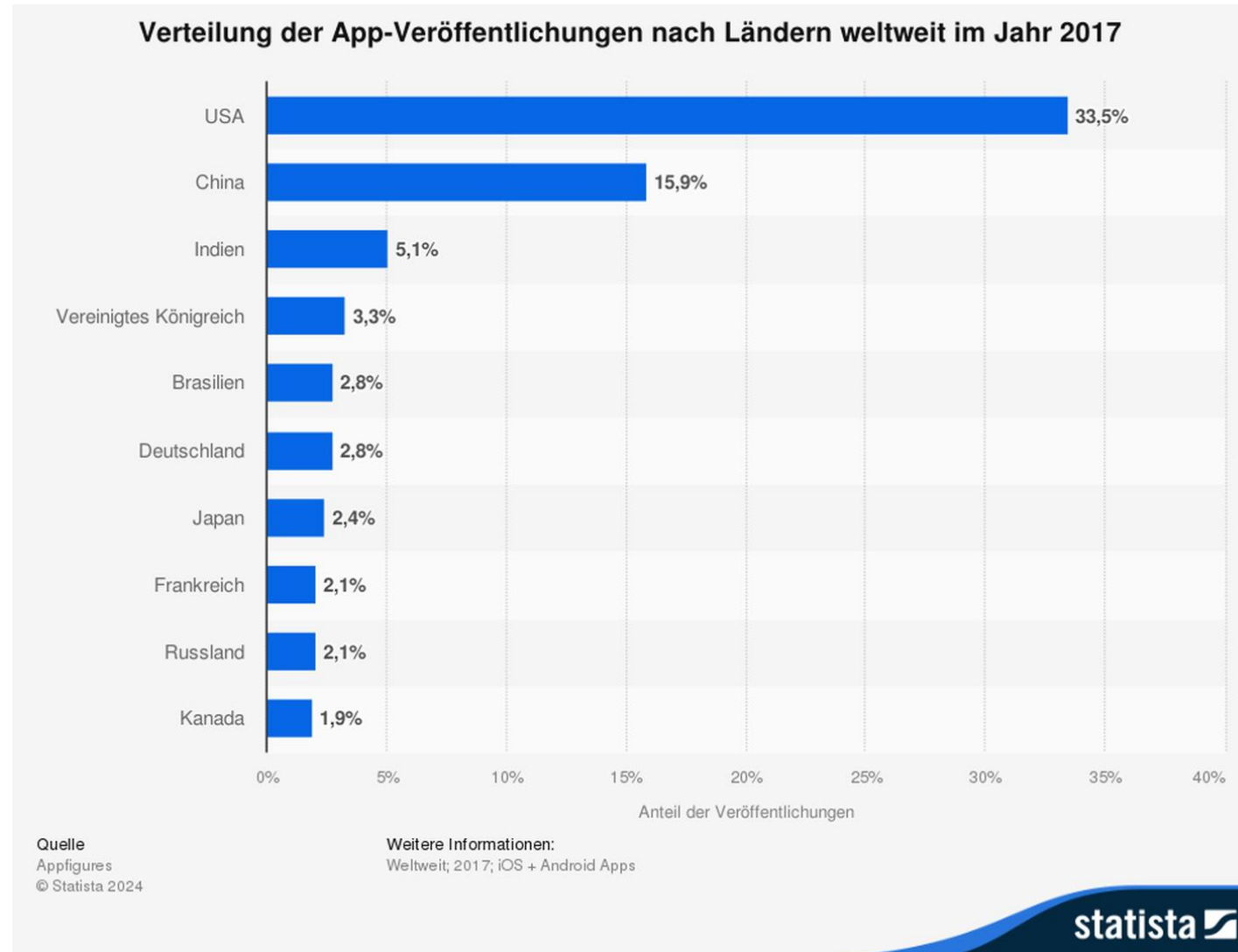
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



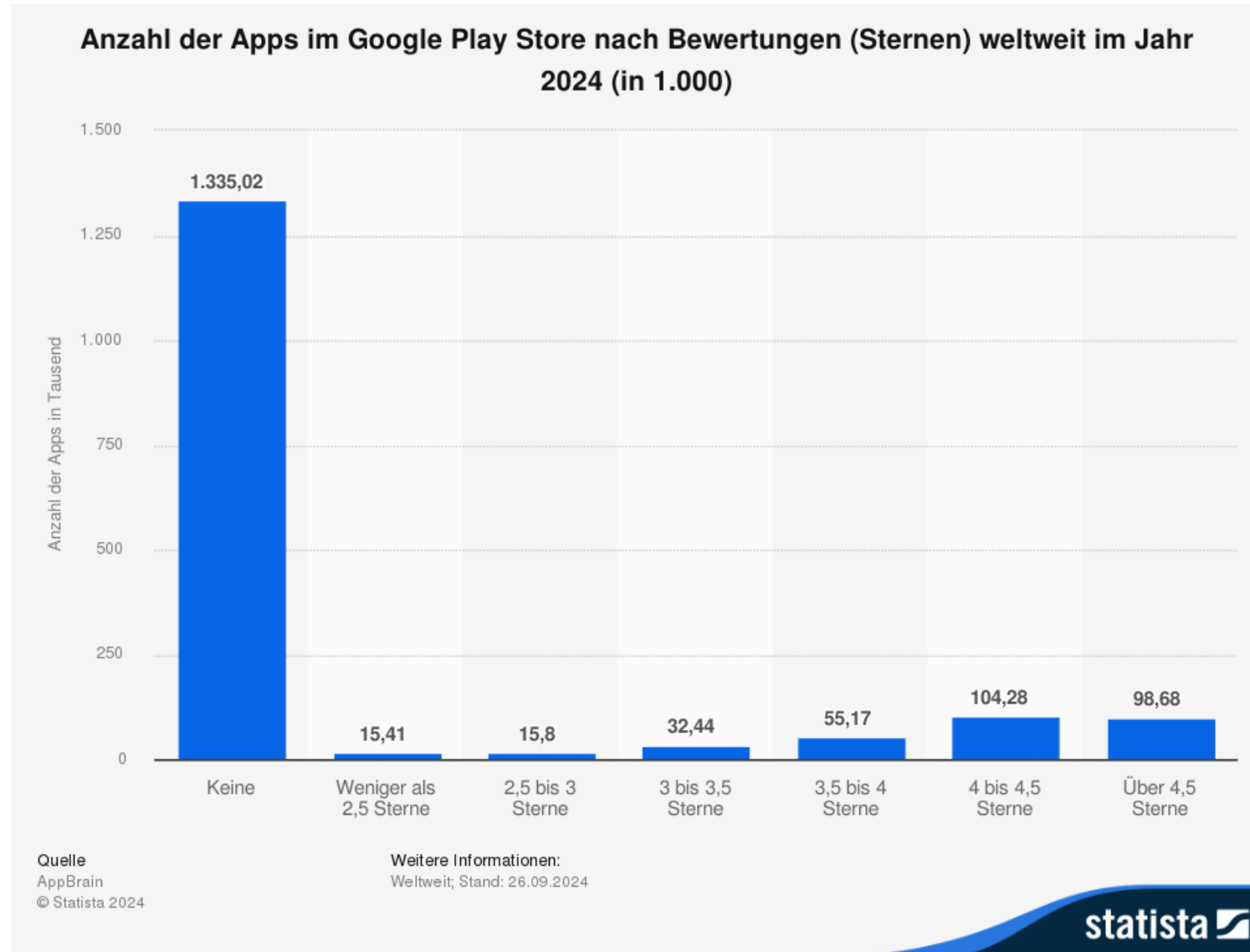
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT

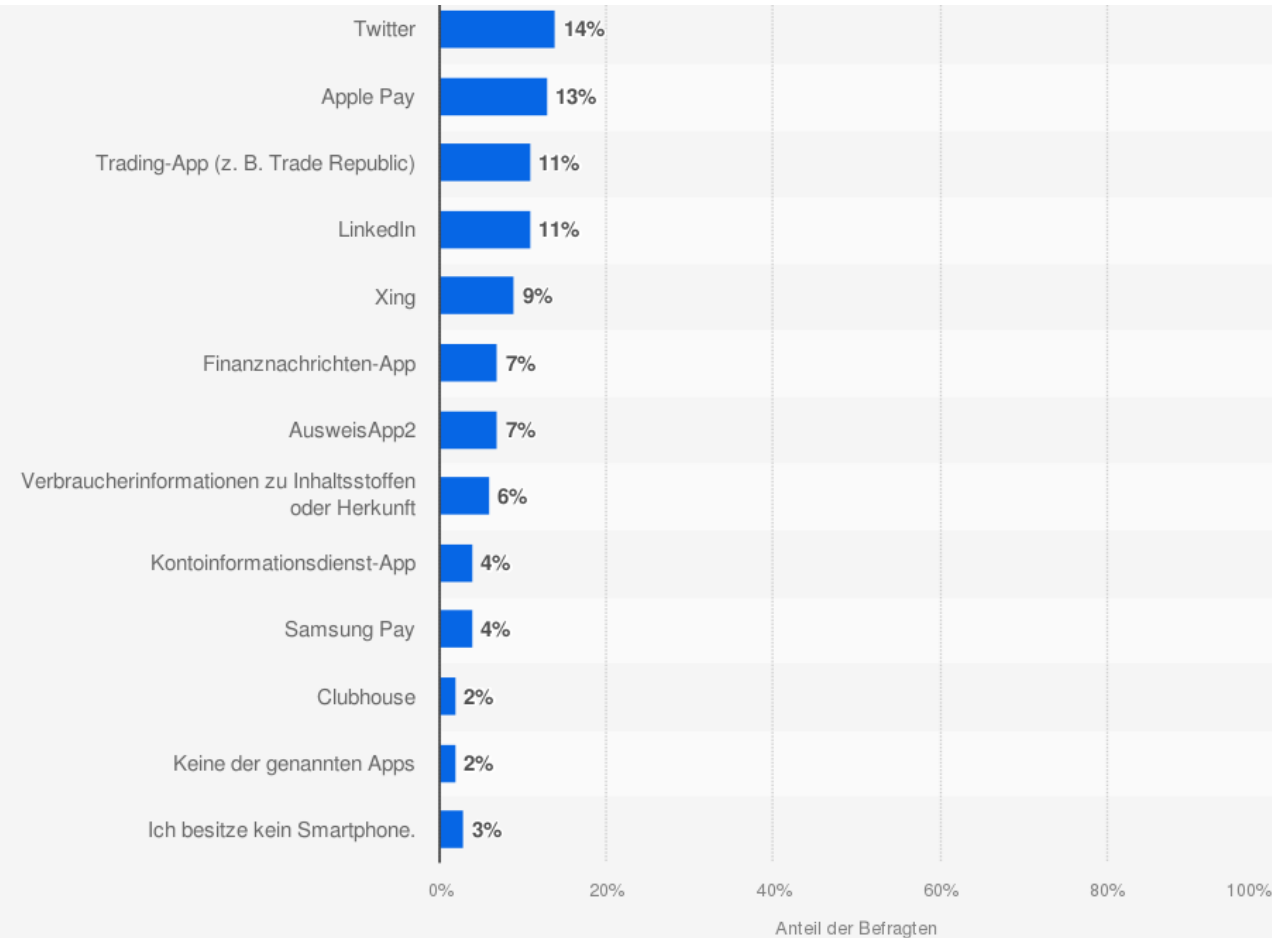
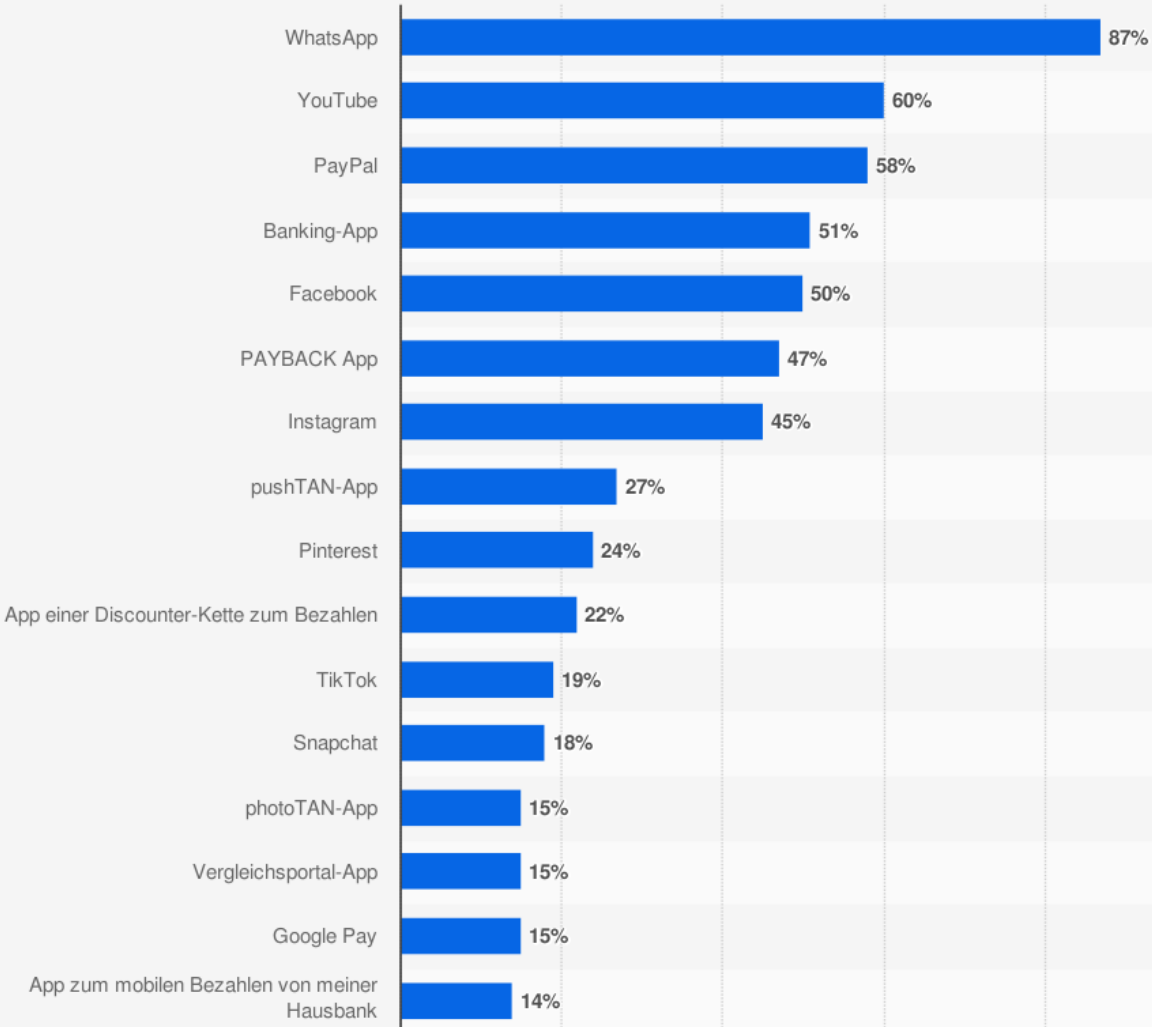


BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT

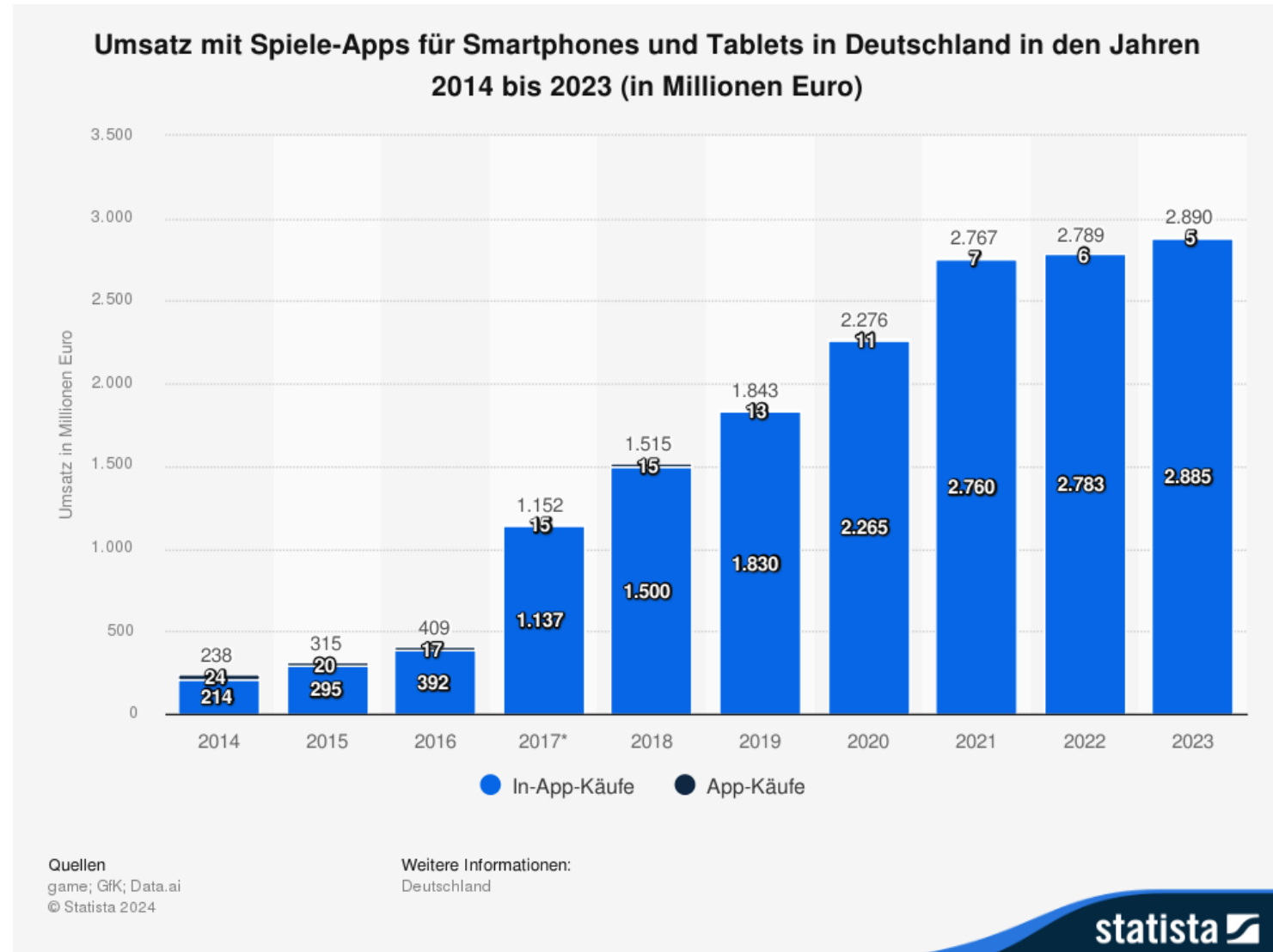
Welche Apps haben Sie auf Ihrem Smartphone installiert?



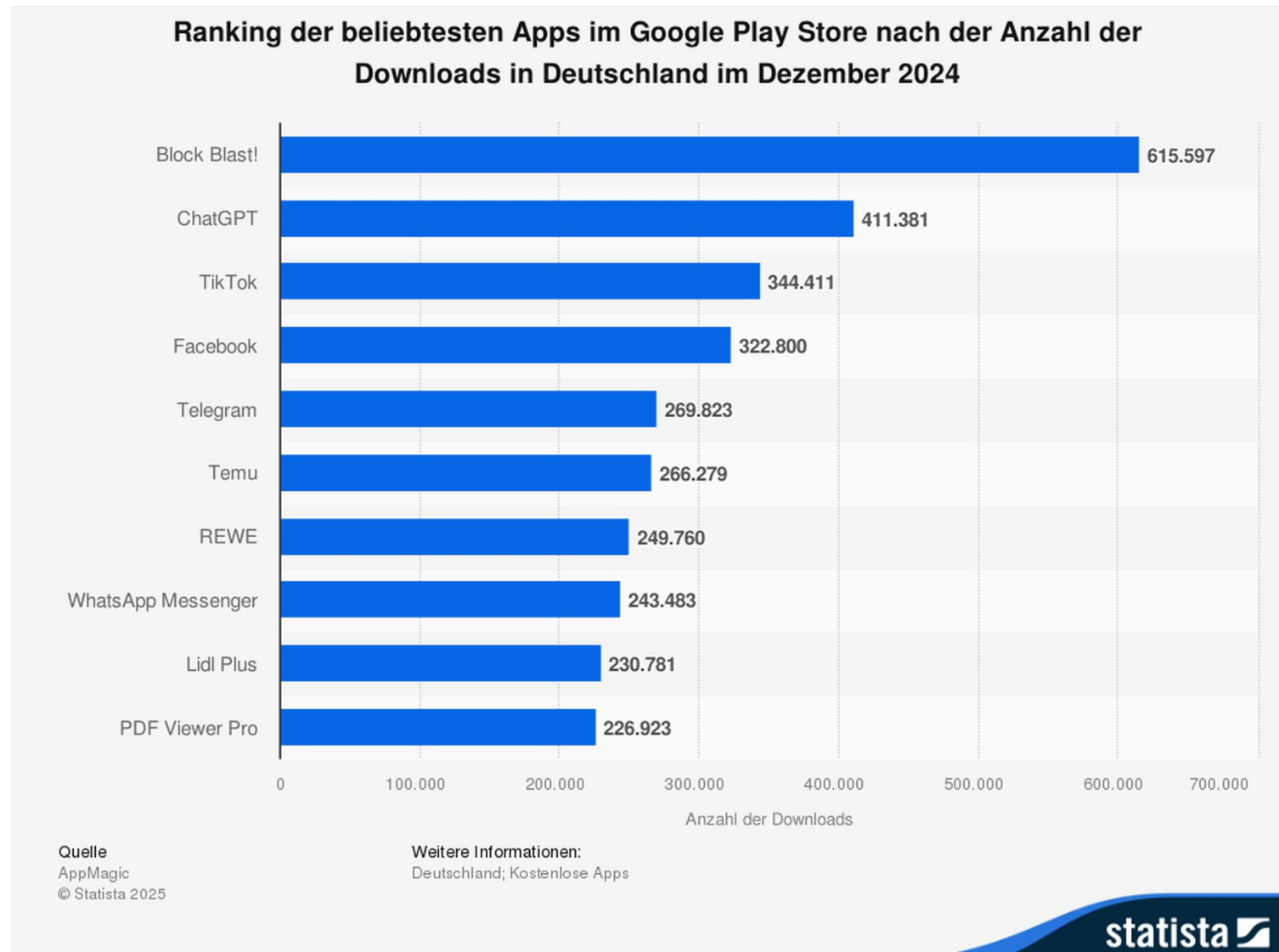
Quelle:
ibi research
© Statista 2024

Weitere Informationen:
Deutschland; Mai 2022; 1.015 Befragte; ab 16 Jahre; Computergestützte
Webinterviews (CAWI)

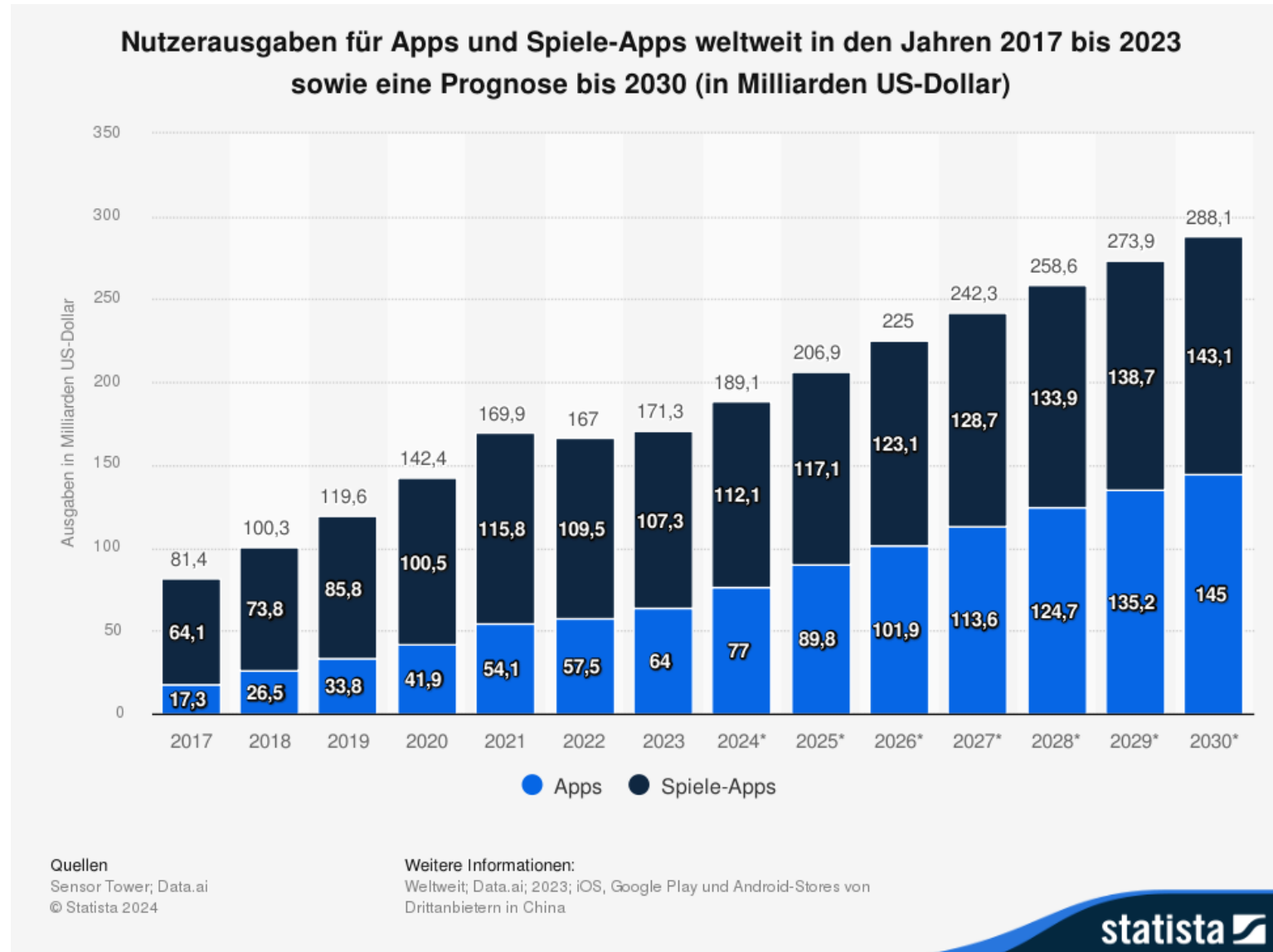
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER MARKT



BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER ALLGEMEINE RAHMEN

Apps als Erfolgstreiber

Merkmale von Apps:

- Zugriff auf Gerätespezifische Funktionen wie GPS und Kompass.
- Offline-Funktionalität durch lokale Datenspeicherung.

Vorteile von Apps gegenüber mobilen Webseiten:

- Voll funktionsfähig auch ohne Internetzugang (z. B. in Funklöchern).
- Höhere Usability durch optimierte, spezialisierte Funktionalitäten.

BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER ALLGEMEINE RAHMEN

Herausforderungen und Einschränkungen mobiler Geräte

Ressourcenbegrenzung:

- Geringere CPU-Leistung und kleinere Displays erfordern fokussierte Entwicklung.
- Einschränkungen durch begrenzten Akku und potenziellen Offline-Betrieb.

Sicherheitsaspekte:

- Höheres Risiko von Geräteverlust oder Diebstahl.
- Empfehlung: Sicherungsrelevante Daten auf Servern speichern, nicht lokal.

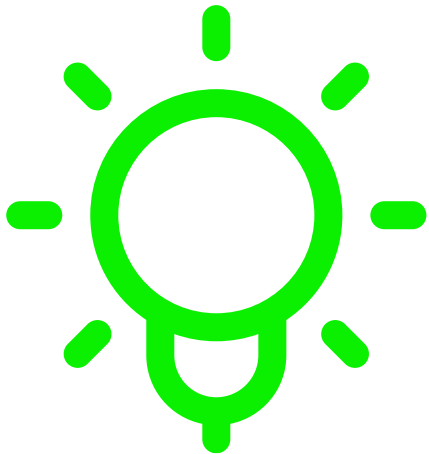
BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER ALLGEMEINE RAHMEN

Besonderheiten mobiler Technologien

Zusatzfunktionen mobiler Geräte:

- **GPS:** Positionsbestimmung für Location-Based Services (LBS).
- **Kompass:** Richtungsbestimmung zur Orientierung.
- **Gyroskop:** Erfassung von Rotations- und Neigungsdaten (wichtig für Spiele).
- **NFC:** Kontaktlose Datenübertragung im Nahbereich (Bezahlterminals, Ticketing, Zugangssysteme).
- **Weitere wie Beschleunigungssensoren oder Barometer**

WIE KÖNNEN SENSOREN FÜR DIE MOBILE APP-ENTWICKLUNG GENUTZT WERDEN?



- A. Bildet Gruppen á 2 Personen (auch die letzte Reihe – auch ihr drei)
- B. Recherchiert drei Apps, die verschiedenen **Sensoren** verwenden, um Funktionalitäten bereitzustellen. Beschreibe welche Sensoren besonders im Vordergrund stehen und wie diese die Funktionalität erweitern.
- C. Erstelle eine Kurzpräsentation für den Mitschrieb

BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER ALLGEMEINE RAHMEN

Best Practices bei der Entwicklung mobiler Apps

Ressourcenschonung:

- Minimierung des Stromverbrauchs durch optimierte Prozesse.
- Vorbereitung auf das außerplanmäßige Beenden von Apps.

Sicherheitsmaßnahmen:

- Vermeidung lokaler Speicherung sensibler Daten.
- Zusätzliche Sicherheitskonzepte aufgrund erhöhter Verlustgefahr.

BESONDERHEITEN VON MOBILEN ENDGERÄTEN | DER ALLGEMEINE RAHMEN

Sicherheit

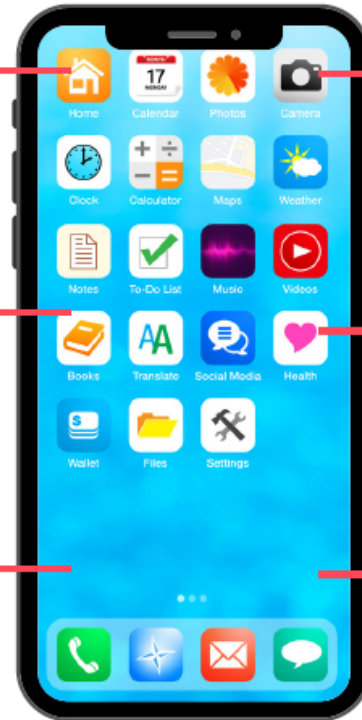
Die Gefahr von Verlust bzw. Diebstahl des Gerätes und damit des unbefugten Zutritts muss berücksichtigt werden.

Multifunktionalität

Mobile Endgeräte kombinieren verschiedene Funktionen in einem Gerät – Telefonieren, Surfen, Messaging, Musik und Videos abspielen, Navigation, Spiele etc.

Hardware

Mobile Endgeräte haben andere Voraussetzungen, was Batterielaufzeit, Netzwerkzugang und Bedienung (Touchscreen) betrifft.



Apps

Das sind auf die Besonderheiten von mobilen Endgeräten ausgerichtete Anwendungen.

Ökosystem

Unterschiedliche Betriebssysteme erfordern spezifische Implementierungen oder Versionen

Sensoren

GPS, Kompass, Gyroskop oder NFC sind erweiterte Eigenschaften von mobilen Endgeräten die genutzt werden können.

WAS HAT EINFLUSS AUF DIE MOBILE APP-ENTWICKLUNG?

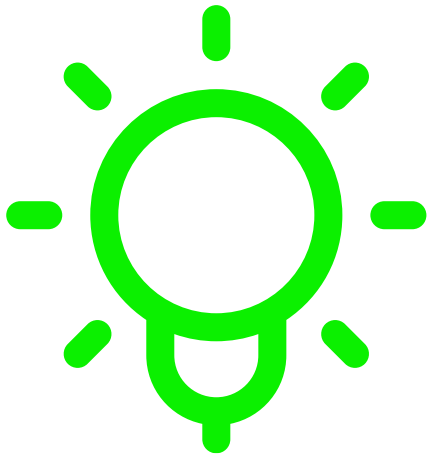
Leichten Einfluss:

Verbesserte Hardware, Leistungsveränderungen im OS, Aufbau der digitalen Tastatur

Starken Einfluss:

App-Store und dortige Regeln, UI/UX, „neues“ OS, dynamische Navigation/Tastaturersatz → Vollständige Steuerung via Touch, „Multitouch“

WAS HAT EINFLUSS AUF DIE MOBILE APP-ENTWICKLUNG?



- A. Bildet Gruppen á 2 Personen (auch die letzte Reihe – auch ihr drei)
- B. Definiert aus eurer Sicht die stärksten Einflussfaktoren für die Entwicklung von Apps
- C. Erarbeitet in eure Gruppen für die folgenden Kundengruppen (bei Unklarheiten die Begriffe eigenständig recherchieren und definieren) drei besonderen Anforderungen in der Entwicklung: Silver Surfer, Generation X, Generation Z, Generation Alpha
- D. Erstelle eine Kurz-Powerpoint mit dem Mitschrieb

Bis: 15:50

01

GRUNDLAGEN DER MOBILEN SOFTWARE-ENTWICKLUNG

BESONDERHEITEN DER MOBILEN SOFTWARE-ENTWICKLUNG

Entwicklungsprozess für stationäre Anwendungen (PCs, Server)

1. Anforderungsanalyse:

- Definition von Funktionen, Zielplattformen und nicht-funktionalen Anforderungen (z. B. Performance).

2. Implementierung:

- Auswahl von Programmiersprachen und Entwicklungsumgebungen.
- Umsetzung der Anforderungen in Sourcecode.

3. Test:

- Erstellung von Testfällen und -daten.
- Durchführung von Tests in einer möglichst realistischen Testumgebung.

4. Deployment:

- Installation und Konfiguration der Software in der produktiven Umgebung.
- Einmalige Installation auf dem Server, kein Aufwand für Endnutzer.

Unterschiede und Herausforderungen bei mobilen Anwendungen

1. Zielplattform:

- Unterschiedliche Gerätearten (Smartphones, Tablets) mit verschiedenen Hardware- und Softwarekonfigurationen.
- Vielzahl an Displaygrößen und -auflösungen.
- Kontinuierliche Weiterentwicklung des Android-Betriebssystems führt zu hoher Fragmentierung.

2. Programmiersprachen:

- Hoffnung, bestehende Java-Kenntnisse nutzen zu können.
- Potenziell neue Frameworks oder Tools erforderlich.

Unterschiede und Herausforderungen bei mobilen Anwendungen

3. Test:

Erhöhter Testaufwand durch:

- Unterschiedliche Endgeräte mit vielfältigen Konfigurationen.
- Fragmentierung der Android-Versionen.
- Unterstützung durch Emulatoren für Entwicklertests.

4. Deployment:

- Jede App muss individuell auf den Endgeräten der Nutzer installiert werden.
- Unterschiedliche Gerätekonfigurationen und Betriebssystemversionen erschweren den Prozess.
- Unterstützung durch App-Stores (z. B. Google Play) für Verteilung und Updates.

Zusätzliche Erkenntnisse und To Dos

Fragmentierung als Herausforderung:

- Klärung, welche Endgeräte und Plattformversionen unterstützt werden müssen.
- Berücksichtigung der Vielfalt bei Entwicklung und Tests.

Neue Tools und Techniken:

- Erkundung von Entwicklungsumgebungen, Emulatoren und Deployment-Tools.
- Nutzung von App-Stores zur Vereinfachung der App-Verteilung.

01

GRUNDLAGEN DER MOBILEN SOFTWARE-ENTWICKLUNG

EINTEILUNG VON MOBILEN ENDGERÄTEN

Hardwareseitig lässt sich der mobile Markt in die folgenden Bereiche aufteilen:

- Smartphones,
- Tablets sowie
- Embedded-Systeme.

Die Lücke zwischen Smartphones und Laptops schließen Tablets. Je nach Betrachtungsweise kann man sie als „große Smartphones ohne Telefonfunktion“ oder „kleine Laptops mit Touchscreen“ bezeichnen.

Neben dieser hardwareseitigen Unterscheidung lässt sich der mobile Endgerätemarkt auch auf der Basis des auf den Endgeräten eingesetzten **Betriebssystems** unterscheiden. Nachdem andere Anbieter wie Microsoft und Blackberry die Entwicklung eigener mobiler Betriebssysteme aufgegeben haben, lassen sich hier im Grunde nur noch Android und iOS aufzählen.

Hardwareseitig lässt sich der mobile Markt in die folgenden Bereiche aufteilen:

- Smartphones,
- Tablets sowie
- Embedded-Systeme.

Die Lücke zwischen Smartphones und Laptops schließen Tablets. Je nach Betrachtungsweise kann man sie als „große Smartphones ohne Telefonfunktion“ oder „kleine Laptops mit Touchscreen“ bezeichnen.

Neben dieser hardwareseitigen Unterscheidung lässt sich der mobile Endgerätemarkt auch auf der Basis des auf den Endgeräten eingesetzten **Betriebssystems** unterscheiden. Nachdem andere Anbieter wie Microsoft und Blackberry die Entwicklung eigener mobiler Betriebssysteme aufgegeben haben, lassen sich hier im Grunde nur noch Android und iOS aufzählen.

iOS:

- Herstellerbindung: Entwickelt von Apple, exklusiv für Apple-Geräte (iPhone, iPad).
- Marktverbreitung: Eingeschränkte Hardwarebasis, aber hoher Marktanteil.
- Testaufwände: Reduziert, da nur wenige Endgeräte unterstützt werden.
- Systemarchitektur: Geschlossenes System, Kontrolle durch Apple (Weiterentwicklung, App-Zulassung).
- Entwicklung:
 - Programmiersprache: Swift.
 - Entwicklungsumgebung: Xcode, optimiert für macOS.

Android:

- Offenes System: Open Source, Weiterentwicklung durch die Open Handset Alliance (Konsortium von ~90 Firmen).
- Marktverbreitung: Breite Hardwarebasis, dominant im Embedded-Markt.
- Entwicklung:
 - Programmiersprachen:
 - Java: Die ursprüngliche Programmiersprache für Android-Apps, weit verbreitet und gut dokumentiert.
 - Kotlin: Moderne, von Google bevorzugte Sprache für Android, mit verbesserter Syntax und höherer Effizienz.
 - Basis:
 - Linux: Das Betriebssystem, auf dem Android aufbaut, sorgt für Stabilität und Sicherheit.
 - XML: Wird für die Gestaltung der Benutzeroberfläche (UI) und die Definition von Layouts verwendet.

Android:

- Offenes System: Open Source, Weiterentwicklung durch die Open Handset Alliance (Konsortium von ~90 Firmen).
- Marktverbreitung: Breite Hardwarebasis, dominant im Embedded-Markt.
- Entwicklung:
 - Programmiersprachen:
 - Java: Die ursprüngliche Programmiersprache für Android-Apps, weit verbreitet und gut dokumentiert.
 - Kotlin: Moderne, von Google bevorzugte Sprache für Android, mit verbesserter Syntax und höherer Effizienz.
 - Basis:
 - Linux: Das Betriebssystem, auf dem Android aufbaut, sorgt für Stabilität und Sicherheit.
 - XML: Wird für die Gestaltung der Benutzeroberfläche (UI) und die Definition von Layouts verwendet.

Zielplattformen und Herausforderungen:

Hardware- und Software-Abhängigkeit:

- Zusammenspiel beider Komponenten essenziell für funktionsfähige Anwendungen.
- Nicht alle Softwareplattformen sind mit allen Hardwaretypen kompatibel.

Getrennte Entwicklung für Plattformen:

- Separate Apps für Android und iOS erforderlich aufgrund unterschiedlicher Systemarchitekturen.
- Erhöhter Entwicklungsaufwand, da keine Plattform vernachlässigt werden darf.

Lösungsansätze für plattformübergreifende Entwicklung:

Cross-Plattform-Entwicklungsumgebungen:

- Beispiele: Flutter, React Native.
- Ziel: Ein Sourcecode für mehrere Plattformen.

Mobile Webseiten:

- Plattformunabhängig, lauffähig in Webbrowsern.
- Alternative Architektur zu nativen Apps.

Marktstudien und Zielgruppen:

Marktsegmentierung:

- Unterteilung nach Hardware (z. B. Smartphone, Tablet, Embedded) oder Software (z. B. Android, iOS).
- Zielgruppenabhängige Ausrichtung der Marktstudien.

Ständige Weiterentwicklung:

- Zielgruppenausrichtungen und Marktanteile unterliegen kontinuierlichem Wandel.

01

GRUNDLAGEN DER MOBILEN SOFTWARE-ENTWICKLUNG

DIE ANDROID-PLATTFORM

DIE ANDROID-PLATTFORM



Kurz: Entwickelt von Google, Veröffentlichung der ersten Version 2008 unter der Open Handset Alliance (Konsortium aus Netzbetreibern, Software- und Hardwarefirmen).

Betriebssystem:

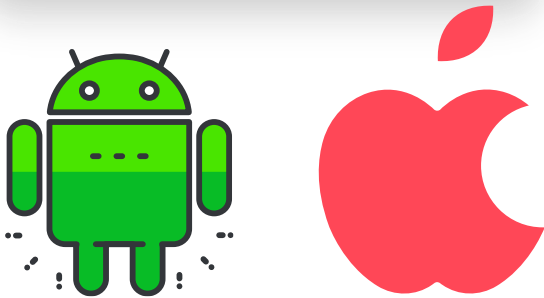
- Android basiert auf einer angepassten Linux-Distribution.
- Ressourcenoptimierung durch Entfernen nicht benötigter Linux-Komponenten.

Plattform: Umfasst neben dem Betriebssystem Tools für Erstellung, Wartung und Deployment von Apps.

— Hier weiter

Native

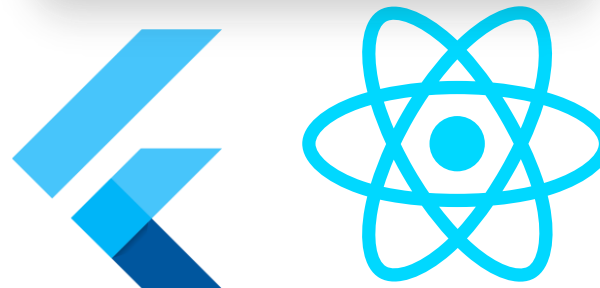
Entwicklung



Native App Entwicklung bezeichnet den Prozess der Erstellung von mobilen Anwendungen, die speziell für **ein bestimmtes Betriebssystem** (z. B. Android oder iOS) entwickelt werden

Crossplattform

Entwicklung



Cross-Plattform-Entwicklungsumgebungen erstellen Apps, bei denen aus einem Sourcecode-Stand lauffähige Apps für **unterschiedliche Plattformen** generiert werden können.

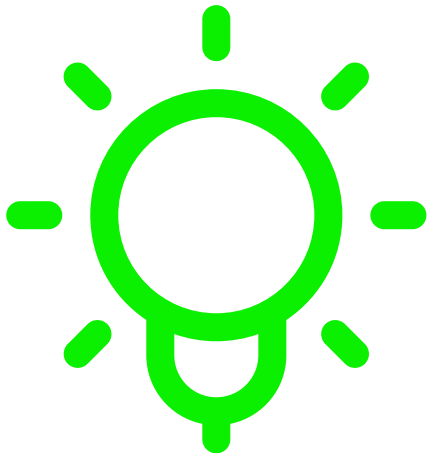
Bundler für

Webapps



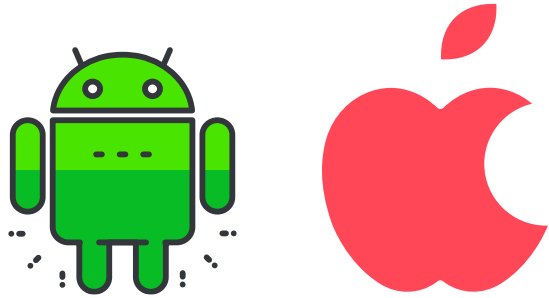
Frameworks wie Ionic ermöglichen es, diese Webanwendungen in **native App-Container** zu verpacken, sodass sie wie native Apps auf den mobilen Plattformen (Android, iOS) laufen können.

WIE SOLLTE MAN ENTWICKELN?



- A. Bildet 2 Gruppen
- B. Erarbeitete jeweils:
 - 1. Recherchiere Vor- und Nachteile die durch die Entscheidung zur Entwicklung einer nativen App entstehen könnte. Überlege verschiedene Szenarien, bei der eine native Entwicklung Sinn machen kann.
 - 2. Recherchiere Vor- und Nachteile die durch die Entscheidung zur Entwicklung einer crossplattform App entstehen könnte. Überlege verschiedene Szenarien, bei der eine native Entwicklung Sinn machen kann.
- C. Erstelle eine Kurz-Powerpoint mit den Ergebnissen

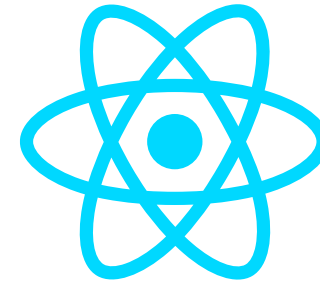
WIE SOLLTE MAN ENTWICKELN?



- Höchste Performance, da direkt auf die nativen APIs und Ressourcen der Geräte zugegriffen wird.
- Optimale Benutzererfahrung, da die App auf die Plattform zugeschnitten ist.
- Voller Zugriff auf Hardware- und Software-Funktionen (z. B. Kamera, GPS).
- Getrennte Codebasen für jede Plattform (höherer Entwicklungsaufwand).
- Längere Entwicklungszeit, wenn beide Plattformen abgedeckt werden sollen.



➡ Höhere Entwicklungskosten, aber maximale Effizienz und Performance für jede Plattform.



- Schnellere Entwicklung, da eine gemeinsame Codebasis für mehrere Plattformen verwendet wird.
- Geringere Entwicklungskosten im Vergleich zu nativen Apps.
- Zugriff auf viele native Funktionen durch Plugins oder Bridges.



- Performance kann in bestimmten Bereichen (z. B. Animationen, rechenintensive Prozesse) etwas niedriger sein als bei nativen Apps.
- Abhängigkeit von Frameworks für nativen Zugriff, was zu Wartungsaufwand führen kann.



Ideal für Projekte mit knappen Budgets und kurzen Entwicklungszyklen, jedoch mit leichten Kompromissen bei Performance und nativer Integration.

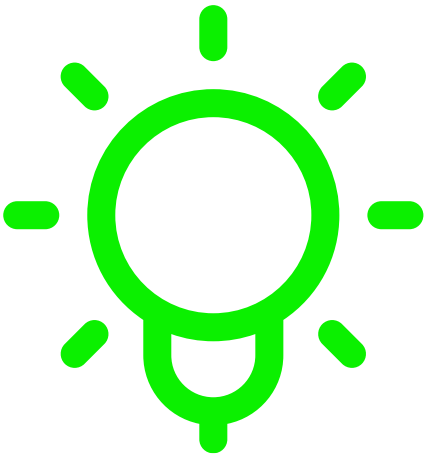
Entwicklung:

Programmiersprachen:

- Hauptsächlich Java, aber Kotlin hat zunehmend an Bedeutung gewonnen, da es eine prägnantere Syntax und erweiterte Sicherheitsfunktionen bietet.
- Aktuelle Trends: Verwendung von Jetpack Compose für deklarative Benutzeroberflächen.

Virtuelle Maschine (VM):

- Apps laufen in einer VM, die Hardware virtuell nachbildet.
- **Java-Bytecode wird während der Installation in maschinenabhängigen Binärcode umgewandelt.**



- A. Bildet Gruppen
- B. Vergleiche **Android Automotive OS** mit **VW.OS**, dem Betriebssystem, das Volkswagen für seine Fahrzeuge entwickelt. Recherchiere, wie Volkswagen mit VW.OS ein eigenes Fahrzeugbetriebssystem entwickelt und welche Ziele damit verfolgt werden.

Vergleiche VW.OS mit Android Automotive OS und fokussiere dich dabei auf folgende Aspekte:

- Vorteile für VW und die Nutzer: Was will Volkswagen mit einem eigenen Betriebssystem erreichen, und welche Vorteile bietet Android Automotive OS im Vergleich?
- Marktperspektive: Wie könnte sich VW.OS in der Automobilbranche entwickeln, und welche Herausforderungen könnten im Vergleich zur etablierten Android-Plattform auftreten?
- Technologie und Architektur: Welche Unterschiede bestehen in der Struktur und den verwendeten Technologien?

- Volkswagen entwickelt mit VW.OS ein eigenes Betriebssystem, um die Kontrolle über die Fahrzeugsoftware zu behalten und seine E-Autos unabhängig von Drittanbietern zu gestalten. Der Vergleich von VW.OS mit Android Automotive OS zeigt deutliche Unterschiede in Bezug auf Technologie, Vorteile und langfristige Perspektiven.
- **Technologie und Architektur:** VW.OS wird speziell für die Fahrzeuge von Volkswagen entwickelt und ist auf die Integration in die gesamte Fahrzeugarchitektur abgestimmt. Android Automotive OS hingegen ist ein plattformübergreifendes Betriebssystem.
- **Vorteile für VW und die Nutzer:** Ein zentrales Argument für VW.OS ist der Schutz der Nutzerdaten. Wenn Volkswagen Android Automotive OS verwenden würde, hätte Google Zugriff auf eine Vielzahl von Daten, die im Auto gesammelt werden – etwa Fahrverhalten, Standorte und Unterhaltungspräferenzen.
- **Marktperspektive:** Die Bedeutung von Software in modernen E-Autos hat im Vergleich zu Verbrennern vor 10 Jahren enorm zugenommen. E-Fahrzeuge sind stark auf Software angewiesen, um effizient zu funktionieren – sei es bei der Steuerung der Batterie, der Implementierung von Assistenzsystemen oder der Kommunikation mit anderen Systemen.

02

ANDROID-SYSTEMARCHITEKTUR

WORUM GEHT ES IN DIESEM KAPITEL?

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- aus welchen Bestandteilen ein Android-System besteht.
- welche dieser Komponenten für die Software-Entwicklung von Bedeutung sind.
- welche Netzwerk- und Kommunikationstechnologien für die mobile Android-Welt relevant sind.

02

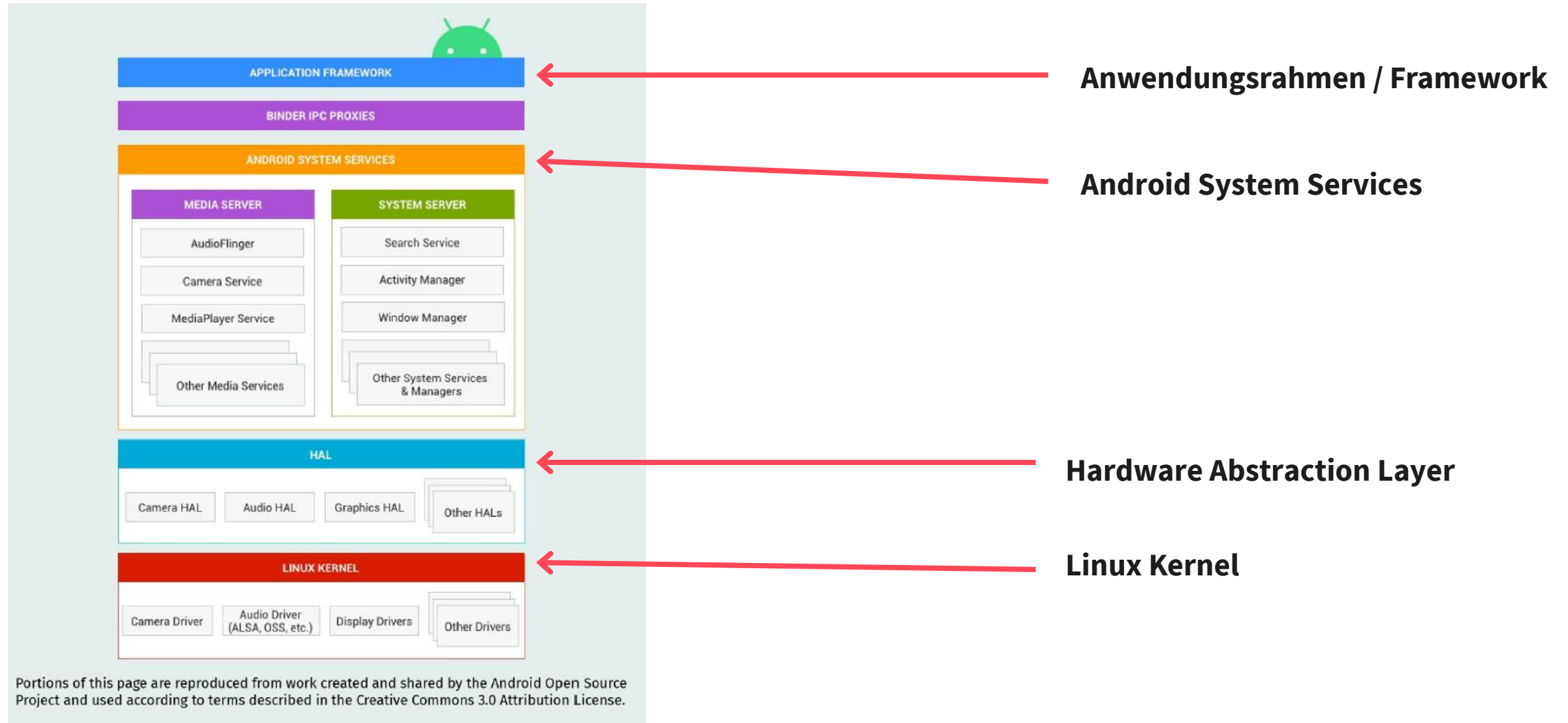
ANDROID-SYSTEMARCHITEKTUR

DAS ANDROID-SYSTEM

DAS ANDROID-SYSTEM

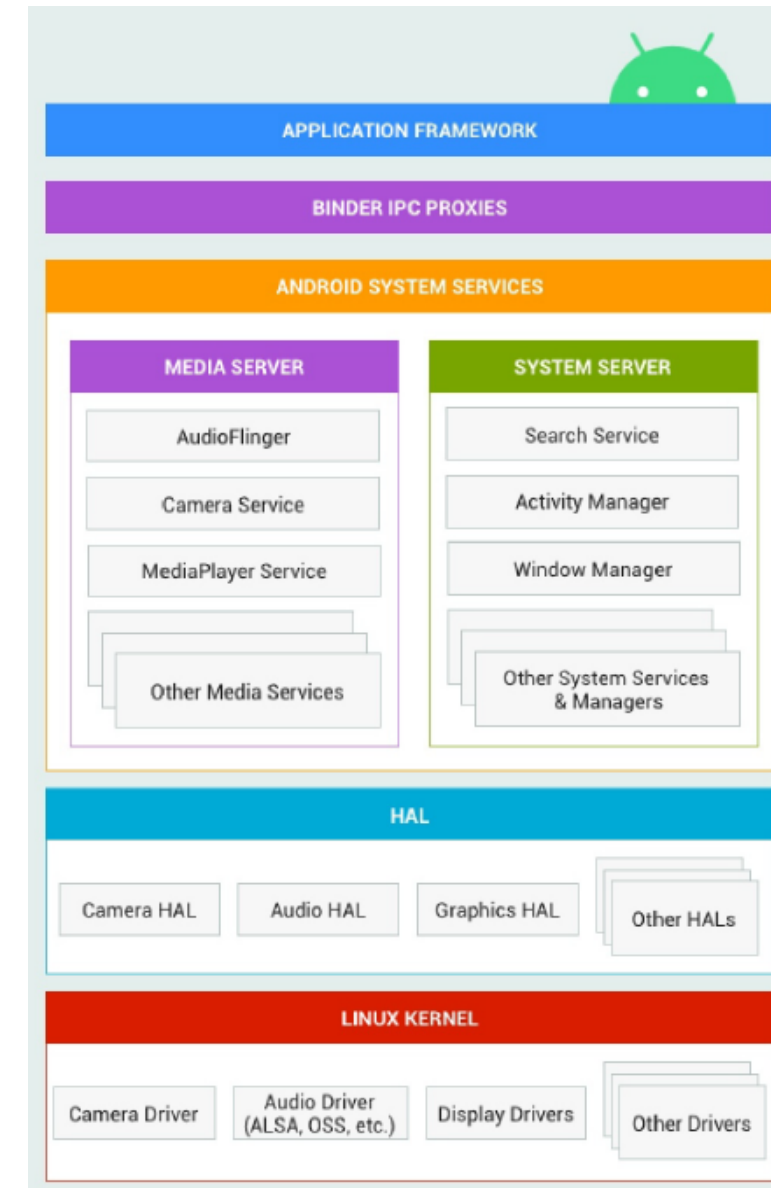


DAS ANDROID-SYSTEM



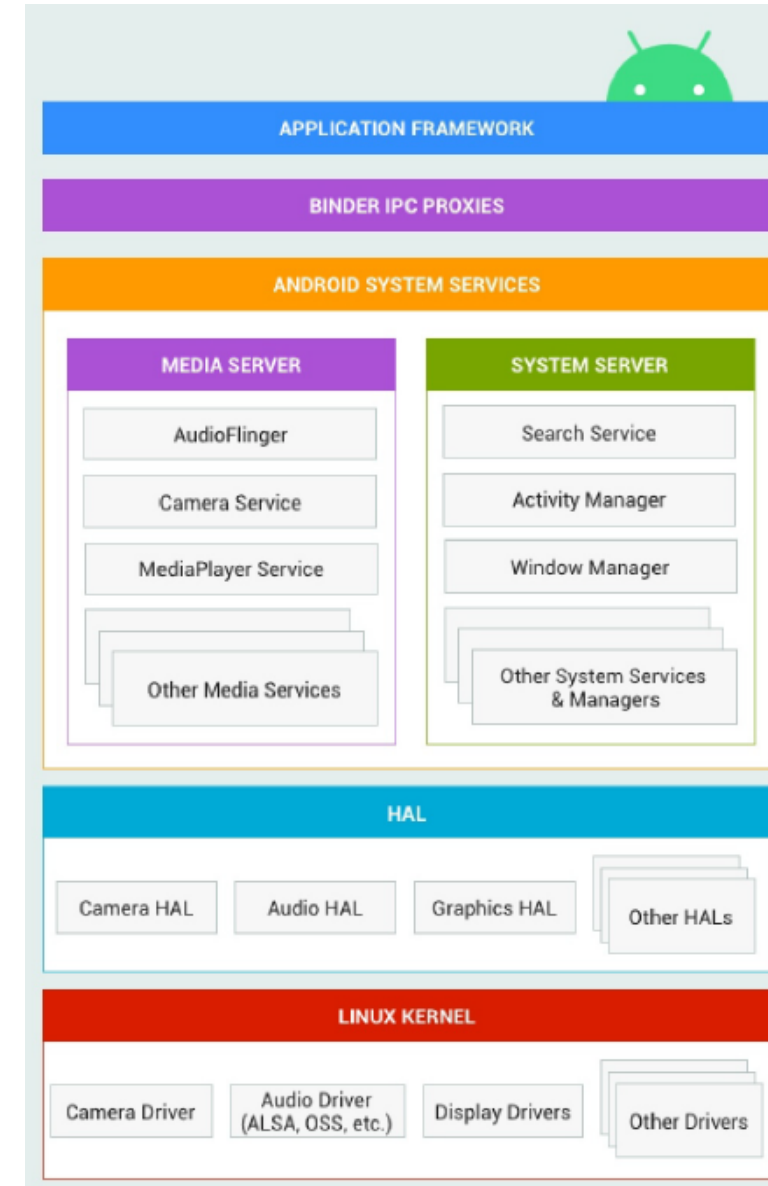
Application Framework (Anwendungsrahmen)

- **Funktion:** Stellt Entwicklern die grundlegenden APIs zur Verfügung, um Anwendungen zu erstellen.
- **Komponenten:**
 - Tools zur Verwaltung von Benutzeroberflächen, Ressourcen und App-Komponenten wie Activities, Services und Content-Providern.
 - Ermöglicht einfachen Zugriff auf Android-Dienste wie Standort, Benachrichtigungen und Sensoren.



Binder IPC Proxies

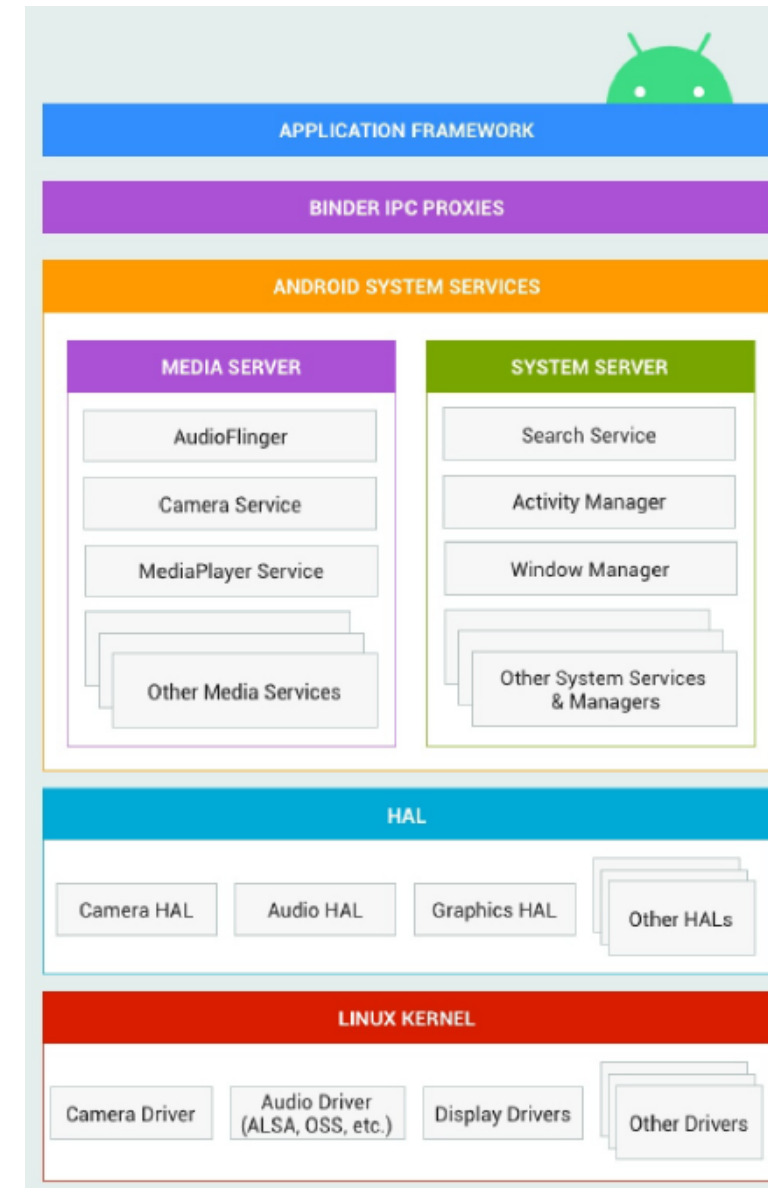
- **Funktion:** Ermöglicht die Kommunikation zwischen verschiedenen Prozessen (Inter-Process Communication, IPC).
- **Besonderheiten:**
 - Vermittelt Daten und Befehle sicher zwischen Anwendungen und Systemdiensten.
 - Stellt eine wichtige Grundlage für die Zusammenarbeit der einzelnen Android-Komponenten dar.



DAS ANDROID-SYSTEM

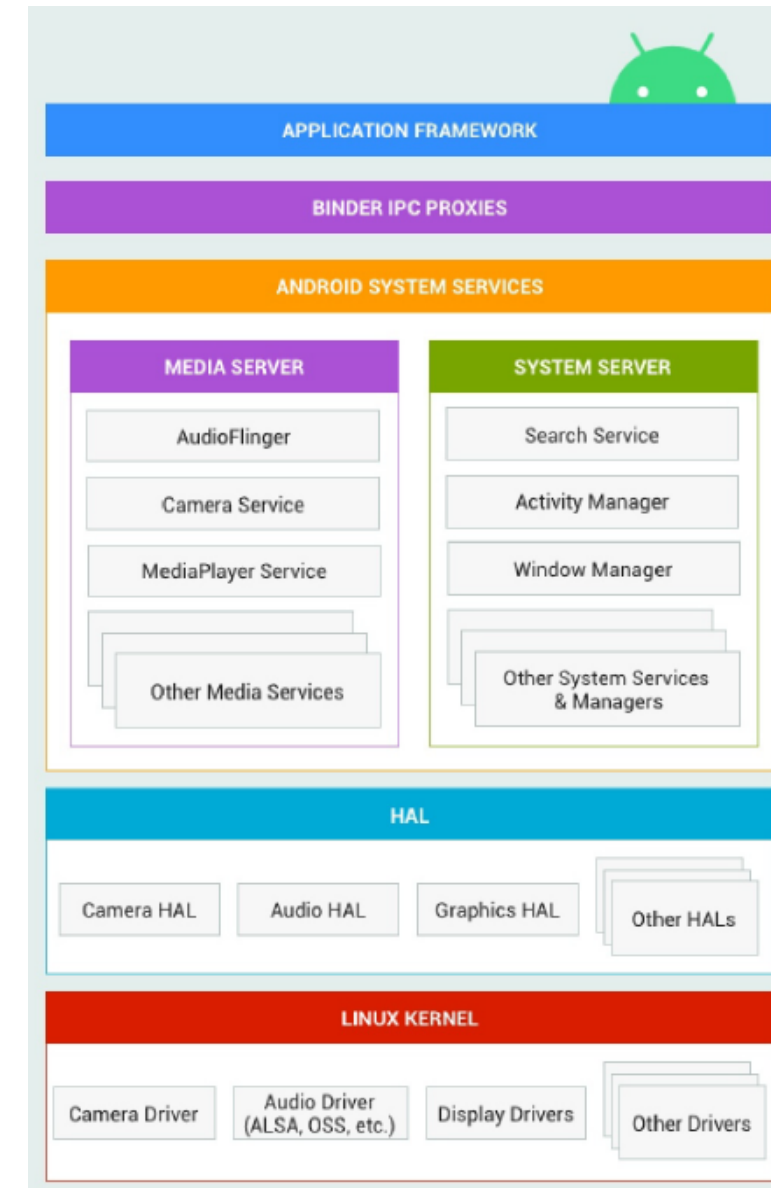
Android System Services

- **Aufteilung:**
 - **Media Server:**
 - Verarbeitet alle medienbezogenen Dienste wie Audio, Kamera und Video.
 - Beinhaltet Dienste wie **AudioFlinger**, **Camera Service** und **MediaPlayer Service**.
 - **System Server:**
 - Zuständig für die Verwaltung systembezogener Aufgaben.
 - Dienste umfassen **Search Service**, **Activity Manager**, **Window Manager** und andere System-Manager.



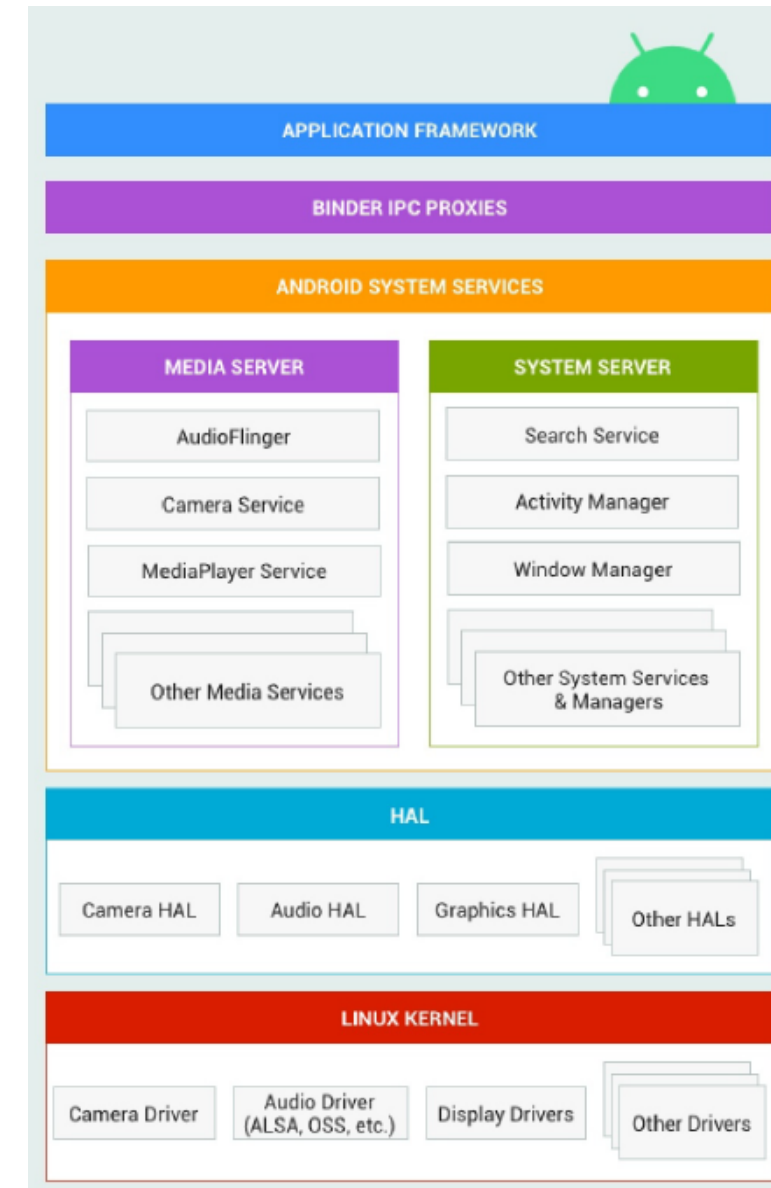
Android System Services - Media Server:

- **AudioFlinger:**
 - Verarbeitet und mischt Audio-Streams, bevor sie an die Hardware weitergegeben werden.
 - Zuständig für die Audioausgabe aller Anwendungen und Systemtöne.
- **Camera Service:**
 - Schnittstelle für den Zugriff auf die Kamera-Hardware.
 - Ermöglicht Funktionen wie Fotos aufnehmen, Videoaufzeichnung und Zugriff auf Kameraeinstellungen.
- **MediaPlayer Service:**
 - Bietet Funktionen zur Wiedergabe von Multimedia-Inhalten (Audio und Video).
 - Unterstützt verschiedene Formate und sorgt für die nahtlose Integration in Android-Apps.
- **Weitere Dienste:**
 - Andere Media Services wie Audio- oder Video-Encoding und Streaming.
 - Erweitern die Funktionalität für spezielle Medienanwendungen.



Android System Services – System Server:

- **Search Service:**
 - Ermöglicht die zentrale Suche auf dem Gerät und im Internet.
 - Unterstützt Funktionen wie Sprachsuche und App-übergreifende Suchanfragen.
- **Activity Manager:**
 - Verwaltet die Lebenszyklen von Anwendungen und Activities.
 - Kontrolliert, welche App im Vordergrund ist und wie der Speicher auf Apps verteilt wird.
- **Window Manager:**
 - Zuständig für die Anordnung und Darstellung von App-Fenstern auf dem Bildschirm.
 - Kontrolliert Layout, Übergänge und Animationen.
- **Weitere System-Manager:**
 - Package Manager: Installiert und verwaltet Apps und deren Berechtigungen.
 - Battery Manager: Überwacht und optimiert den Energieverbrauch.



```
class MainActivity : AppCompatActivity() {

    // Eine Variable, die den Rückgabewert für die Kamera definiert
    val CAMERA_REQUEST_CODE = 100

    // Bildanzeige-Element in der UI, um das geschossene Bild anzuzeigen
    lateinit var imageView: ImageView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Button und ImageView aus dem Layout mit den entsprechenden IDs verbinde
        val cameraButton = findViewById<Button>(R.id.cameraButton)
        imageView = findViewById(R.id.imageView)

        // Wenn der Button geklickt wird, wird die Kamera geöffnet
        cameraButton.setOnClickListener {
            // Intent, um die Kamera-App zu starten
            val cameraIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

            // Überprüfen, ob es eine App gibt, die den Intent behandeln kann
            if (cameraIntent.resolveActivity(packageManager) != null) {
                // Die Kamera-App starten
                startActivityForResult(cameraIntent, CAMERA_REQUEST_CODE)
            }
        }

        // Diese Methode wird aufgerufen, wenn die Kamera ein Bild gemacht hat
        override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
            super.onActivityResult(requestCode, resultCode, data)

            // Überprüfen, ob das Bild erfolgreich aufgenommen wurde
            if (requestCode == CAMERA_REQUEST_CODE && resultCode == Activity.RESULT_OK) {
                // Das Bild wird als Bitmap abgerufen
                val imageBitmap = data?.extras?.get("data") as Bitmap

                // Das Bild in der ImageView anzeigen
                imageView.setImageBitmap(imageBitmap)
            }
        }
    }
}
```



Die Berechtigung zur Verwendung der Kamera muss in der **AndroidManifest.xml** hinzugefügt werden.



Wir verwenden einen Intent, um die Kamera-App zu starten (**Intent(MediaStore.ACTION_IMAGE_CAPTURE)**)



Der Intent startet die Kamera und das Resultat (das Bild) wird in der Methode **onActivityResult** zurückgegeben.



Das Bild wird als Bitmap abgerufen und in einer ImageView angezeigt.

DAS ANDROID-SYSTEM - PRAXISBEISPIEL

Ziel des Berechtigungskonzepts: Schutz der Privatsphäre und Sicherheit des Benutzers - der Endnutzer entscheidet, welche App auf welche Daten und Dienste zugreifen darf. So wird der Missbrauch von sensiblen Daten verhindert.

Vordefinierte Berechtigungen: Android stellt eine Liste von vordefinierten Berechtigungen bereit. Diese können bei der Entwicklung einer App vom Entwickler angefordert werden. Zum Beispiel:

- Kontakte: Auslesen der gespeicherten Kontakte.
- GPS: Ermittlung der aktuellen Position des Geräts.



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

Zwei Arten von Berechtigungen: Normale Berechtigungen werden automatisch erteilt, da sie keine Risiken für den Benutzer darstellen. Gefährliche Berechtigungen müssen explizit vom Benutzer genehmigt werden, da sie potenziell sensible Informationen betreffen.

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

    <!-- Definiert die minimale und die Ziel-Android-Version -->
    <uses-sdk
        android:minSdkVersion="21"
        android:targetSdkVersion="33" />

    <!-- Berechtigung für den Zugriff auf das Internet -->
    <!-- Wird z.B. für Netzwerkverbindungen, APIs, etc. benötigt -->
    <uses-permission android:name="android.permission.INTERNET" />

    <!-- Berechtigung für den Zugriff auf den Standort des Benutzers (GPS) -->
    <!-- ACCESS_FINE_LOCATION gibt Zugriff auf den genauen Standort -->
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <!-- Berechtigung für den Zugriff auf den ungefähren Standort des Benutzers -->
    <!-- ACCESS_COARSE_LOCATION gibt Zugriff auf einen ungenauen Standort -->
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

    <!-- Berechtigung zum Auslesen der gespeicherten Kontakte -->
    <!-- Wird benötigt, wenn eine App auf die Kontaktliste zugreifen soll -->
    <uses-permission android:name="android.permission.READ_CONTACTS" />

    <!-- Berechtigung zum Versenden von SMS-Nachrichten -->
    <!-- Diese Berechtigung ist sehr sensibel und wird oft nur von speziellen Apps -->
    <uses-permission android:name="android.permission.SEND_SMS" />

    <!-- Berechtigung zum Speichern und Lesen von Dateien auf dem Gerät -->
    <!-- Wird benötigt, wenn die App Dateien auf dem Speicher des Geräts sichern möchte -->
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

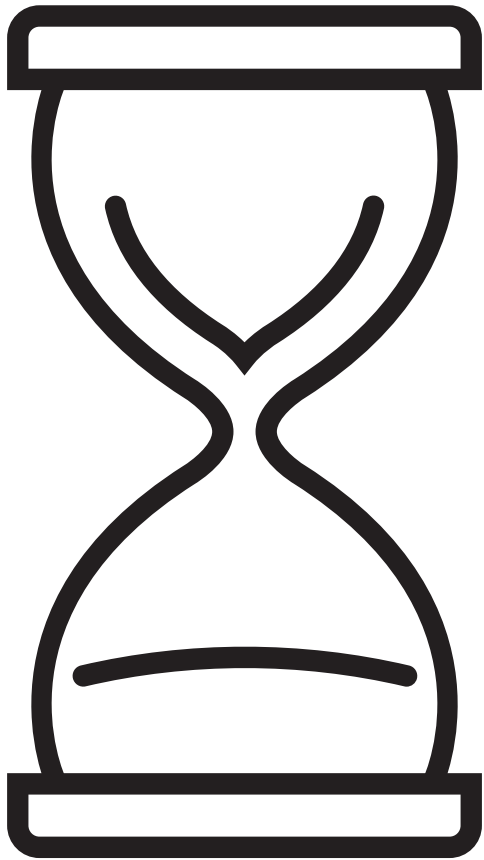
    <!-- Berechtigung zum Verwenden der Kamera -->
    <!-- Wird benötigt, um Fotos oder Videos aufzunehmen -->
    <uses-permission android:name="android.permission.CAMERA" />

</manifest>
```

Diese Berechtigung wird für jede App benötigt, die Daten über das Internet sendet oder empfängt, z.B. bei der Verwendung von APIs oder dem Herunterladen von Inhalten.

Diese Berechtigungen werden benötigt, wenn die App den Standort des Benutzers erfassen möchte. Der genaue Standort (z.B. GPS) erfordert ACCESS_FINE_LOCATION, während der ungenaue Standort (z.B. über WLAN oder Mobilfunkmasten) durch ACCESS_COARSE_LOCATION ermittelt werden kann.

Diese Berechtigung wird benötigt, um die Kamera des Geräts zu verwenden, z.B. für Foto-Apps oder Apps, die QR-Codes scannen.



Sandbox-Prinzip: Diese dient zur technischen Abschottung eines Prozesses gegen unbefugte Zugriffe von und nach außen

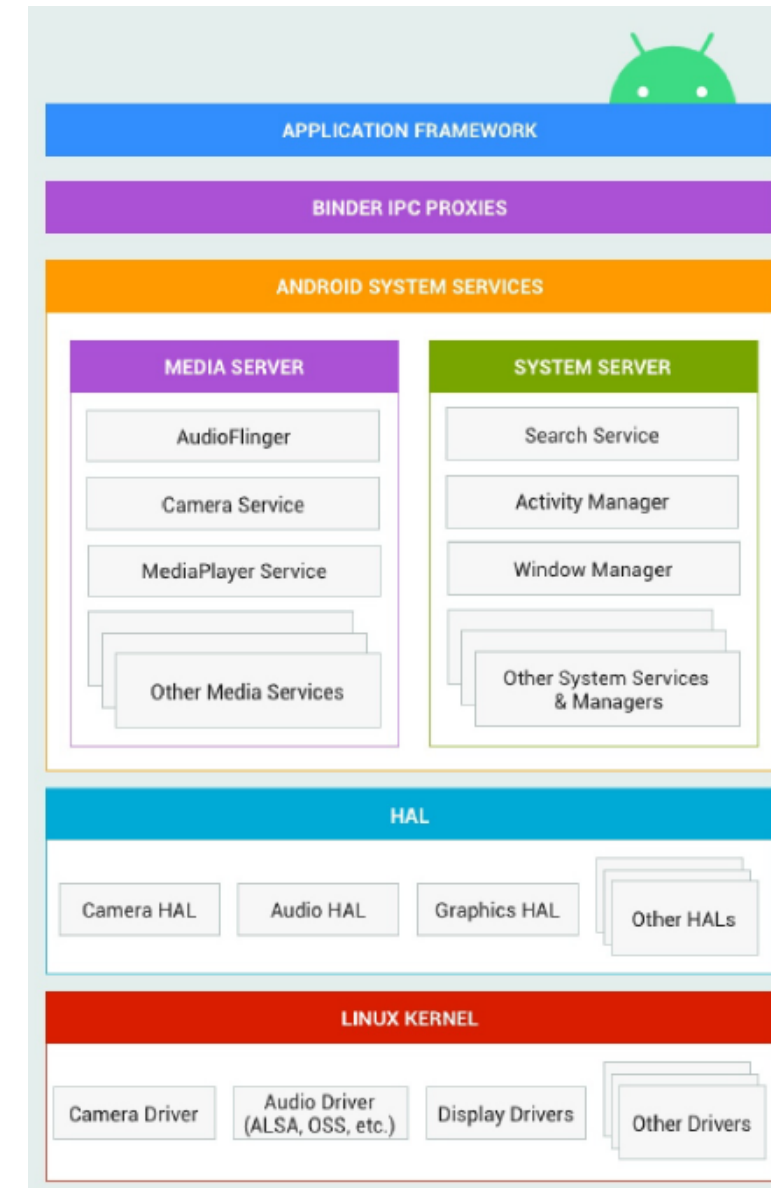
Abschottung der App in einer Sandbox: Jede App besitzt in einen eigenen Speicherbereich (sowohl im Hauptspeicher als auch im Dateisystemspeicher), auf den zunächst einmal nur der Prozess selber zugreifen kann

Schützt vor Datenmissbrauch. Neben diesem Schutz der eigenen Daten werden auf diese Art aber auch Zugriffe auf fremde Speicherbereiche unterbunden. Der Prozess bzw. die App läuft also defaultmäßig in einer nach außen abgeschotteten Sandbox.

DAS ANDROID-SYSTEM

HAL (Hardware Abstraction Layer)

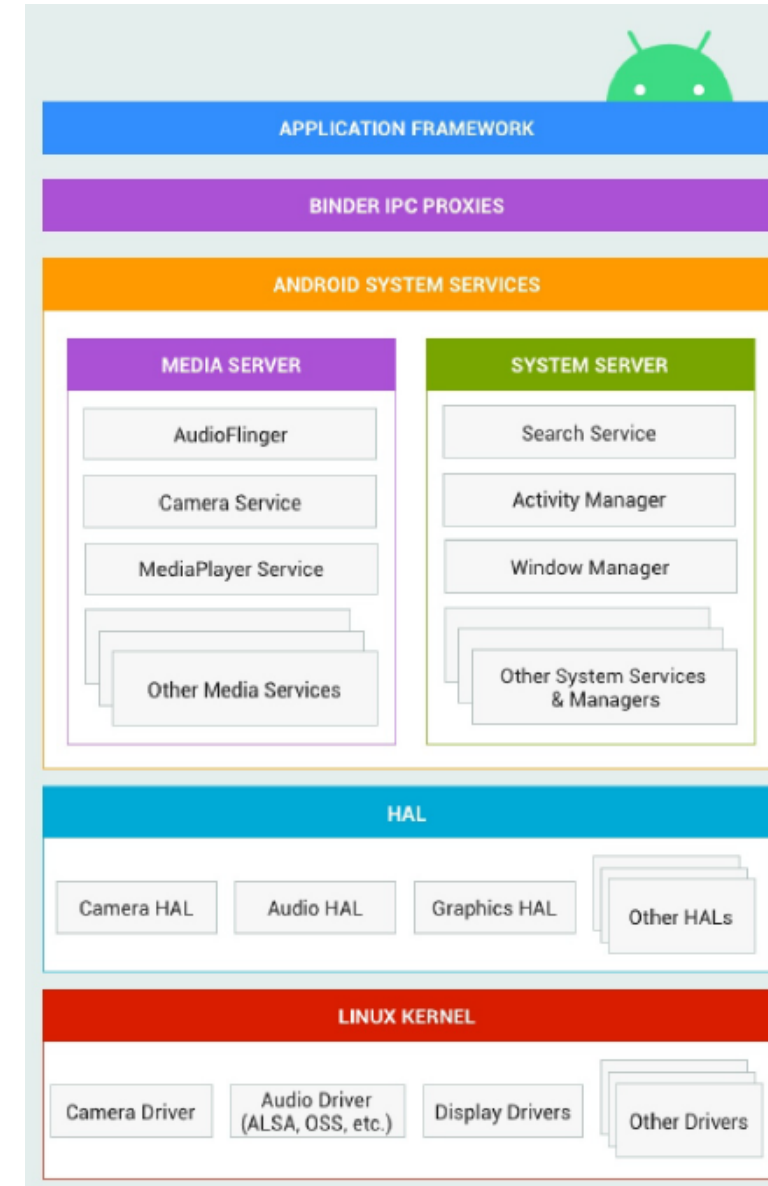
- **Funktion:** Vermittelt zwischen der Hardware und dem Android-System.
- **Komponenten:**
 - **Camera HAL:** Schnittstelle zur Kamerahardware.
 - **Audio HAL:** Verwaltung von Audiogeräten und -signalen.
 - **Graphics HAL:** Steuerung der Grafikausgabe und -darstellung.
 - **Other HALs:** Unterstützung weiterer Hardwarekomponenten, je nach Gerät.



DAS ANDROID-SYSTEM

Linux Kernel

- **Funktion:** Die technische Basis des Android-Systems, angepasst an die Anforderungen mobiler Geräte.
- **Komponenten:**
 - **Camera Driver:** Zugriff auf die Kamerafunktionalität.
 - **Audio Driver:** Verarbeitung und Steuerung von Audioeingabe und -ausgabe (ALSA, OSS, etc.).
 - **Display Drivers:** Steuerung der Display-Hardware.
 - **Other Drivers:** Treiber für weitere Hardwarekomponenten wie Speicher, Sensoren oder Netzwerkgeräte.



Laufzeitumgebung/Android Runtime:

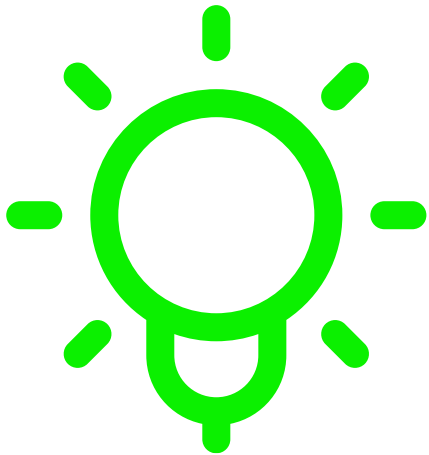
Bis Android 4.4 **Just-In-Time-Compiler**

- Dieser erzeugt den ausführbaren prozessorabhängigen Maschinen-Bytecode aus dem vorliegenden prozessorunabhängigen Bytecode bei jedem Start einer Anwendung.

Ab Version 5.0 ist **Ahead-Of-Time-Compiler**:

- Dieser erzeugt den ausführbaren prozessorabhängigen Maschinen-Bytecode aus dem vorliegenden prozessorunabhängigen Bytecode einmalig bei der Installation einer Anwendung.
- Nachteil: Erhöhter Speicherbedarf und Installation der App dauert länger.
- Vorteil: Der Start der Anwendung läuft schneller ab → insbesondere bei Anwendungen, die nicht ständig durchlaufen und stattdessen mehrfach gestartet werden, ist das ein spürbarer Performance-Gewinn für den Endbenutzer.

Wähle einen Systemdienst aus dem Application Framework (z. B. den **Activity Manager** oder **Window Manager**) und recherchiere folgende Punkte:



- Welche Rolle spielt dieser Dienst in der Android-Systemarchitektur?
- Wie interagiert der Dienst mit anderen Komponenten (z. B. dem Linux Kernel oder den APIs)?
- Gib ein konkretes Beispiel aus der Praxis, das zeigt, wie dieser Dienst verwendet wird.

02

ANDROID-SYSTEMARCHITEKTUR

SICHERHEIT

Berechtigungen (Permissions)

- Apps fordern vor der Installation Berechtigungen für bestimmte Zugriffe an.
- Ab Android 6.0 ("Marshmallow") können Berechtigungen auch zur Laufzeit (dynamisch) genehmigt oder verweigert werden, anstatt nur bei der Installation.
- Endnutzer muss die Berechtigungen vor der Installation explizit genehmigen.
- Ablehnung von Berechtigungen kann die Installation verhindern.

Berechtigungskonzept schützt die Privatsphäre und ermöglicht Kontrolle über App-Zugriffe.

Beispiele für Berechtigungen:

- Aufbau einer Internetverbindung.
- Auslesen von Kontakten.
- Ermittlung des Standorts über GPS.
- Versand und Empfang von SMS-Nachrichten.

Sandbox-Prinzip

- Jede App läuft in einem eigenen Betriebssystemprozess.
- Apps haben einen dedizierten Speicherbereich im Hauptspeicher und Dateisystem, auf den andere Apps nicht zugreifen können.

Vorteile des Sandbox-Prinzips:

- Schutz der App-Daten vor Zugriff durch andere Apps.
- Reduziert Risiken von Datenmissbrauch und Sicherheitslücken.

Erweiterung des Schutzes durch Berechtigungen:

- Nutzer entscheidet, ob und welche App auf bestimmte Daten oder Dienste zugreifen darf.

Client-Server-Architektur in Android

- Apps basieren oft auf einer Client-Server-Architektur.

Funktionen und Daten können zwischen Gerät (Client) und Server verteilt werden, z. B.:

- Apps mit Cloud-Synchronisation.
- Kommunikation über Internet-APIs.

02

ANDROID-SYSTEMARCHITEKTUR

KOMMUNIKATION MIT NETZWERKEN

Verteilte Systeme und Client-Server-Architektur

■ Verteilte Systeme:

- Funktionen und Daten auf mehrere Rechner verteilt.
- Kommunikation zwischen Rechnern erfolgt über Nachrichten in Netzwerken.
- Netzwerke unterscheiden sich in Sicherheit und Eigenschaften.

■ Client-Server-Architektur:

- Server:
 - Bereitstellung zentraler Dienste (z. B. Datenbanken, Berechtigungsprüfungen).
 - Sicherer Standort, meist Rechenzentrum mit Zugriffskontrolle.
- Client:
 - Endgerät, das die Schnittstelle zum Nutzer darstellt (z. B. mobiles Gerät).

URI, URL und URN

- URI (Uniform Resource Identifier): Identifikator für Ressourcen, unabhängig von der Lokalisierung.
 - Bestandteile:
 - ✓ Scheme (z. B. http): Kontext/Protokoll.
 - ✓ Authority (z. B. www.example.com): Optional, Host-Information.
 - ✓ Path (z. B. /resource): Hierarchische Lokalisierung.
 - ✓ Query (z. B. ?id=123): Zusätzliche Parameter, oft für Datenbankabfragen.
 - ✓ Fragment (z. B. #section): Angabe eines Ressourcenteils.
 - ✓ **URI = <scheme>:[<authority>]<path>[„?“<query>][„#“<fragment>]**
- URL (Uniform Resource Locator): **URI mit Zugriffsmethode** (z. B. http://...).
- URN (Uniform Resource Name): URI ohne Lokalisierungsangabe (z. B. urn:isbn:9783123456789).

HTTP und REST

- **HTTP (Hypertext Transfer Protocol):**
 - Protokoll zur Datenübertragung in Netzwerken, z. B. Internetseiten.
 - Ermöglicht auch technische Kommunikation zwischen Anwendungen.
- **REST (Representational State Transfer):**
 - Prinzip für verteilte Systeme.
 - Ressourcen werden durch URIs eindeutig adressiert.
 - Fokus auf einfache, ressourcenorientierte Schnittstellen.
- **SOAP:**
 - Protokoll zum Datenaustausch auf Basis von XML und HTTP.

Mobiler Netzzugang: GSM, LTE, 5G, WLAN

Mobilfunknetze und WLAN:

- GSM (2G): 9,6 kBit/s; LTE (4G): Bis zu 100 Mbit/s; 5G (5. Generation): Bis zu 10 Gbit/s, geringere Latenzzeiten.
- WLAN: Schnellere Verbindung und geringere Kosten im Vergleich zu Mobilfunk, aber keine konstante Verfügbarkeit.

Uplink vs. Downlink:

- Uplink: Endgerät → Server.
- Downlink: Server → Endgerät.

Herausforderungen für mobile Anwendungen:

- Netzwerkabbrüche: Design und Test müssen dies berücksichtigen.
- Minimierung der Anzahl von Netzwerkanfragen für bessere Performance.

03

ENTWICKLUNGSUMGEBUNG

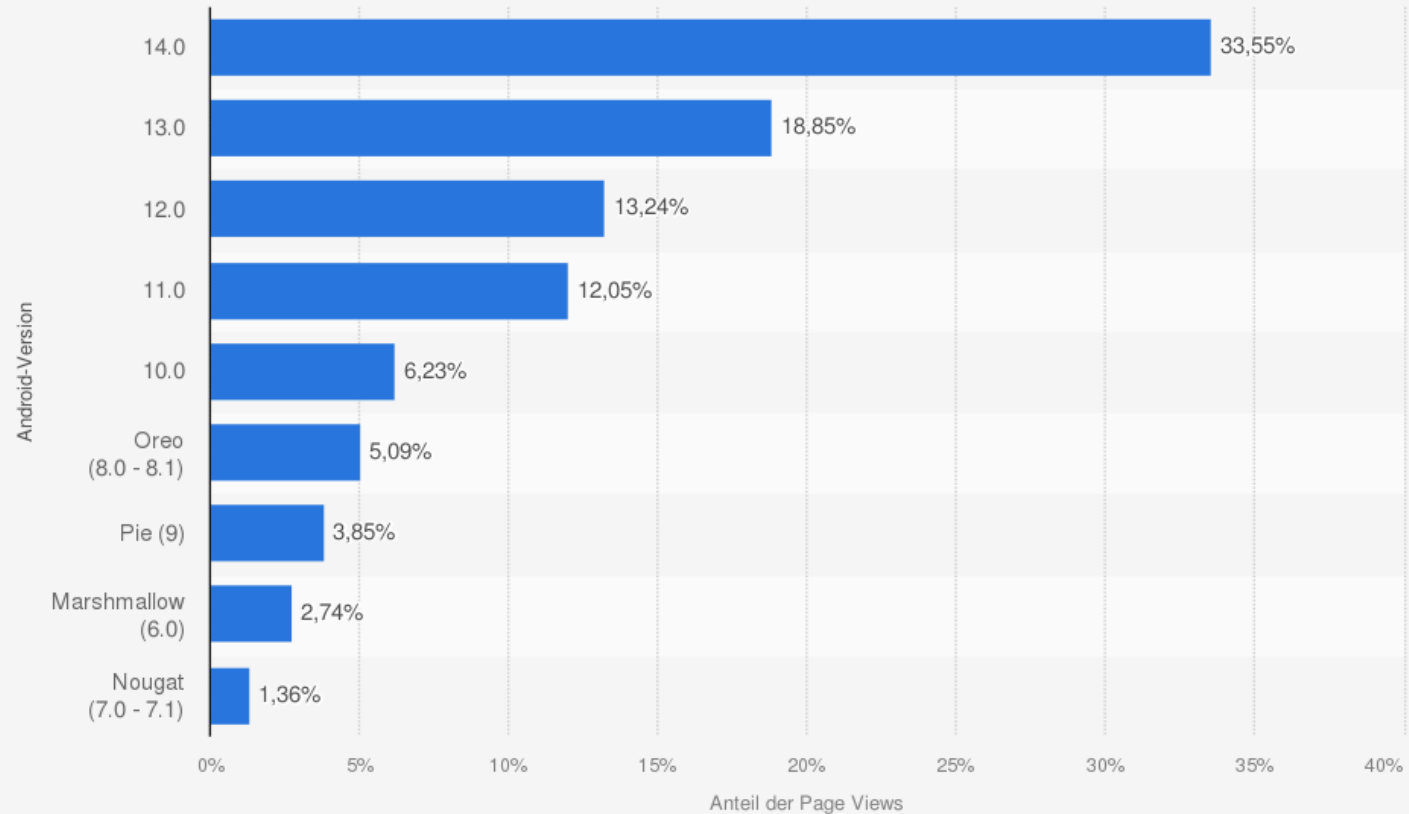
WORUM GEHT ES IN DIESEM KAPITEL?

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- wie man die Entwicklungsumgebung Android Studio installiert und einsetzt.
- wie ein Test einer Android-Anwendung auf einem Emulator durchgeführt werden kann.
- wie man eine Android-Applikation auf ein reales Endgerät lädt.

- Android Studio ist die offizielle, von Google empfohlene Umgebung für Android-Entwicklung.
 - Google hat die Weiterentwicklung wichtiger Plugins für Eclipse eingestellt, weshalb Android Studio zukunftssicher ist.
 - Android Studio wird über die offiziellen Android-Developer-Webseiten bezogen.
-
- Ihr installiert euch bitte die aktuellste Version und die Erweiterungen, die ihr für Kotlin/Jetpack benötigt

Anteile der verschiedenen Android-Versionen an der Internetnutzung von Geräten mit Android OS weltweit im Oktober 2024



Quelle
StatCounter
© Statista 2025

Weitere Informationen:
Weltweit

Kompatibilität der Android-Versionen

- Die Kompatibilität zwischen verschiedenen Android-Versionen und Geräten ist ein wichtiger Aspekt für Benutzer, die sicherstellen möchten, dass ihre Apps und Funktionen reibungslos funktionieren. Hier sind einige Schlüsselpunkte zur Kompatibilität von Android-Versionen:

Name	Versionsnummer	Veröffentlichungsdatum	API-Level
Android 8.0 (Oreo)	8.0	21. August 2017	26
Android 9 (Pie)	9	6. August 2018	28
Android 10	10	3. September 2019	29
Android 11	11	8. September 2020	30
Android 12	12	4. Oktober 2021	31
Android 13	13	15. August 2022	33
Android 14	14	4. Oktober 2023	34

Allgemeine Kompatibilität

Android-Apps und API-Level:

- Jede Android-Version hat ein spezifisches API-Level, das angibt, welche Funktionen und Klassen in der Programmierschnittstelle verfügbar sind.
- Entwickler müssen sicherstellen, dass ihre Apps auf die Mindestanforderungen der Android-Versionen abzielen, um im Google Play Store veröffentlicht zu werden.

Name	Versionsnummer	Veröffentlichungsdatum	API-Level
Android 8.0 (Oreo)	8.0	21. August 2017	26
Android 9 (Pie)	9	6. August 2018	28
Android 10	10	3. September 2019	29
Android 11	11	8. September 2020	30
Android 12	12	4. Oktober 2021	31
Android 13	13	15. August 2022	33
Android 14	14	4. Oktober 2023	34

Allgemeine Kompatibilität

- Sicherheitsupdates: Google stellt Sicherheitsupdates nur für Android-Versionen ab 11 bereit.
- Ältere Versionen erhalten keine neuen Sicherheitsupdates, was die Nutzung unsicher machen kann.
- Herstellerabhängigkeit: Die Verfügbarkeit von Updates und spezifischen Funktionen kann je nach Gerätehersteller variieren. Nicht alle Hersteller aktualisieren ihre Geräte zeitnah oder bieten alle Funktionen an, die in den neuesten Android-Versionen verfügbar sind

Name	Versionsnummer	Veröffentlichungsdatum	API-Level
Android 8.0 (Oreo)	8.0	21. August 2017	26
Android 9 (Pie)	9	6. August 2018	28
Android 10	10	3. September 2019	29
Android 11	11	8. September 2020	30
Android 12	12	4. Oktober 2021	31
Android 13	13	15. August 2022	33
Android 14	14	4. Oktober 2023	34

WAS SIND DIE UNTERSCHIEDE? BEISPIELE:

Android 4.4 „KitKat“

Transparente Status- und Navigationsleisten: Ermöglicht Apps, die Statusleiste zu überlagern, sodass sie in das Design der App integriert werden können.

Immersive Mode: Apps können nun im Vollbildmodus ausgeführt werden, wodurch Status- und Navigationsleisten ausgeblendet werden können, bis der Benutzer eine Geste macht.

NFC Host Card Emulation: Diese Funktion ermöglichte es Geräten, als NFC-Karten zu agieren, was eine Grundlage für mobile Zahlungen und andere NFC-basierte Apps bildete.

Android 9.0 “Pie”

App Actions und Slices: Apps können proaktiv Vorschläge und Aktionen basierend auf den Nutzungsgewohnheiten des Benutzers anbieten. Slices ermöglichen es Apps, Teile ihrer Benutzeroberfläche direkt in Suchergebnisse oder Google Assistant einzubetten.

Digital Wellbeing: Einführung von Funktionen, die den Nutzern helfen, ihre Bildschirmzeit zu verwalten, darunter Dashboard, App Timer und der „Wind Down“-Modus, um die Nutzung vor dem Schlafengehen zu reduzieren.

Sicherheitsverbesserungen: Apps im Hintergrund können nun nicht mehr auf Kamera, Mikrofon und andere Sensoren zugreifen, was den Datenschutz verbessert.

Android 12.0 “Snow Cone”

Privatsphäre-Dashboard: Ein zentrales Dashboard, das den Benutzern zeigt, welche Apps kürzlich auf Kamera, Mikrofon und Standortdaten zugegriffen haben.

Mikrofon- und Kamerazugriffsindikatoren: Ein kleines Symbol in der Statusleiste zeigt an, wenn eine App die Kamera oder das Mikrofon verwendet.

Game Mode API: Ermöglicht Entwicklern, die Benutzererfahrung von Spielen durch spezielle Optimierungen zu verbessern, z.B. durch Leistungsmodi oder die Verlängerung der Akkulaufzeit während des Spielens.

03

ENTWICKLUNGSUMGEBUNG

ANDROID STUDIO

- **Android Studio** ist die offizielle Entwicklungsumgebung (IDE) für Android-Anwendungen, basierend auf IntelliJ IDEA. Es bietet eine umfassende Suite von Tools zur Entwicklung, Testung und Fehlerbehebung von Android-Apps.
- Ähnlich wie auch andere IDE's umfasst Android Studio die folgenden Hauptfunktionen:
 - Intelligentes Code-Editing mit Unterstützung für Java, Kotlin, und C++.
 - Visueller Layout-Editor für einfaches Design von UI-Komponenten.
 - Emulator für Tests ohne physische Geräte.
 - Integriertes Version Control System (Git, GitHub, etc.).



- **Android Studio** ist die offizielle Entwicklungsumgebung (IDE) für Android-Anwendungen, basierend auf IntelliJ IDEA. Es bietet eine umfassende Suite von Tools zur Entwicklung, Testung und Fehlerbehebung von Android-Apps.
- Warum Android Studio?
 - Offizielle IDE für Android-Entwicklung
 - Integriertes Gradle-Build-System
 - Verbesserter visueller Layout-Editor
 - Integrierter Android Emulator



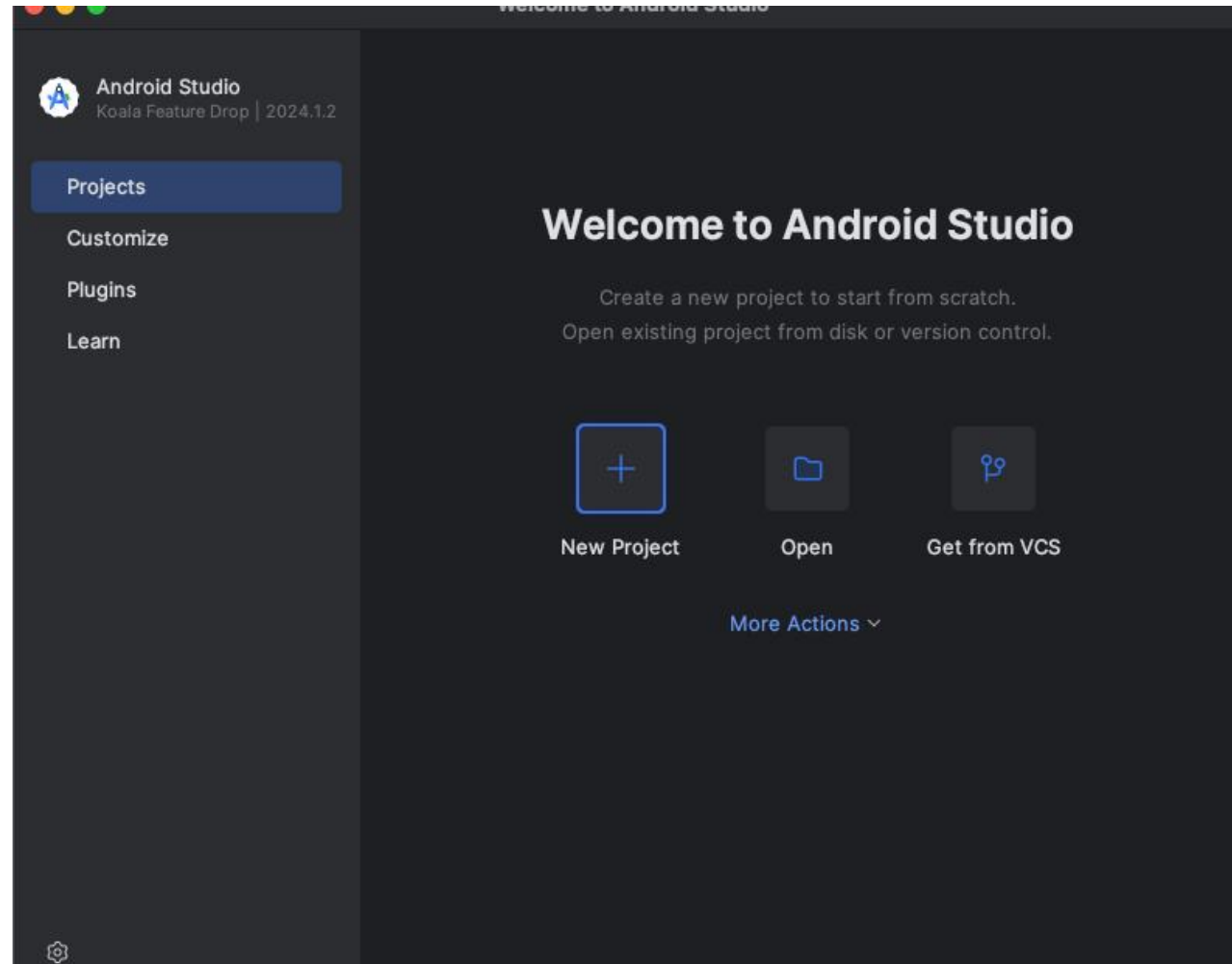
- Das **Android Software Development Kit** (SDK) ist eine Sammlung von Tools, Bibliotheken und APIs, die für die Entwicklung von Android-Apps notwendig sind. Es umfasst alles, was Entwickler benötigen, um Android-spezifische Funktionen zu nutzen, wie die Steuerung von Hardware-Sensoren, die Verwaltung von Benutzeroberflächen und die Kommunikation zwischen verschiedenen Komponenten.
 - Umfasst Tools, Bibliotheken und APIs.
 - Ermöglicht den Zugriff auf Android-spezifische Hardware und Software-Funktionen.
 - Integriert in Android Studio über den SDK Manager.
 - Unterstützt verschiedene Android-Versionen.



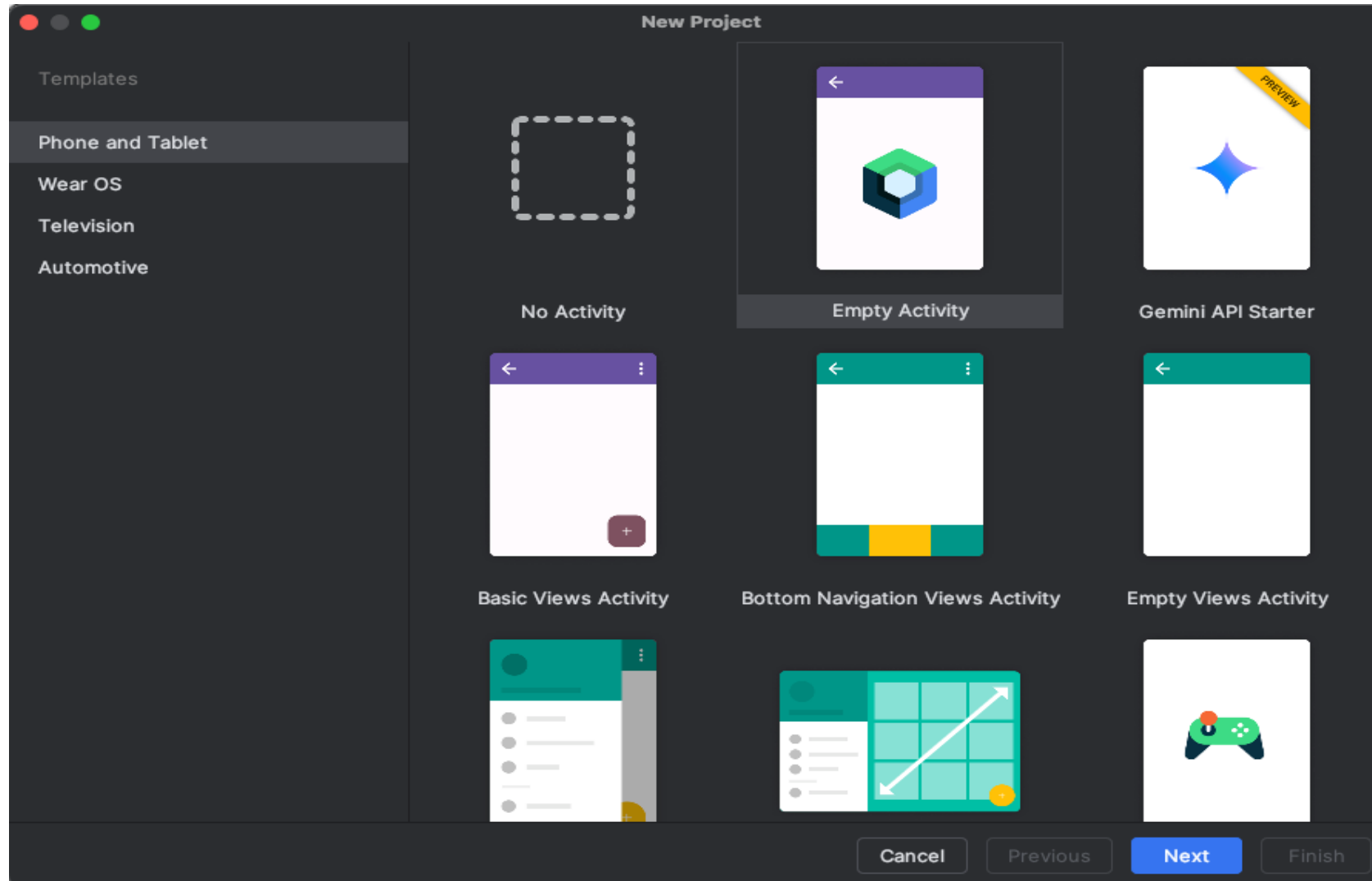
03

ENTWICKLUNGSUMGEBUNG

ERSTE APP UND DER EMULATOR-TEST



ANDROID STUDIO



ANDROID STUDIO – HALLO WELT

```
package com.example.helloworld

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloworld.ui.theme.HelloWorldTheme
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "World",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

import android.os.Bundle

→ Bundle ist so etwas wie ein Datenpaket, in dem Android dir Zustandsdaten übergibt (ob eine App pausiert, neu gestartet wird, ein früherer Zustand wieder hergestellt wird)

ANDROID STUDIO – HALLO WELT

```
package com.example.helloworld

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloworld.ui.theme.HelloWorldTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "World",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

import androidx.activity.ComponentActivity
→ Basis-Klasse für Activity (optimiert für JetPack)

import androidx.activity.compose.setContent
→ Wird genutzt um Compose-Inhalte in einer Activity darzustellen (entgegen älterer Android Versionen, wo der Inhalt der App in XML-Dateien gespeichert wurde, wird nun der Inhalt direkt in die App geschrieben)

import androidx.activity.enableEdgeToEdge
→ Diese Funktion aktiviert die Edge-to-Edge-Darstellung. Das bedeutet: Inhalte können unter Statusleiste, Navigationsleiste etc. gezeichnet werden – wie bei modernen, rahmenlosen Designs.

ANDROID STUDIO – HALLO WELT

```
package com.example.helloworld

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloworld.ui.theme.HelloWorldTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "World",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

import androidx.compose.foundation.layout.fillMaxSize

→ Diese Funktion ist ein Modifier, der dafür sorgt, dass ein Element den gesamten verfügbaren Platz im übergeordneten Container ausfüllt – sowohl in der Breite als auch in der Höhe.

import androidx.compose.foundation.layout.padding

→ Auch das ist ein Modifier, mit dem du einem Composable Abstand von seinen Rändern geben kannst – also Innenabstand (Padding).

ANDROID STUDIO – HALLO WELT

```
package com.example.helloworld

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloworld.ui.theme.HelloWorldTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "World",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

import androidx.compose.material3.Scaffold

→ Scaffold ist ein Layout-Grundgerüst für typische App-Oberflächen. Es stellt dir vorgefertigte Bereiche zur Verfügung – wie z. B. Top App Bar, FloatingActionButton, Drawer, etc.

import androidx.compose.material3.Text

→ Text ist ein Composable, um einfachen Text auf dem Bildschirm darzustellen.

import androidx.compose.runtime.Composable

→ Dies importiert die Annotation @Composable, die du brauchst, um eigene UI-Funktionen in Compose zu erstellen.

ANDROID STUDIO – HALLO WELT

```
package com.example.helloworld

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloworld.ui.theme.HelloWorldTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "World",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

import androidx.compose.ui.Modifier

→ Modifier wird verwendet, um das Layout und das Aussehen von Composables zu verändern, z. B. für Größenanpassungen oder Abstände.

import androidx.compose.ui.tooling.preview.Preview

→ Mit der @Preview-Annotation kannst du eine Vorschau deiner Composables direkt im Android Studio anzeigen lassen, ohne die App auf einem Emulator ausführen zu müssen.

import com.example.helloworld.ui.theme.HelloWorldTheme

→ Dieser Import sorgt dafür, dass das benutzerdefinierte Design (Theme) aus der HelloWorldTheme-Klasse auf die Composables angewendet wird, was das visuelle Erscheinungsbild der App steuert.

ANDROID STUDIO – HALLO WELT

```
package com.example.helloworld

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloworld.ui.theme.HelloWorldTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "World",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

@Composable → Das, was wir direkt in der App, bzw. der Emulation sehen

@Preview

@Composable → Das, was wir in der Vorschau sehen

Warum haben wir eine Preview-Möglichkeit?

- Man kann visuell testen, wie eine UI aussieht.
 - Ohne Emulator.
 - Ohne echte App-Installation

Wichtig: @Preview-Funktionen dürfen keine Parameter erwarten.

Frage: Warum sind Vorschau und App manchmal nicht gleich?

```
package com.example.helloworld

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloworld.ui.theme.HelloWorldTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "World",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

Warum sind Vorschau und App manchmal nicht gleich?

Sie können unterschiedlich sein – aber sie müssen es nicht.

Das liegt daran, dass Preview völlig unabhängig von der "echten App" läuft. Du kannst:

- in der App Greeting("World") aufrufen,
- aber in der Vorschau Greeting("Android").

Das ist bewusst so, um dir mehr Flexibilität beim Designen zu geben.

Praktisches Beispiel: Login/Logout-Button (Im Preview beide Zuständig, im normalen Compose nur ein Zustand)

ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP

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

    <!-- Definiert die minimale und die Ziel-Android-Version -->
    <uses-sdk
        android:minSdkVersion="21"
        android:targetSdkVersion="33" />

    <!-- Berechtigung für den Zugriff auf das Internet -->
    <!-- Wird z.B. für Netzwerkverbindungen, APIs, etc. benötigt -->
    <uses-permission android:name="android.permission.INTERNET" />

    <!-- Berechtigung für den Zugriff auf den Standort des Benutzers (GPS) -->
    <!-- ACCESS_FINE_LOCATION gibt Zugriff auf den genauen Standort -->
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <!-- Berechtigung für den Zugriff auf den ungefähren Standort des Benutzers -->
    <!-- ACCESS_COARSE_LOCATION gibt Zugriff auf einen ungenauen Standort -->
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

    <!-- Berechtigung zum Auslesen der gespeicherten Kontakte -->
    <!-- Wird benötigt, wenn eine App auf die Kontaktliste zugreifen soll -->
    <uses-permission android:name="android.permission.READ_CONTACTS" />

    <!-- Berechtigung zum Versenden von SMS-Nachrichten -->
    <!-- Diese Berechtigung ist sehr sensibel und wird oft nur von speziellen Apps -->
    <uses-permission android:name="android.permission.SEND_SMS" />

    <!-- Berechtigung zum Speichern und Lesen von Dateien auf dem Gerät -->
    <!-- Wird benötigt, wenn die App Dateien auf dem Speicher des Geräts sichern m -->
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

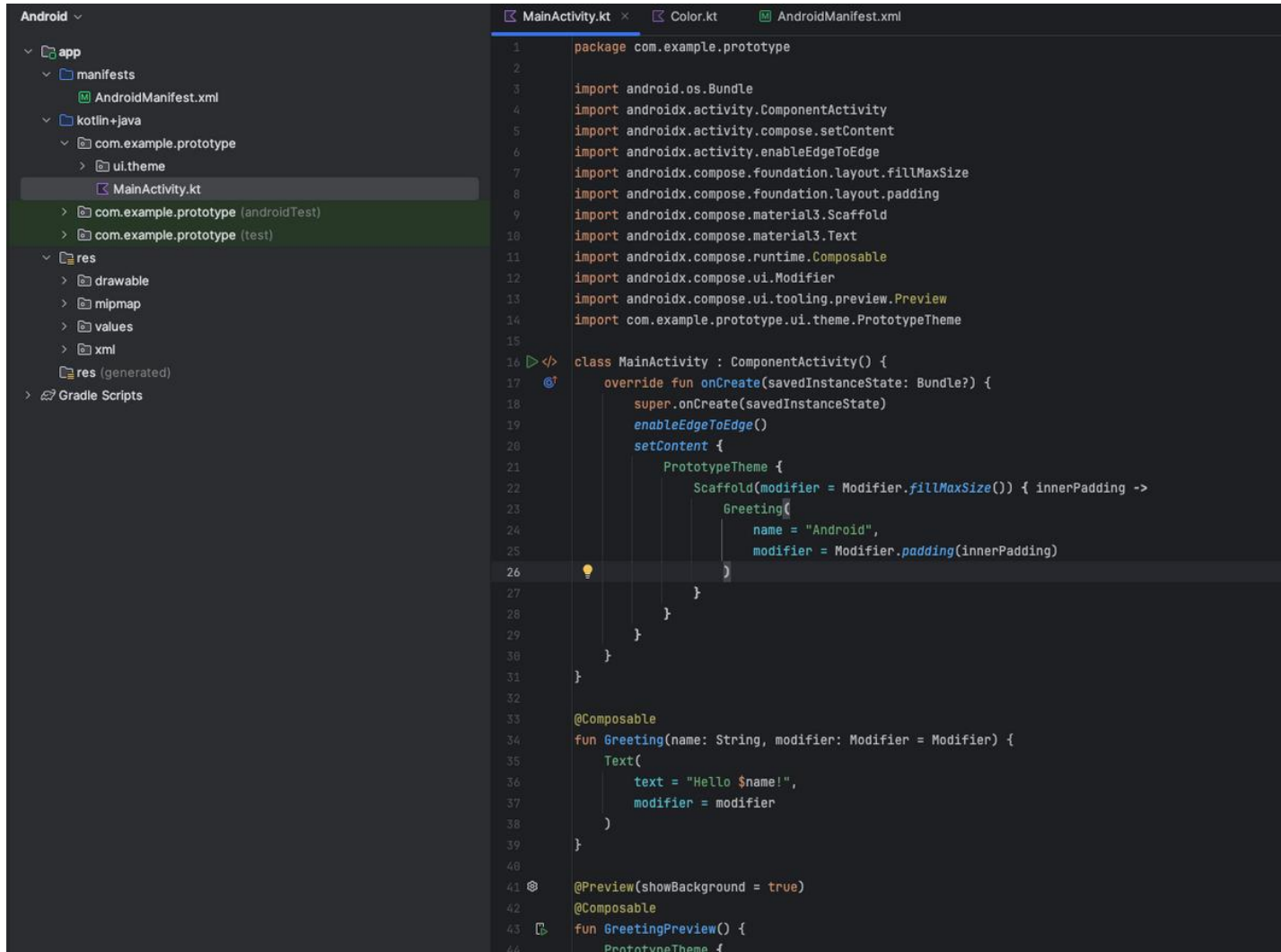
    <!-- Berechtigung zum Verwenden der Kamera -->
    <!-- Wird benötigt, um Fotos oder Videos aufzunehmen -->
    <uses-permission android:name="android.permission.CAMERA" />
```

Diese Berechtigung wird für jede App benötigt, die Daten über das Internet sendet oder empfängt, z.B. bei der Verwendung von APIs oder dem Herunterladen von Inhalten.

Diese Berechtigungen werden benötigt, wenn die App den Standort des Benutzers erfassen möchte. Der genaue Standort (z.B. GPS) erfordert ACCESS_FINE_LOCATION, während der ungenaue Standort (z.B. über WLAN oder Mobilfunkmasten) durch ACCESS_COARSE_LOCATION ermittelt werden kann.

Diese Berechtigung wird benötigt, um die Kamera des Geräts zu verwenden, z.B. für Foto-Apps oder Apps, die QR-Codes scannen.

ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP



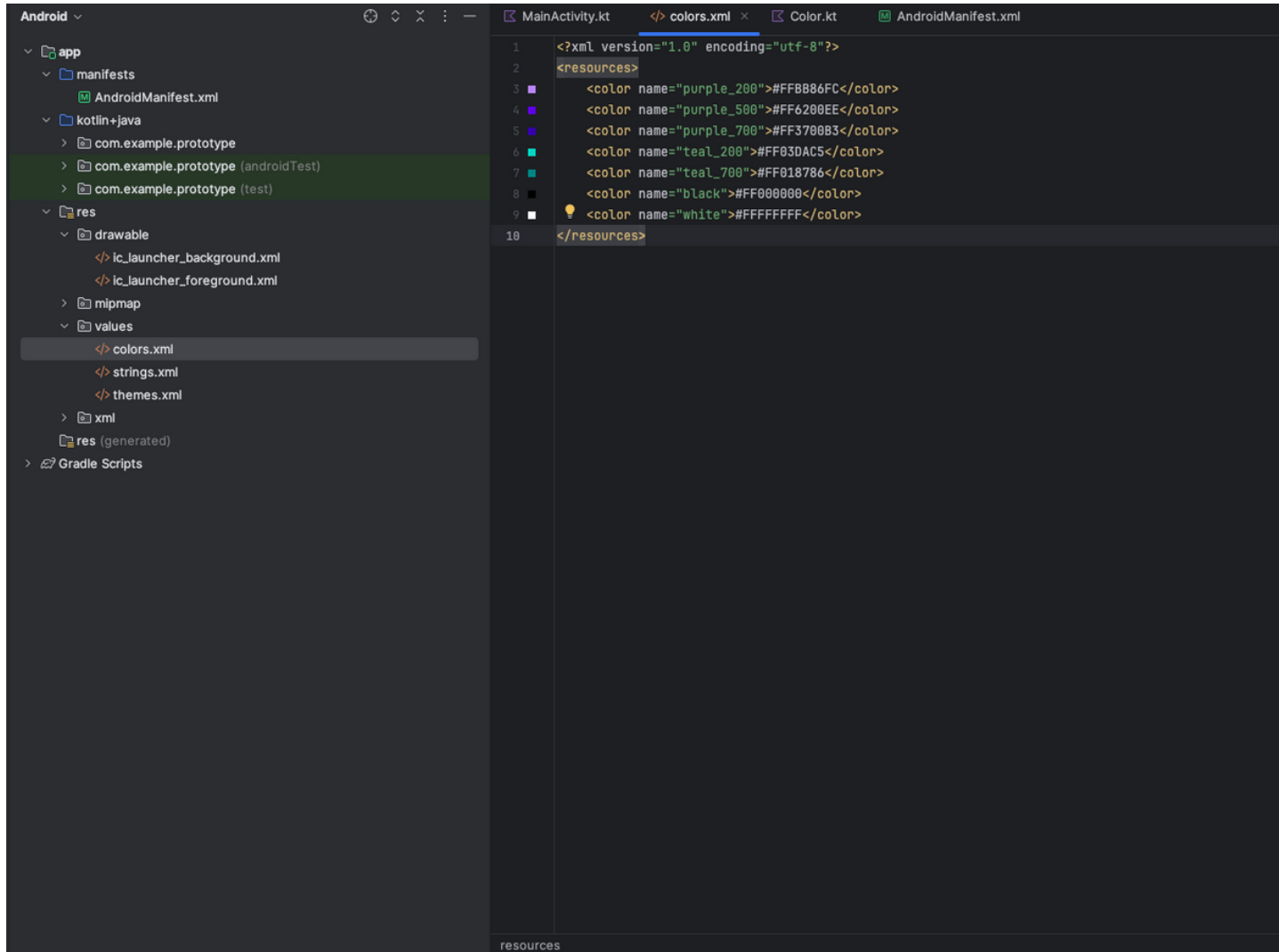
The screenshot shows the Android Studio interface. On the left, the 'Project' view displays the file structure of an Android app. The 'app' directory is expanded, showing 'manifests' (containing 'AndroidManifest.xml'), 'kotlin+java' (containing 'com.example.prototype' with 'ui.theme' and 'MainActivity.kt'), and 'res' (containing 'drawable', 'mipmap', 'values', 'xml', and 'res (generated)'). The 'MainActivity.kt' file is selected. The main editor shows the code for 'MainActivity.kt'. The code includes package declarations, imports for Android and Jetpack components, and the implementation of the 'MainActivity' class. The class overrides 'onCreate' to set the content view to 'R.layout.activity_main' and uses 'Scaffold' to display a 'Greeting' component. A 'Greeting' composable is also defined, and a preview function 'GreetingPreview' is provided for testing the UI.

```
1 package com.example.prototype
2
3 import android.os.Bundle
4 import androidx.activity.ComponentActivity
5 import androidx.activity.compose.setContent
6 import androidx.activity.enableEdgeToEdge
7 import androidx.compose.foundation.layout.fillMaxSize
8 import androidx.compose.foundation.layout.padding
9 import androidx.compose.material3.Scaffold
10 import androidx.compose.material3.Text
11 import androidx.compose.runtime.Composable
12 import androidx.compose.ui.Modifier
13 import androidx.compose.ui.tooling.preview.Preview
14 import com.example.prototype.ui.theme.PrototypeTheme
15
16 class MainActivity : ComponentActivity() {
17     override fun onCreate(savedInstanceState: Bundle?) {
18         super.onCreate(savedInstanceState)
19         enableEdgeToEdge()
20         setContent {
21             PrototypeTheme {
22                 Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
23                     Greeting(
24                         name = "Android",
25                         modifier = Modifier.padding(innerPadding)
26                     )
27                 }
28             }
29         }
30     }
31 }
32
33 @Composable
34 fun Greeting(name: String, modifier: Modifier = Modifier) {
35     Text(
36         text = "Hello $name!",
37         modifier = modifier
38     )
39 }
40
41 @Preview(showBackground = true)
42 @Composable
43 fun GreetingPreview() {
44     PrototypeTheme {
```

Dieses Verzeichnis enthält die Logik der Anwendung.

- Hier definierst du das Verhalten deiner Aktivitäten, Dienste und anderer Komponenten
- Klassen, Schnittstellen und andere Codedateien
- Herzstück der Anwendung

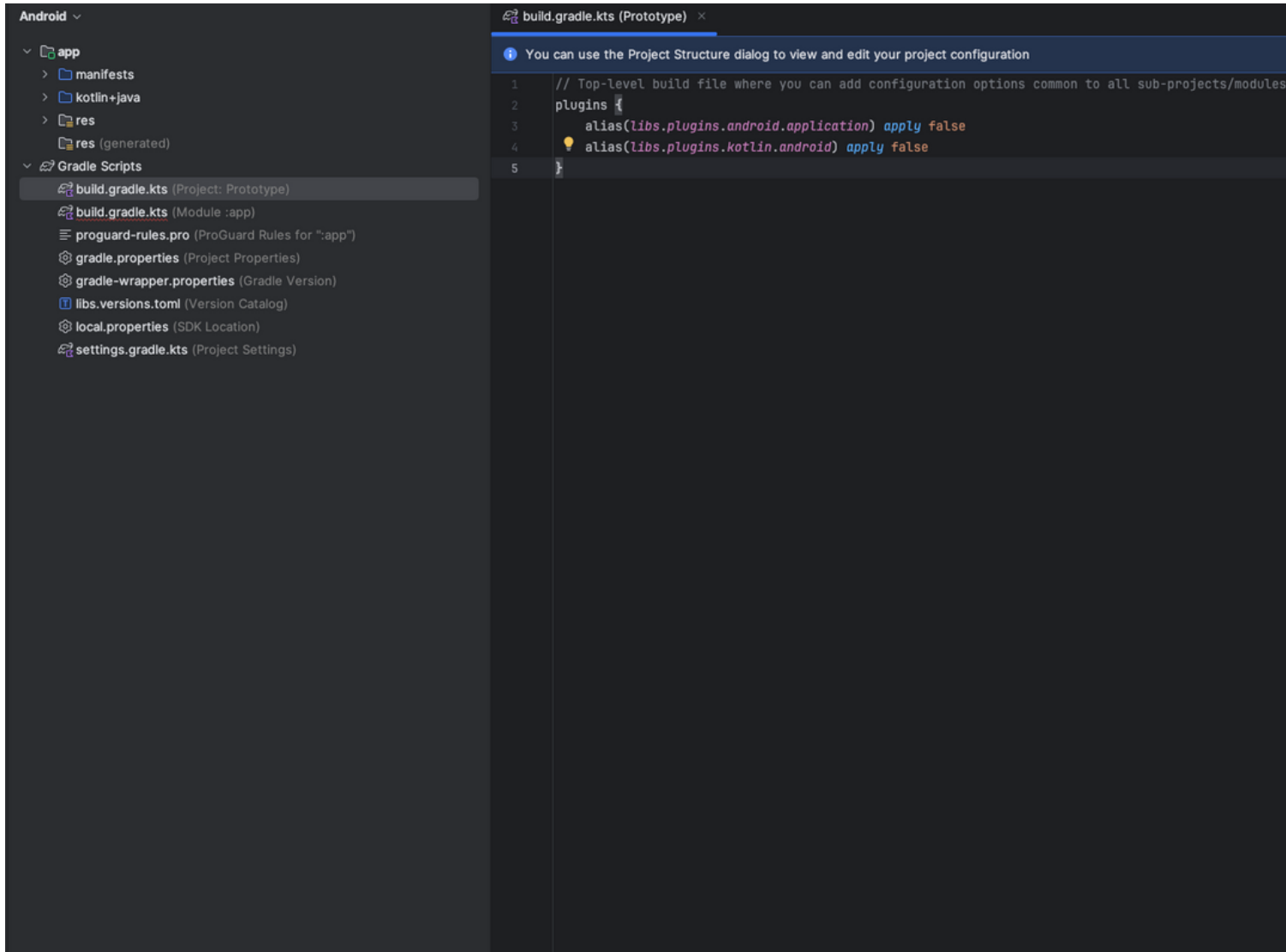
ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP



Dieser Ordner enthält verschiedene Assets, die Ihre Anwendung verwendet

- drawable/: Bilder, Icons und andere grafische Ressourcen.
- layout/: XML-Dateien, die die Benutzeroberfläche Ihrer Aktivitäten und Fragmente definieren.
- values/: String-Werte, Farben, Stile, Abmessungen und andere Konfigurationen.
- mipmap/: Verschiedene Versionen Ihres App-Symbols für verschiedene Bildschirmichten.
- raw/: Rohdateien wie Audio oder Video, auf die Ihre Anwendung zugreifen muss.

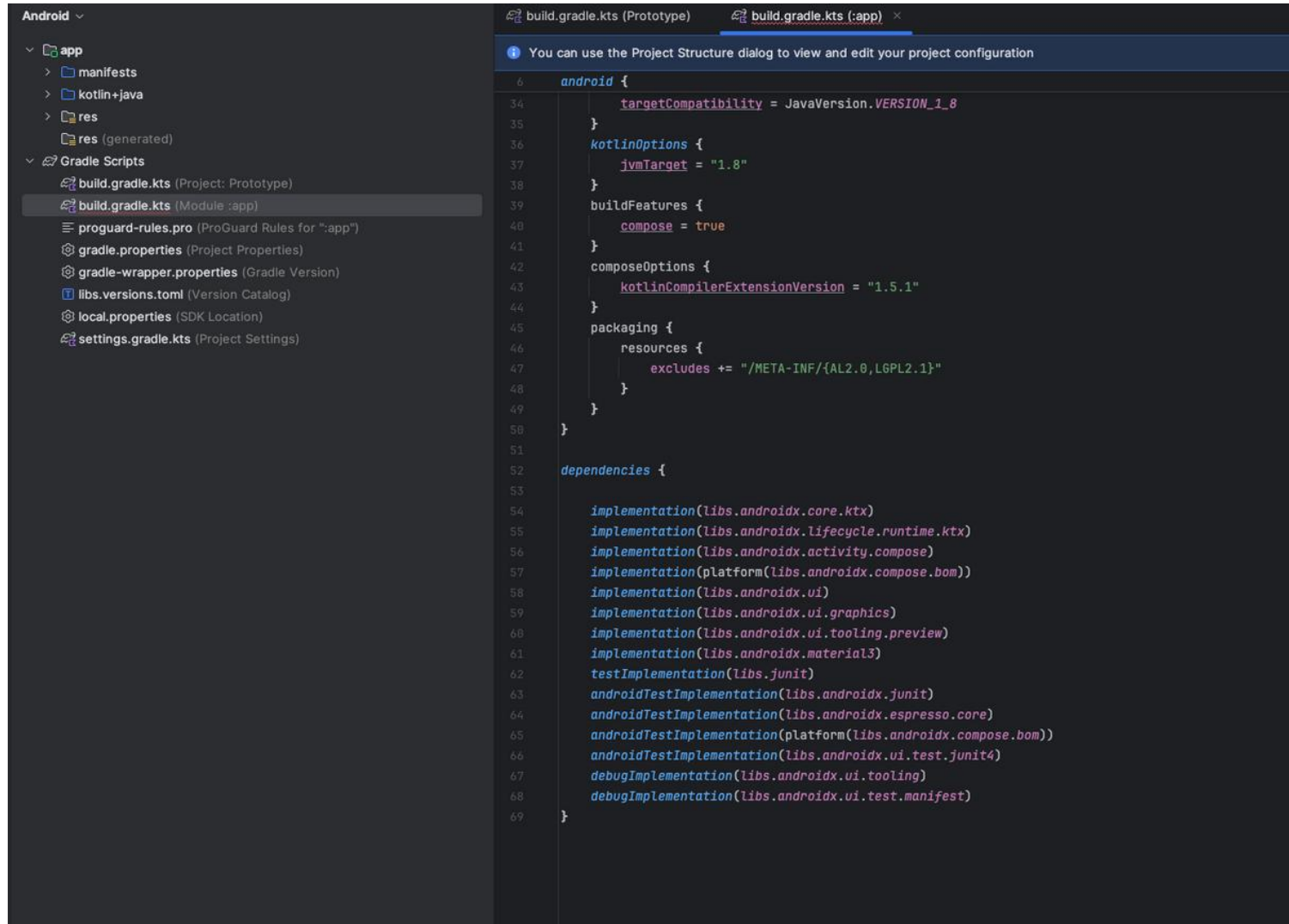
ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP



Enthält Abhängigkeiten für deine Anwendung, Build-Konfigurationen und andere Einstellungen, die für Ihr Anwendungsmodul spezifisch sind.

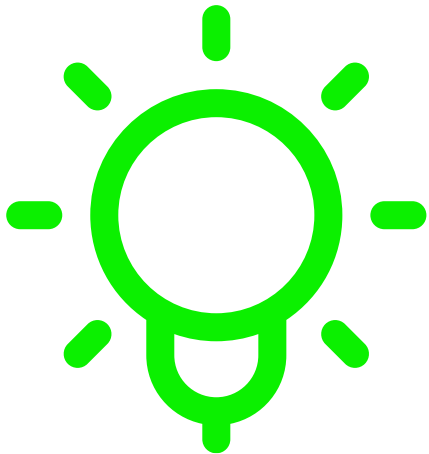
build.gradle (Projektebene): Gilt für das gesamte Projekt; in der Regel werdet ihr diese Datei nicht oft ändern müssen. Es definiert Build-Konfigurationen und Abhängigkeiten, die für alle Module in deinem Projekt gelten.

ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP



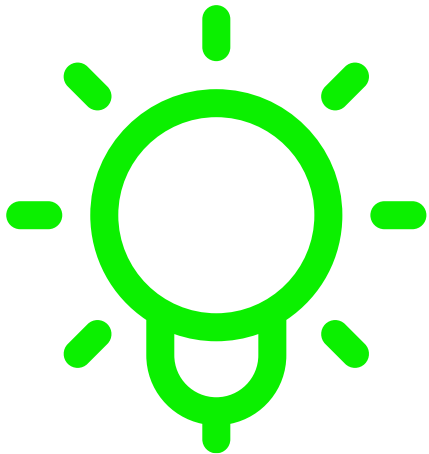
Bibliotheken bieten vorgefertigte Funktionen,
die Euch Zeit und Entwicklungsaufwand sparen.

Vergleichbar: **package.json**



- Erstelle ein einfaches Android-Projekt, um dich mit der Entwicklungsumgebung Android Studio vertraut zu machen.

- Installiere Android Studio, das Android SDK und ein Virtual Device, falls noch nicht geschehen.
- Erstelle ein neues Projekt mit folgenden Spezifikationen:
 - Projekttyp: Empty Activity.
 - Programmiersprache: Kotlin
 - Name des Projekts: Prototype (oder eigene Wahl).
- API-Level: Mindestens Android 8.0 (API Level 26).
 - Starte das Projekt entweder auf einem Emulator oder einem physischen Android-Gerät und dokumentiere die Schritte und das Ergebnis.



App Idee

Überlege dir ein Thema bzw. ein kleines(!) Projekt für eine mobile Anwendung.
Gehe auf folgende Fragen ein: :

- Beschreibe die App in einem Satz („Die App ist die ...“).
- Nenne die wichtigsten Funktionen (Chat, Navigation etc.)
- Hebe interessante Aspekte hervor

Output: Präsentiert euer Projekt anschließend der Gruppe!

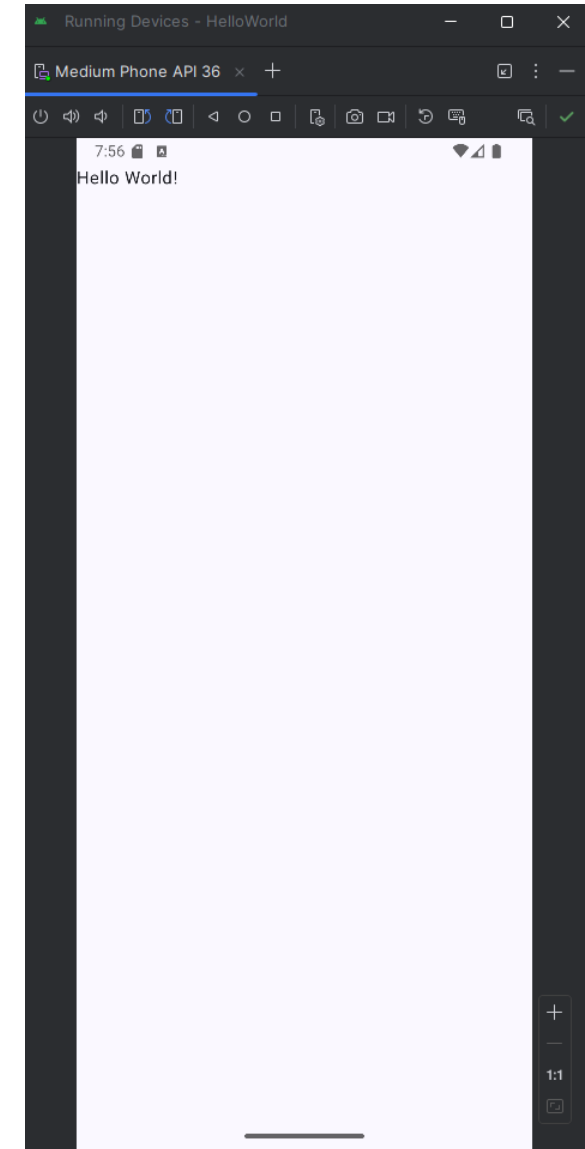
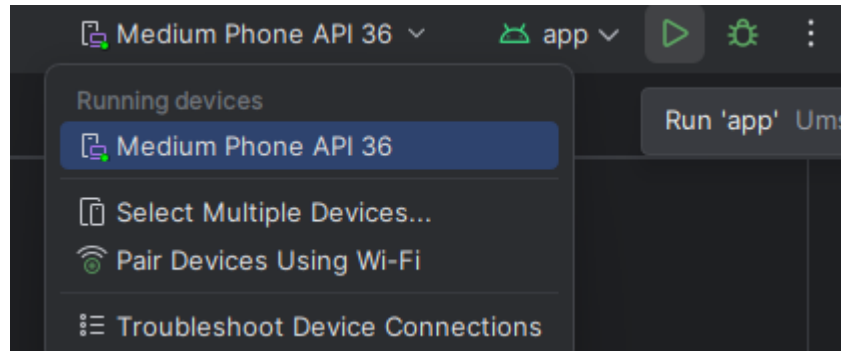
03

ENTWICKLUNGSUMGEBUNG

ANWENDUNGSDEPLOYMENT

Zwei Möglichkeiten des Anwendungsdeployment:

1. Starten auf einem angeschlossenen mobilen Endgerät
2. Starten auf dem Emulator



KERNKOMPONENTEN EINER ANDROID-APP

WORUM GEHT ES IN DIESEM KAPITEL?

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- aus welchen Bestandteilen ein Android-System besteht.
- welche dieser Komponenten für die Software-Entwicklung von Bedeutung sind.
- welche Netzwerk- und Kommunikationstechnologien für die mobile Android-Welt relevant sind.

KERNKOMPONENTEN EINER ANDROID-APP

ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP

ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP

- ✓ Android-Apps bestehen aus lose gekoppelten Komponenten mit spezifischen Aufgaben.
- ✓ Komponenten kommunizieren miteinander, um Benutzereingaben zu verarbeiten oder Systemereignisse zu melden.
- ✓ Entwickler planen Apps komponentenbasiert, teilen Funktionalitäten auf und nutzen die vorhandenen Komponententypen!

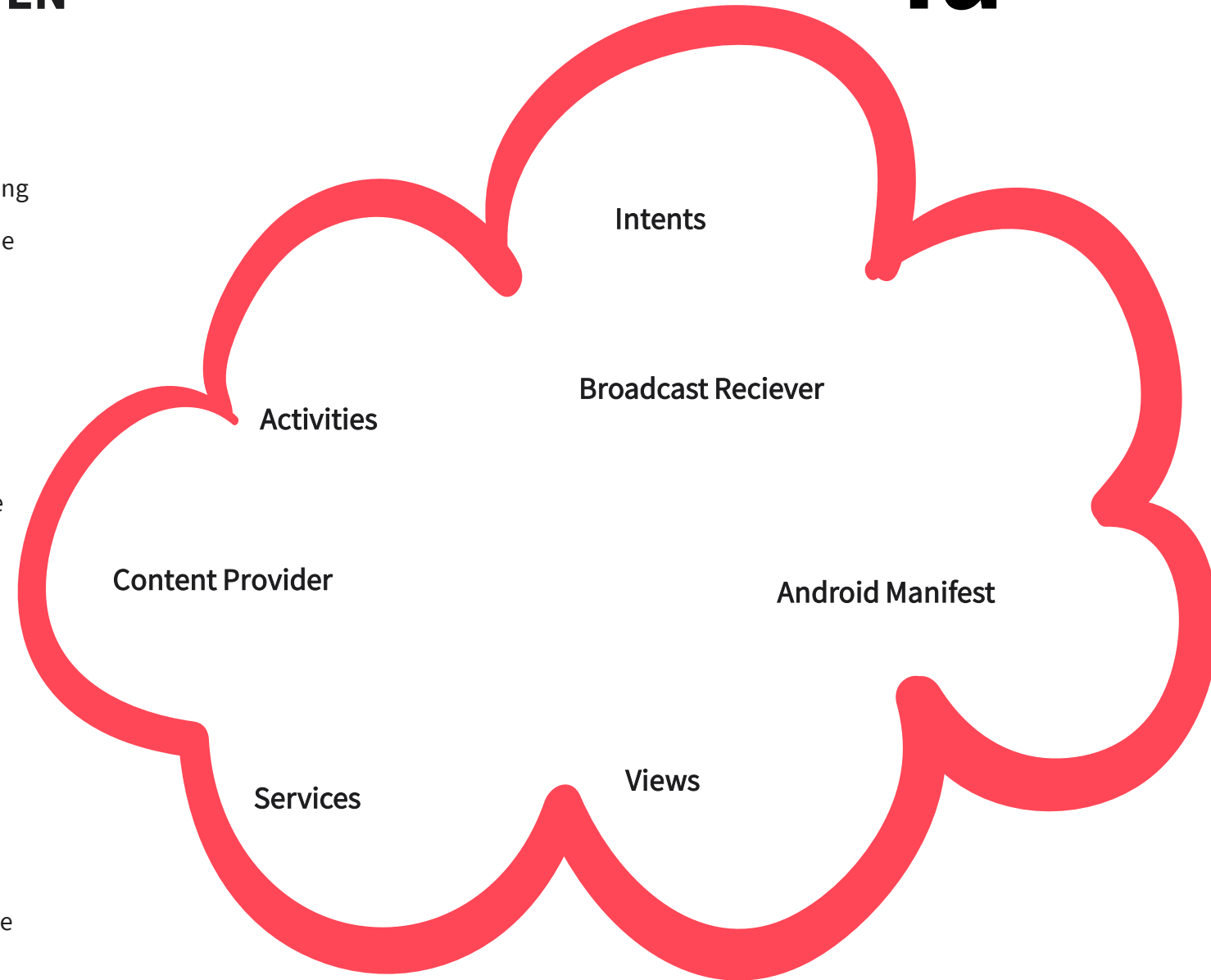
ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP

Komponenten basierte Entwicklung: Eine Android-App ist eine Sammlung von lose gekoppelten Komponenten. Jede dieser Komponenten hat eine spezielle Aufgabe.

Komponenten. Diese sind Bestandteile eines Softwaresystems, die unabhängig von anderen Bestandteilen eine bestimmte Funktionalität zur Verfügung stellen.

Werkzeugkasten: Welche Komponententypen werden angeboten und andererseits wie kann die angeforderte App-Funktionalität in solche Komponenten aufgeteilt werden.

Third Party: Komponenten können auch über Drittanbieter in das eigene System integriert werden (Bsp. Komponenten Libraries)



ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP

Activities (Präsentationsschicht):

- Repräsentieren einzelne Benutzeroberflächen (Bildschirmseiten) einer App.
- Verantwortlich für die Darstellung und Interaktion mit dem Nutzer.
- Zentrale Rolle für den ersten Eindruck und die Benutzererfahrung.
- Der Wechsel zwischen Activities erfolgt über *Intent*

Intents (Nachrichtenübermittlung):

- Ermöglichen die Kommunikation zwischen App-Komponenten oder zwischen verschiedenen Apps.
- Explizite Intents: Zielkomponente innerhalb der eigenen App ist bekannt.
- Implizite Intents: System wählt eine geeignete App für die Aktion (z. B. Mailversand).

ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP

Broadcast Receiver (Reaktionskomponenten)

- Reagieren auf empfangene Intents und führen entsprechende Aktionen aus.
- Beispiel: Verarbeitung eines "Mail senden"-Intents durch eine E-Mail-App.

Content Provider (Datenzugriffsschicht)

- Ermöglichen den strukturierten Zugriff auf App-interne Daten (z. B. Kontakte).
- Häufig in Kombination mit SQLite-Datenbanken verwendet.
- Unterstützen die Datenfreigabe für andere Apps bei entsprechender Berechtigung.

ÜBERBLICK ÜBER DIE KOMPONENTEN EINER ANDROID-APP

Services (Hintergrundverarbeitung)

- Arbeiten ohne Benutzeroberfläche und laufen im Hintergrund.
- Einsatzbeispiele: Automatischer Foto-Upload, periodischer Abruf von E-Mails.
- Ideal für lang andauernde oder regelmäßig wiederkehrende Aufgaben.

Android Manifest (Konfigurationsdatei)

- Zentrale XML-Datei (AndroidManifest.xml) zur Definition aller App-Komponenten.
- Gibt Start-Activity an und regelt Berechtigungen und Intents.
- Wird beim Erstellen eines neuen Projekts automatisch generiert.

04

KERNKOMPONENTEN EINER ANDROID-APP

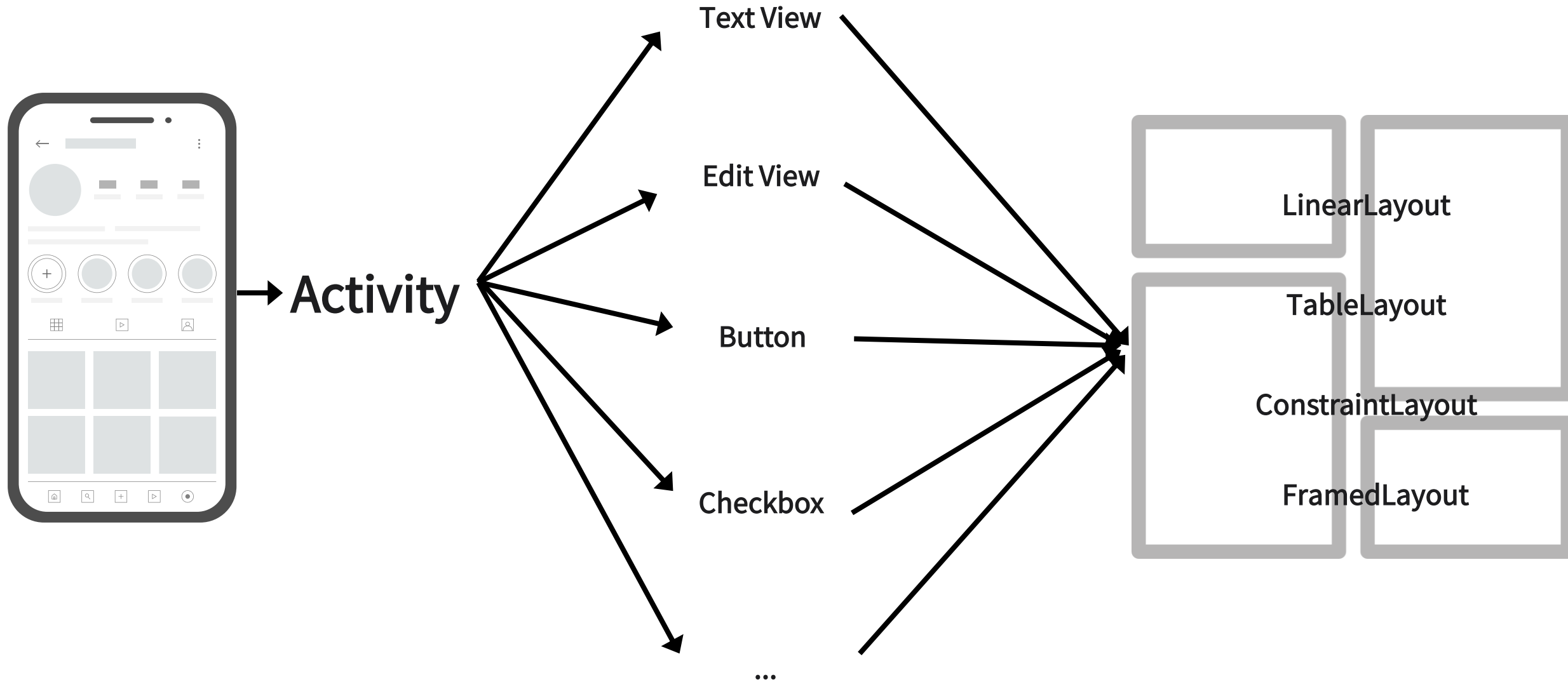
ACTIVITIES, LAYOUTS UND VIEWS

- **Activities.** Die Präsentationsschicht einer Android-App setzt sich aus Activities zusammen. Eine Activity umfasst in der Regel eine Bildschirmseite, also alles, was der Benutzer aktuell auf dem Display sieht.
- **User Interface (UI).** Das UI ist das Aushängeschild der App. Es bestimmt maßgeblich die Benutzererfahrung und beeinflusst den Erfolg der App, auch wenn die für den Endnutzer unsichtbaren Hintergrundaktivitäten die komplexesten Aufgaben übernehmen.
- **Entwicklung.** Bildschirmseiten werden durch Activities entworfen. Für eine App werden zunächst die UI-Seiten definiert (pro Seite eine Activity). Die dahinterliegende Funktionalität wird später hinzugefügt.

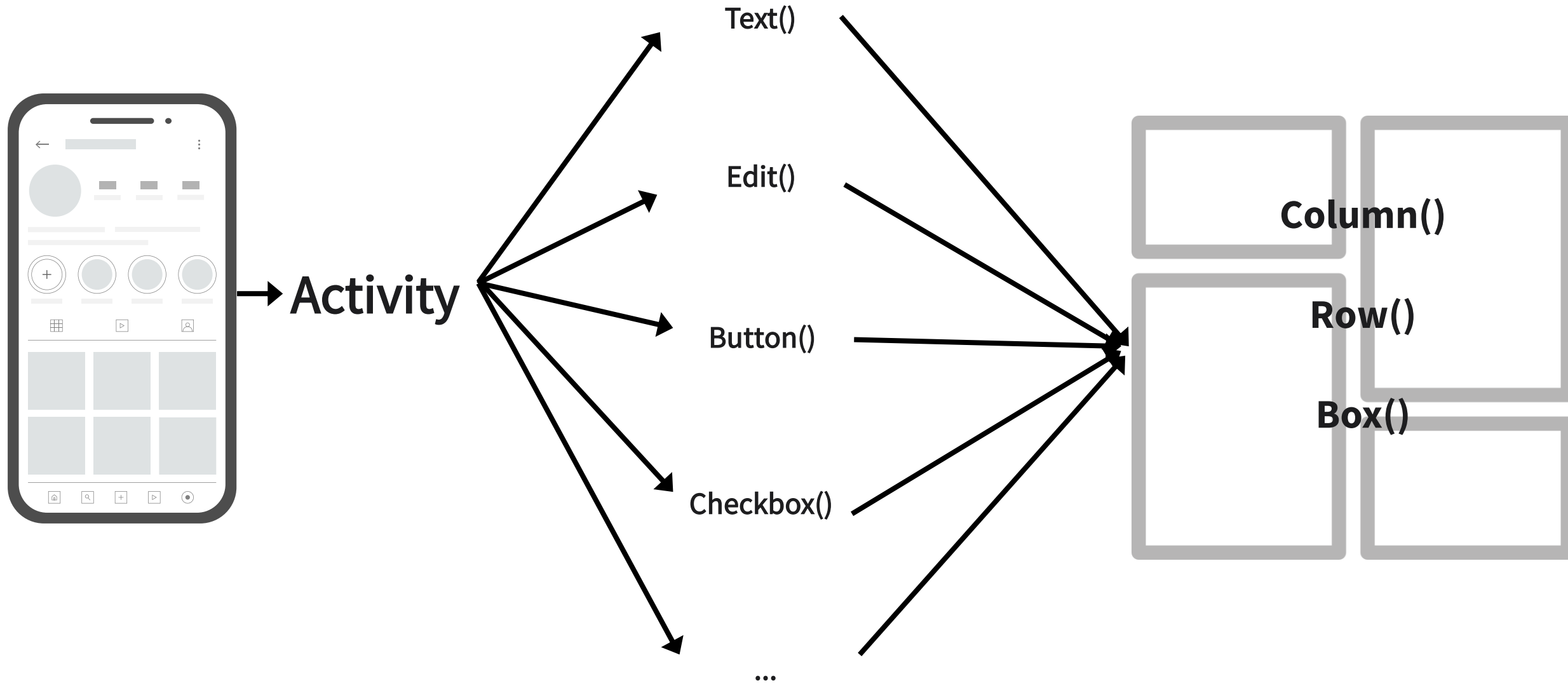


**Jede Activity durchläuft einen Lebenszyklus
und enthält Views und Layouts**

ACTIVITIES, LAYOUTS UND VIEWS



ACTIVITIES, LAYOUTS UND VIEWS



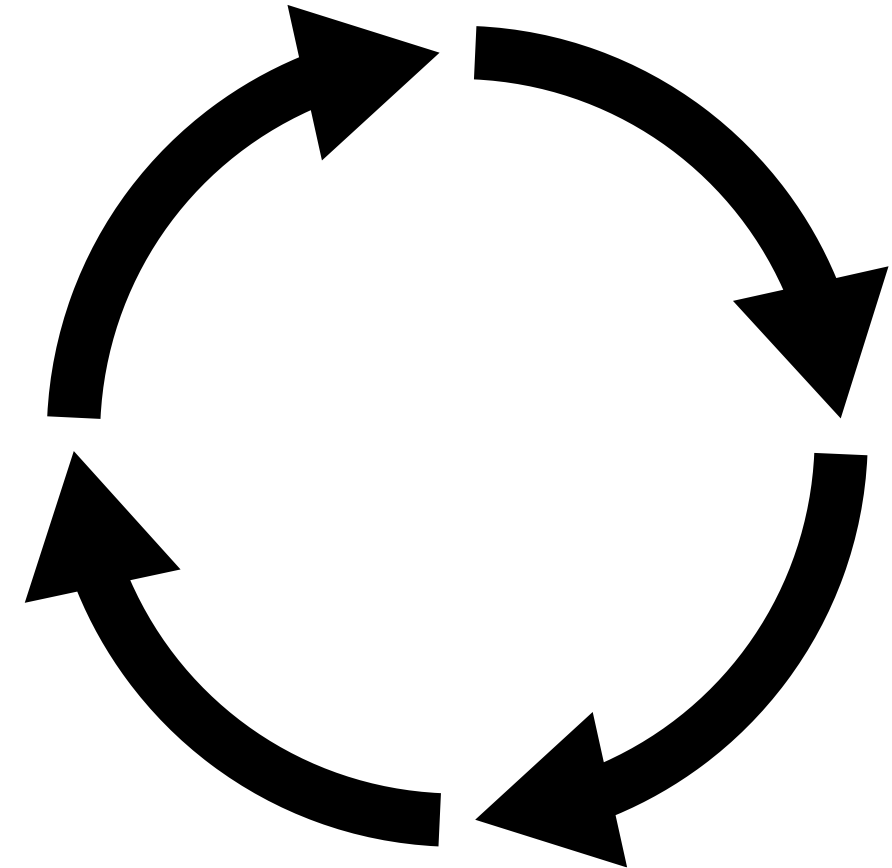
Lebenszyklus. Der Lebenszyklus einer Activity ist notwendig, da mobile Geräte nur begrenzte Ressourcen haben, insbesondere Hauptspeicher. Das Android-System überwacht den Ressourcenverbrauch und greift ein, um Prozesse mit niedriger Priorität zu beenden, wenn Engpässe drohen.

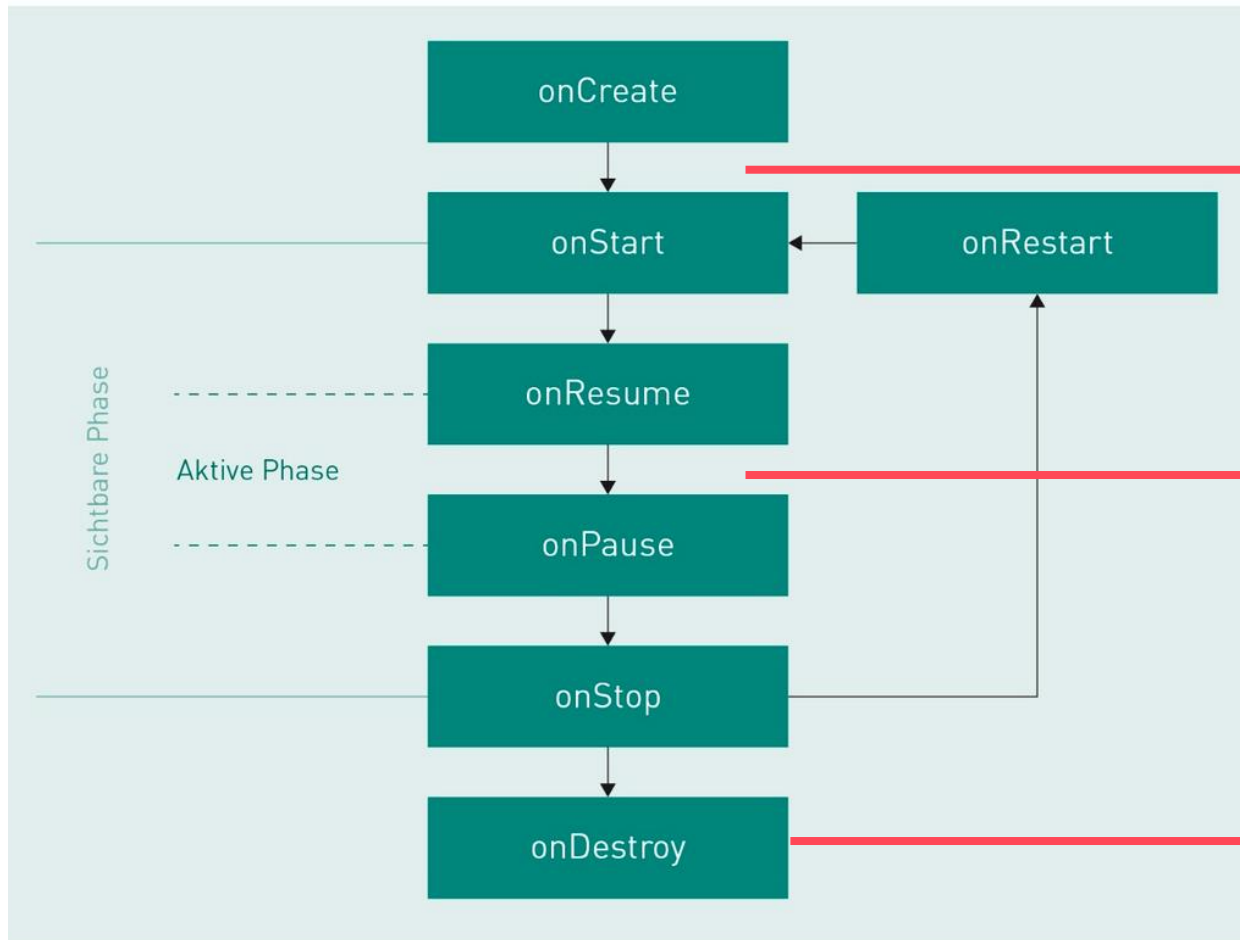
Prioritäten von Prozessen:

Die Priorität eines Prozesses wird durch die Sichtbarkeit und Interaktion mit dem Nutzer bestimmt:

- Höchste Priorität: Prozess führt eine Activity im Vordergrund aus.
- Mittlere Priorität: Activity ist sichtbar, aber durch eine andere verdeckt.
- Niedrigste Priorität: Activity ist nicht sichtbar.

Zustandswechsel. Apps können vom Android-System in verschiedene Zustände versetzt werden. Entwickler können diese Zustandsänderungen nicht verhindern, aber Callback-Methoden nutzen, um darauf zu reagieren (z.B. `onCreate()` beim Start).





Sichtbare Phase

Zwischen `onStart()` (bzw. `onRestart()`) und `onStop()` ist die Activity sichtbar, aber möglicherweise durch andere Activities verdeckt. Der Benutzer kann sie nicht direkt steuern.

Interaktive Phase

Zwischen `onResume()` und `onPause()` kann der Benutzer aktiv mit der App interagieren, indem er UI-Komponenten nutzt.

Inaktive Phase

Die Activity ist nicht sichtbar und befindet sich im Hintergrund.

ACTIVITIES, LAYOUTS UND VIEWS

Intents. Intents sind Nachrichten, die innerhalb des Android-Systems verschickt werden. Sie können entweder innerhalb einer App oder zwischen verschiedenen Apps gesendet werden.

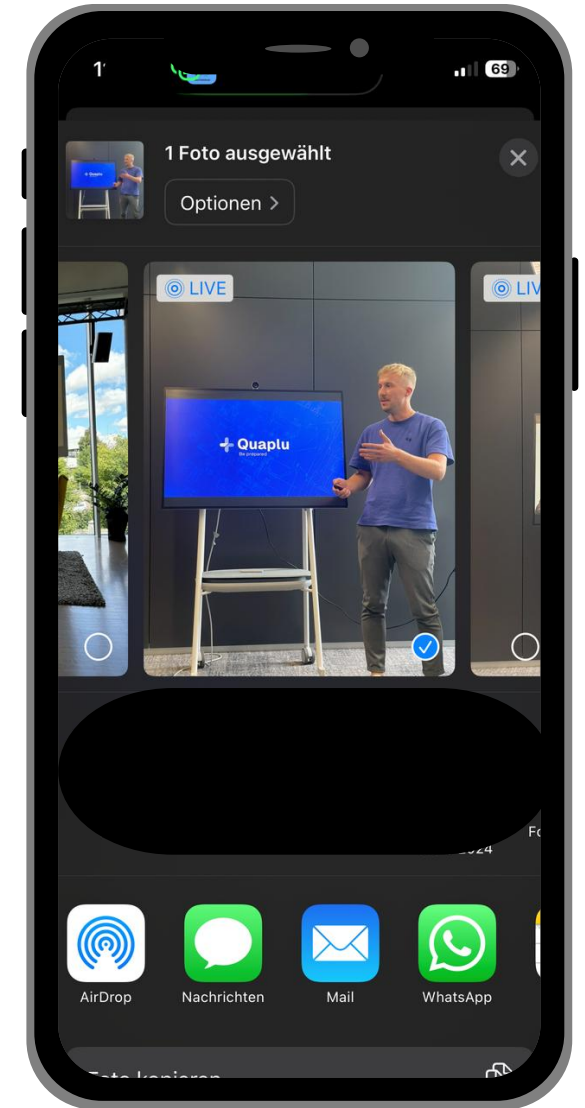
Arten von Intents

Explizite Intents: Eine Nachricht wird an eine spezifische Komponente innerhalb derselben App geschickt, z.B. das Starten einer neuen Activity.

Implizite Intents: Das Android-System leitet die Nachricht an eine passende App weiter, die eine bestimmte Aktion ausführen kann.

Beispiel aus dem Alltag. Das Teilen eines Fotos aus der Galerie: Ein Intent wird gesendet, und das System bietet Apps wie WhatsApp oder Instagram an, um das Foto zu teilen. Das Öffnen eines Links in einem Browser: Eine App sendet einen Intent, und das System lässt den Nutzer zwischen installierten Browsern wählen.

Broadcast Receiver stellen die Funktionalität zur Verfügung, auf implizite Intents zu reagieren.



ACTIVITIES, LAYOUTS UND VIEWS

Content Provider. Ein Content Provider ermöglicht den Zugriff auf Daten, die von einer App gespeichert werden, und stellt diese für andere Apps bereit. Typischerweise handelt es sich dabei um Daten, die lokal auf dem Gerät gespeichert sind, z.B. Kontakte oder Mediendateien.

Anwendungsbeispiel. Die Kontakte-App eines Smartphones stellt die gespeicherten Kontaktinformationen über einen Content Provider zur Verfügung. Andere Apps (z.B. Messenger) können – bei entsprechender Berechtigung – auf diese Daten zugreifen.

Datenquelle. Häufig liegt hinter einem Content Provider eine SQLite-Datenbank, die als lokale Datenspeicherung fungiert.

ACTIVITIES, LAYOUTS UND VIEWS

Services. Services sind Hintergrundprozesse, die keine eigene Benutzeroberfläche haben. Sie führen längere Operationen aus, die nicht unmittelbar mit der Benutzerinteraktion verbunden sind.

Beispiele für Services

- **Upload eines Fotos:** Der Upload eines Fotos wird in der UI gestartet, läuft dann aber im Hintergrund, während der Nutzer andere Aktivitäten ausführt.
- **E-Mail-Check:** Ein Service überprüft regelmäßig ein E-Mail-Konto auf neue Nachrichten, ohne dass eine Benutzerinteraktion erforderlich ist. Bei neuen E-Mails wird der Nutzer benachrichtigt.

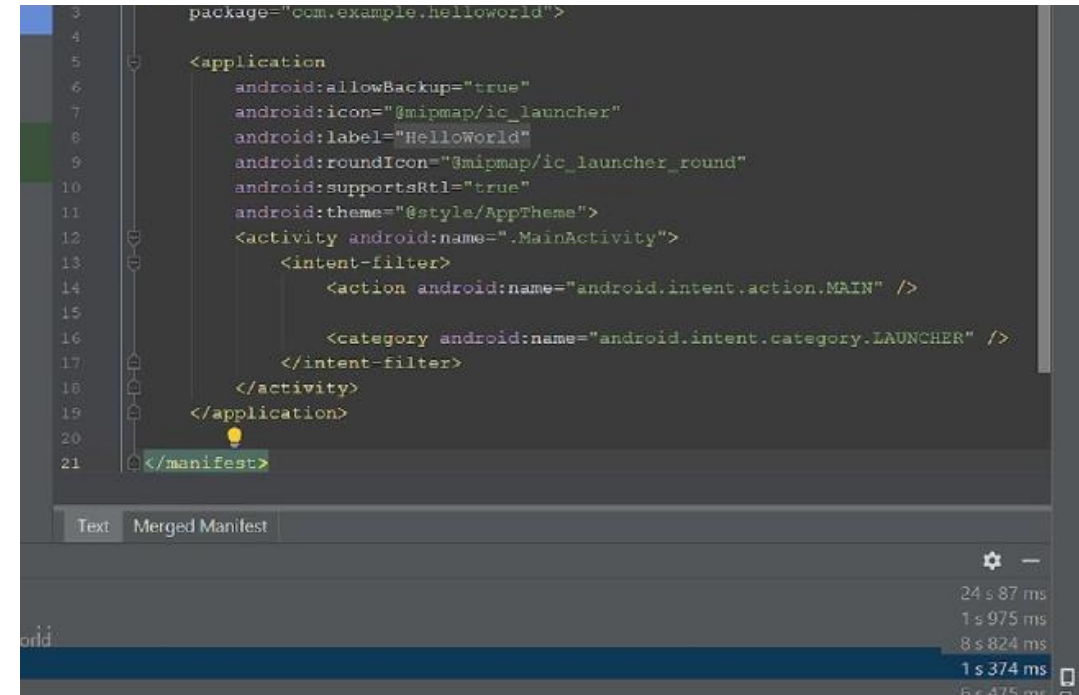
Funktion. Services ermöglichen es, Aufgaben im Hintergrund auszuführen, ohne die laufende UI oder andere Activities zu unterbrechen.

ACTIVITIES, LAYOUTS UND VIEWS

Android Manifest. Die AndroidManifest.xml ist die zentrale Konfigurationsdatei jeder Android-App. Sie enthält Informationen darüber, welche Komponenten zur App gehören und welche Activity gestartet wird sowie Berechtigungen die vom Nutzer erteilt werden können.

Komponenten. Im <application>-Element werden alle wichtigen App-Komponenten wie Activities, Services und Receiver definiert. Jede App benötigt mindestens eine Activity, die als Startpunkt fungiert.

Intent-Filter. Das <intent-filter>-Element bestimmt, welche Intents von einer Activity oder anderen Komponenten empfangen werden können. Zum Beispiel definiert es, welche Activity durch einen Start-Intent beim Öffnen der App gestartet wird.



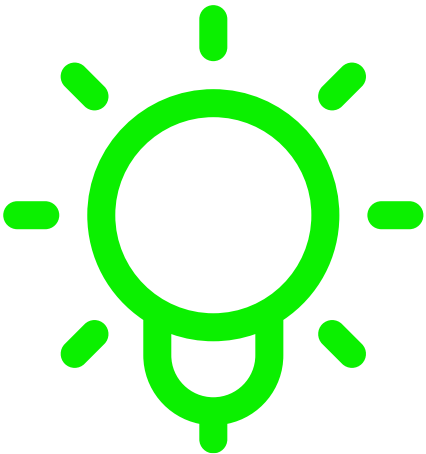
```
1 package="com.example.helloworld">
2
3 <application
4     android:allowBackup="true"
5     android:icon="@mipmap/ic_launcher"
6     android:label="@string/app_name"
7     android:roundIcon="@mipmap/ic_launcher_round"
8     android:supportRtl="true"
9     android:theme="@style/AppTheme">
10     <activity android:name=".MainActivity">
11         <intent-filter>
12             <action android:name="android.intent.action.MAIN" />
13
14             <category android:name="android.intent.category.LAUNCHER" />
15         </intent-filter>
16     </activity>
17 </application>
18
19 </manifest>
```

Text Merged Manifest

24 s 87 ms
1 s 975 ms
8 s 824 ms
1 s 374 ms
6 s 475 ms

Reminder:

- Eine Android App besteht aus einer losen Kopplung von einzelnen Komponenten.
- einer Activity werden gemäß den Vorgaben eines Layouts diverse Einzelelemente auf dem Bildschirm angezeigt.
- Intents sind Nachrichten, die über das Android System zu anderen Activities oder auch zu anderen Apps verschickt werden können.

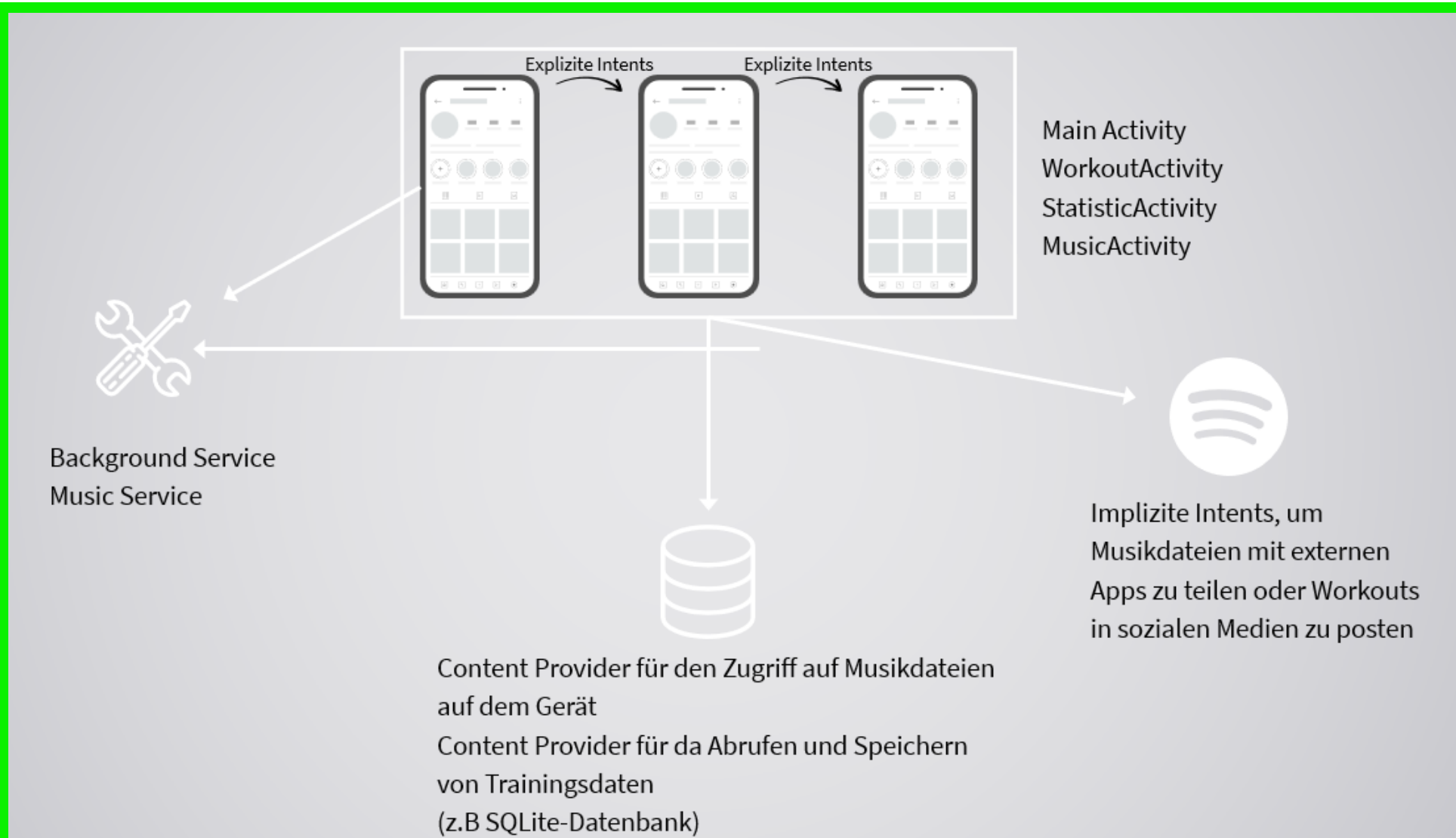


Aufgabe 1 – 45 Minuten

Szenario: Du bist Teil eines Entwicklungsteams, das eine Fitness-App entwickeln soll. Diese App ermöglicht es Nutzern, ihre täglichen Aktivitäten zu protokollieren, Workouts zu planen, und Fortschritte anhand von Statistiken zu verfolgen. Zusätzlich bietet die App eine Funktion, um Musik während des Trainings abzuspielen und Fortschritte mit Freunden zu teilen.

Aufgabenstellung: Erstelle ein Architekturdiagramm, das die wichtigsten Kernkomponenten einer Android-App darstellt, die für diese Fitness-App notwendig sind. Verwende folgende Komponenten und füge bei Bedarf weitere hinzu: Activities, Services, Content Provider, Broadcast Receiver und Intents.

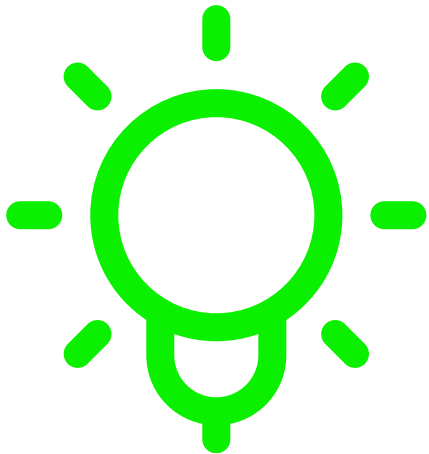
45 Min.



AUFGABE: EINSTIEG IN JETPACK COMPOSE MIT ANDROID-KOMPONENTEN

Ziel:

Entwicklung einer einfachen Notiz-App zur Veranschaulichung moderner UI-Entwicklung mit Jetpack Compose und dem Einsatz klassischer Android-Komponenten wie Activities, Intents und dem Android Manifest.



Aufgabenbeschreibung:

1. Erstellen Sie ein neues Android-Projekt mit der Vorlage Empty Compose Activity in Android Studio.
2. Entwickeln Sie eine Composable UI mit folgenden Elementen:
 - Ein TextField zur Eingabe einer Notiz
 - Ein Button, um die Notiz abzusenden
3. Implementieren Sie eine zweite Activity (SecondActivity) zur Anzeige der Notiz.
4. Verwenden Sie einen expliziten Intent, um beim Klick auf den Button die eingegebene Notiz an SecondActivity zu übergeben.
5. Registrieren Sie beide Activities korrekt in der Datei AndroidManifest.xml.
6. Zeigen Sie die empfangene Notiz in der zweiten Activity mithilfe von Compose an (Text()).
7. Dokumentieren Sie im Team, wie Sie die Komponenten miteinander verknüpft haben und welche Rolle sie jeweils spielen.

Dauer: 75 Minuten – Vergleich zur Pause

04

KERNKOMPONENTEN EINER ANDROID-APP

RESSOURCEN

Reminder:

- Eine Android App besteht aus einer losen Kopplung von einzelnen Komponenten.
- einer Activity werden gemäß den Vorgaben eines Layouts diverse Einzelelemente auf dem Bildschirm angezeigt.
- Intents sind Nachrichten, die über das Android System zu anderen Activities oder auch zu anderen Apps verschickt werden können.

KERNKOMPONENTEN EINER ANDROID-APP


ZUSAMMENFASSUNG IN EINER APP

COMPOSABLE FUNCTIONS

Eine Composable-Funktion ist eine moderne Art der UI-Entwicklung in Android. Es handelt sich um eine Kotlin-Funktion, die mit **@Composable** annotiert ist. Diese Annotation teilt dem Compose-Compiler mit, dass die Funktion für die UI-Konstruktion gedacht ist.

Was sind Built-in Composable Functions?

- Text (Anzeige von Text)
- Button (Interaktive Buttons)
- Image (Anzeige von Bildern)
- TextField (Eingabefelder)
- Card (Container mit Schatten & Design)
- LazyColumn / LazyRow (Listenansicht)



```
@Composable
private fun FirstComposable() {
    Text(text: "Ich bin auch eine Composable Funktion")
}
```

Anwendungsfall:

- Anzeigen von statischem oder dynamischem Text
- Formatierung durch style-Parameter

```
@Composable
private fun MyApp() {
    Text(
        text = "Hallo, Jetpack Compose!",
        fontSize = 20.sp,
        fontWeight = FontWeight.Bold,
        color = Color.Blue
    )
}
```

Anwendungsfall:

- Benutzereingaben ermöglichen
- für jede Form von Formularen z.B Login oder Registrierung

```
@Composable
private fun MyApp() {
    var text by remember { mutableStateOf( value: "" ) }
    TextField(
        value = text,
        onChange = { text = it },
        label = { Text( text: "Gib etwas ein" ) }
    )
}
```

Anwendungsfall:

- Anzeigen von Bildern aus Ressourcen oder URLs

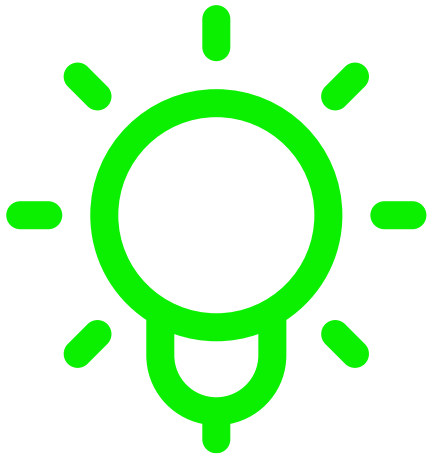
```
@Composable
private fun MyApp() {
    Image(
        painter = painterResource(R.drawable.sample_image)
        contentDescription = "Beispielbild",
        modifier = Modifier.size(100.dp)
    )
}
```

04

KERNKOMPONENTEN EINER ANDROID-APP

GRAFISCHE GESTALTUNG





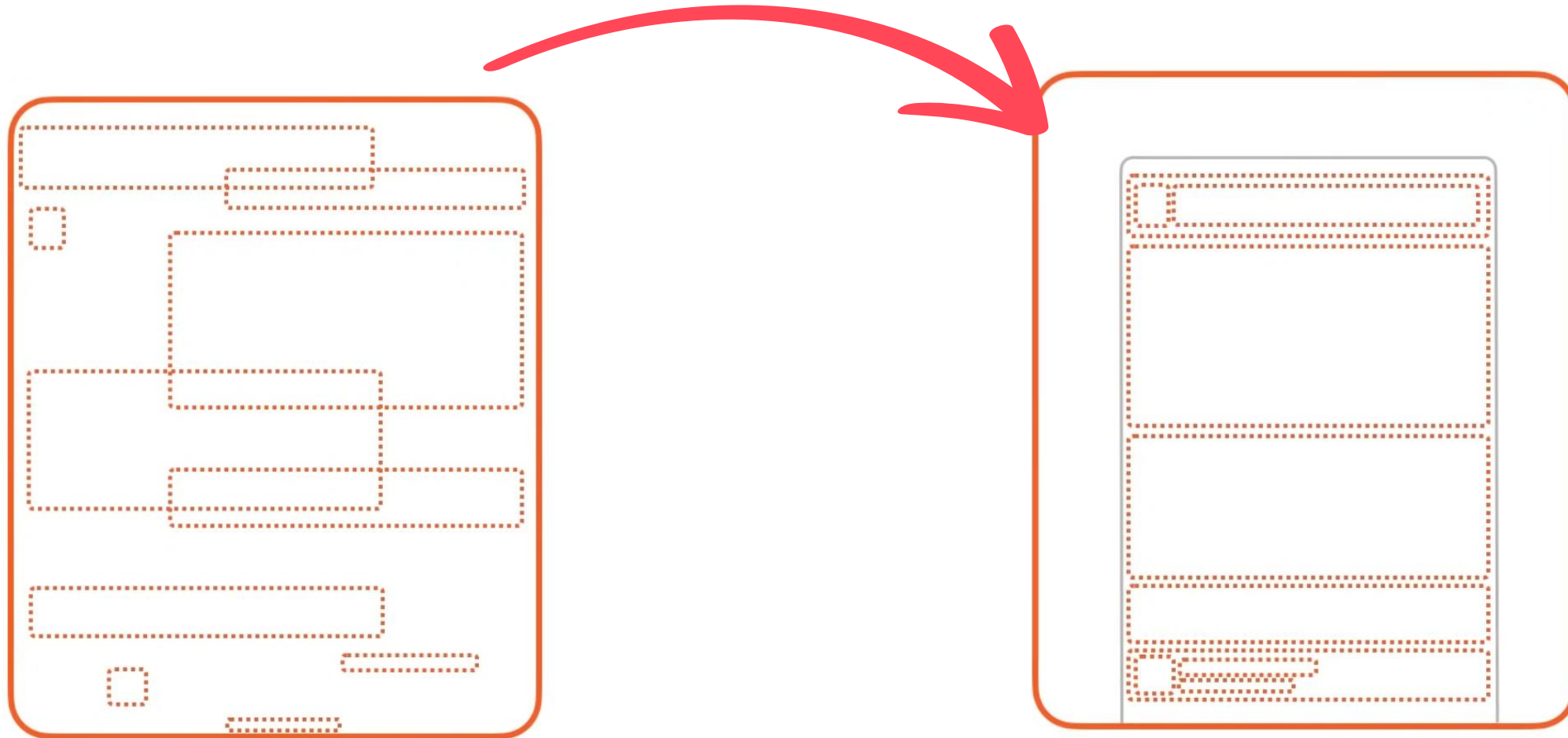
Recherche:

- Wie können GUI für Android-Apps einfacher entworfen werden?
- Sind euch hierfür Tools bekannt (Figma, UXPin, Android Studio Preview, FlutterFlow?)?

→ Recherchiert ggf. passende Tools und erstellt ein kleines Beispiel eurer Wahl um uns einen kleinen Einblick in das Tool zu geben.

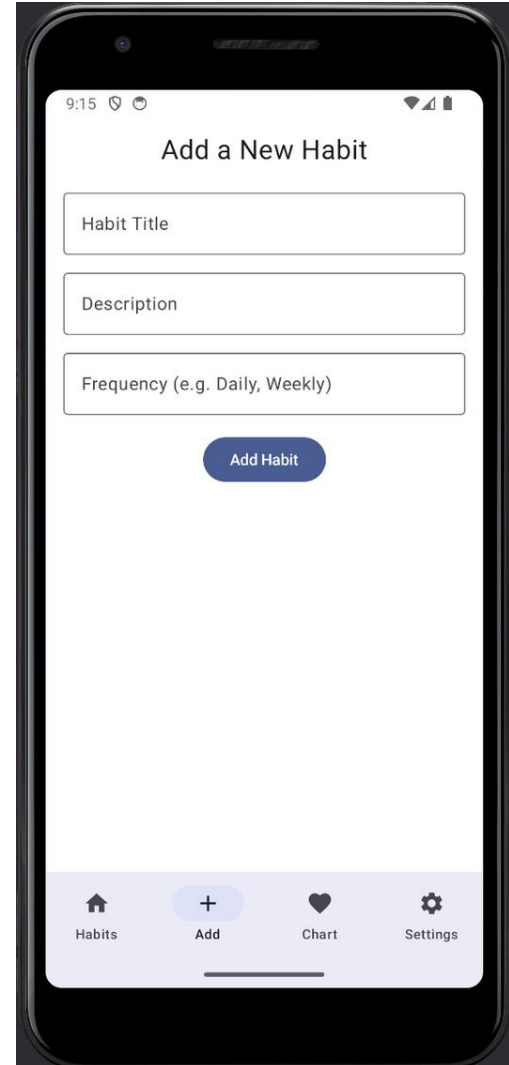
40 Minuten.

DIE GRAFISCHE GESTALTUNG



DIE GRAFISCHE GESTALTUNG

ROW()



COLUMN()

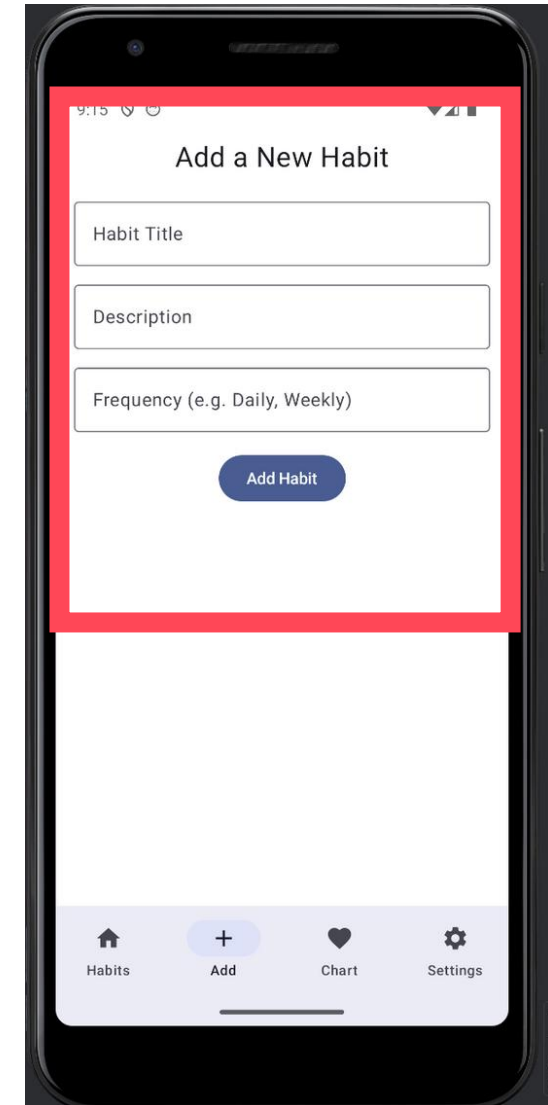


DIE GRAFISCHE GESTALTUNG

Das **Column()-Layout** ist ideal für die vertikale Stapelung von Elementen. Es wird üblicherweise für die Erstellung vertikaler Listen von Elementen oder Abschnitten in einer Benutzeroberfläche verwendet.

verticalAlignment \longrightarrow Ausrichtung innerhalb der Zeile

horizontalArrangement \longrightarrow Steuerung der Abstände

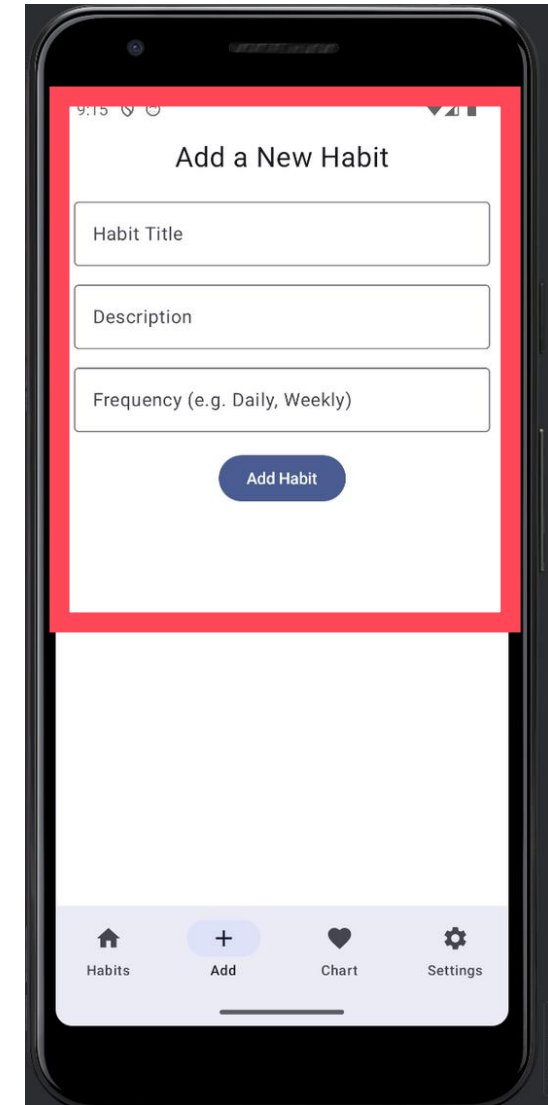


DIE GRAFISCHE GESTALTUNG

Das **Row()-Layout** für die horizontale Anordnung von Elementen verwendet. Dies ist nützlich, um horizontale Listen zu erstellen oder Elemente nebeneinander auszurichten.

horizontalArrangement → Steuerung der Abstände

verticalAlignment → Ausrichtung innerhalb der Zeile



Das Box-Layout BOX() ist ein vielseitiger Container, mit dem Elemente überlagert und gestapelt werden können. Es wird häufig für die Erstellung von benutzerdefinierten UI-Komponenten oder Overlays verwendet.

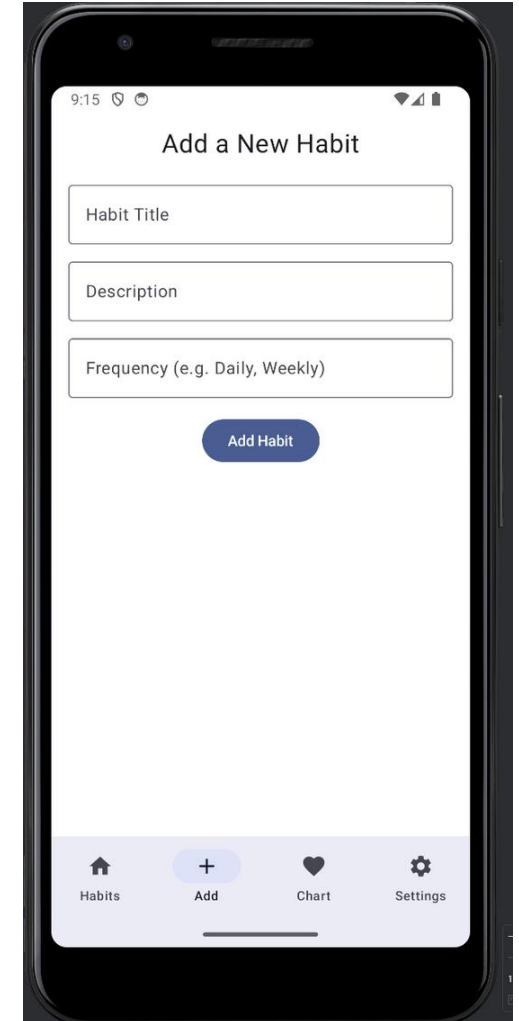
contentAlignment —→ Positionierung des Inhalts

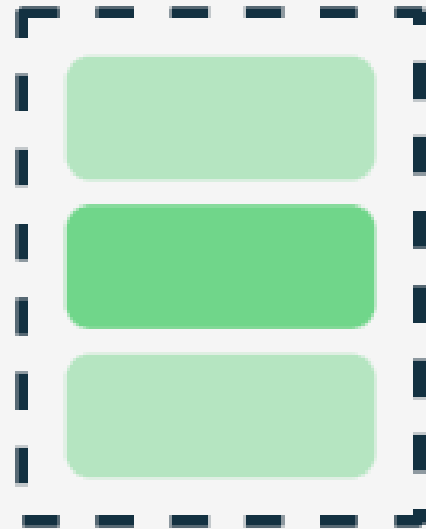


DIE GRAFISCHE GESTALTUNG

Das Scaffold-Layout ist ein übergeordnetes Layout, das eine Struktur für gängige UI-Elemente wie die App-Leiste, die Floating Action Buttons und mehr bietet.

- `AppBar` (Titelzeile)
- `BottomNavigation` (Navigation unten)
- `FloatingActionButton`





Column




Row



Box

```
@Composable
fun SearchResult() {
    Row {
        Image(
            // ...
        )
        Column {
            Text(
                // ...
            )
            Text(
                // ...
            )
        }
    }
}
```

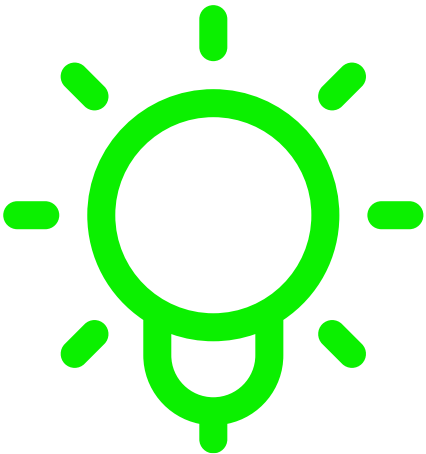
[LayoutBasicsSnippets.kt](#) 

DIE GRAFISCHE GESTALTUNG

Mit Modifikatoren können wir ein Composeable verzieren oder ergänzen. Mit Modifikatoren können wir Folgendes tun:

- Größe, Layout, Verhalten und Darstellung des Composeables ändern
- Informationen wie Labels für die Barrierefreiheit hinzufügen
- Nutzereingabe verarbeiten
- allgemeine Interaktionen z. B. dass ein Element anklickbar, scrollbar, verschiebbar oder zoombar ist.

```
@Composable
private fun Greeting(name: String) {
    Column(modifier = Modifier.padding(24.dp)) {
        Text(text = "Hello,")
        Text(text = name)
    }
}
```



Aufgabe 1: App-Grundstruktur mit Scaffold

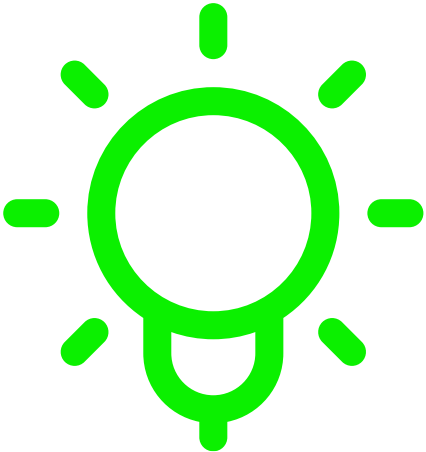
In dieser Aufgabe erstellst du die Grundstruktur:

- Verwende Scaffold mit einer TopAppBar, die als Titel "Profil-Galerie" anzeigt
- Bereite den Bereich für den Hauptinhalt vor, der später die Profilgalerie enthalten wird

Aufgabe 2: Willkommensbereich gestalten

Erstelle eine Begrüßungsanzeige im oberen Bereich der App:

- Zeige einen Text an, der "Willkommen zu Jetpack Compose!" sagt
- Verändere die Schriftgröße auf mindestens 20.sp und die Schriftfarbe nach deiner Wahl
- Erstelle darunter eine Row mit drei Text-Elementen ("Profile", "Favoriten", "Einstellungen")
- Die Row soll die volle Breite (fillMaxWidth()) nutzen
- Die Elemente sollen gleichmäßig verteilt sein (Arrangement.SpaceEvenly)



Aufgabe 3: Profilliste mit LazyColumn

Erstelle eine dynamische Liste mit LazyColumn:

- Erstelle eine Liste mit LazyColumn für fünf Namen (z.B. "Anna", "Ben", "Clara", "David", "Emma")
- Jedes Profil soll mit Card umrandet werden und etwas Padding haben
- Nutze modifier = Modifier.fillMaxWidth(), um die Liste über die gesamte Breite anzuzeigen

Aufgabe 4: Detaillierte Profilkarte

Für jedes Profil in der LazyColumn erstelle eine detaillierte Ansicht:

- Profilbild (Platzhalter: Modifier.background(Color.Gray)) mit einer festen Größe (100x100 dp) mit Name der Person als Text
- Füge einen Button hinzu, der die Anzeige ändert (Button-Text: "Folgen" → "Gefolgt" nach Klick)

05

INTERAKTION ZWISCHEN ANWENDUNGSKOMPONENTEN

WORUM GEHT ES IN DIESEM KAPITEL?

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

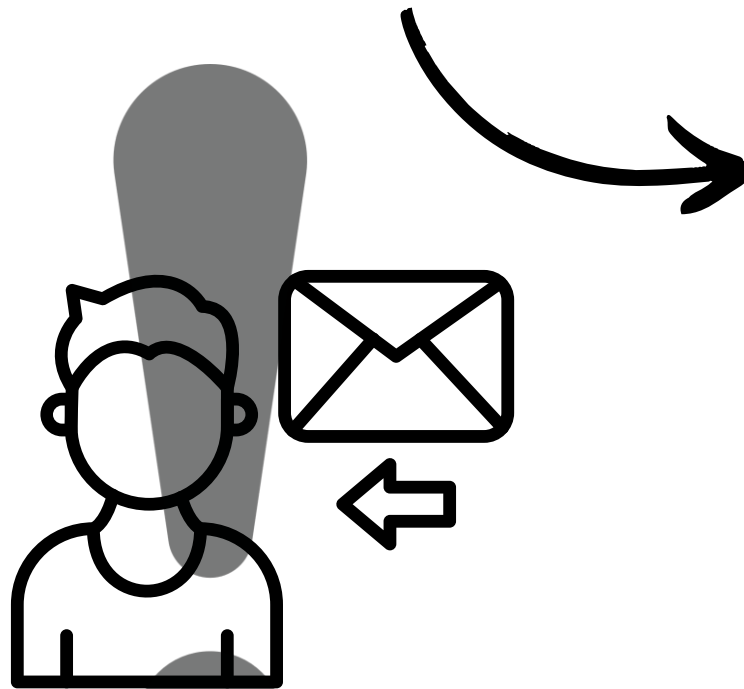
- aus welchen Bestandteilen ein Android-System besteht.
- welche dieser Komponenten für die Software-Entwicklung von Bedeutung sind.
- welche Netzwerk- und Kommunikationstechnologien für die mobile Android-Welt relevant sind.

05

INTERAKTION ZWISCHEN ANWENDUNGSKOMPONENTEN

INTENTS

Android ist komponentenbasiert aufgebaut, d. h., eine Android- Anwendung besteht aus einzelnen Komponenten, für die noch ein Weg zur Kommunikation untereinander gesucht wird.



Explizite

Intents

Diese Kommunikation läuft über Intents



Implizierte Intents

Aktivität starten

Dienst Starten

Broadcast senden

Was ist ein impliziter Intent?

- Ein impliziter Intent beschreibt eine allgemeine Aktion, die durchgeführt werden soll, ohne eine spezifische Komponente (z. B. eine bestimmte Aktivität) anzugeben.
- Das System sucht automatisch eine passende Komponente (Activity oder Service), die diese Aktion ausführen kann.
- Häufig verwendet, um Interaktionen zwischen Apps zu ermöglichen.



Anwendungsfälle:

- Öffnen eines Webbrowsers für eine bestimmte URL.
- Versenden einer Nachricht mit einer beliebigen App.
- Starten eines Anrufs.



Was ist ein impliziter Intent?

- Ein impliziter Intent beschreibt eine allgemeine Aktion, die durchgeführt werden soll, ohne eine spezifische Komponente (z. B. eine bestimmte Aktivität) anzugeben.
- Das System sucht automatisch eine passende Komponente (Activity oder Service), die diese Aktion ausführen kann.
- Häufig verwendet, um Interaktionen zwischen Apps zu ermöglichen.

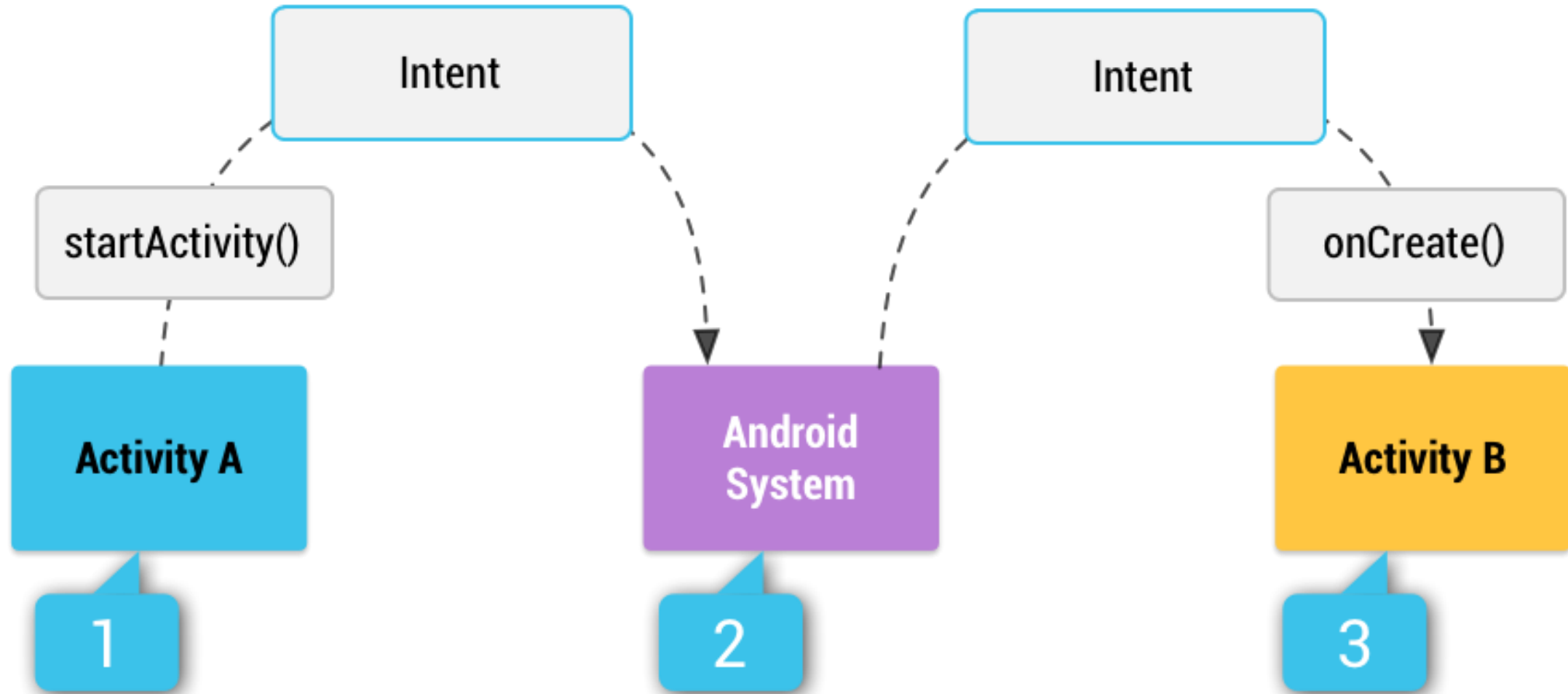


```
fun openWebLink(url: String) {  
    val intent = Intent(Intent.ACTION_VIEW).apply {  
        data = Uri.parse(url)  
    }  
    if (intent.resolveActivity(packageManager) != null) {  
        startActivity(intent)  
    }  
}
```

- Die Aktion `Intent.ACTION_VIEW` signalisiert, dass der Nutzer einen Inhalt (in diesem Fall eine URL) betrachten möchte.
- *data* wird auf die gewünschte URL gesetzt (`Uri.parse(url)`).
- Mit `resolveActivity(packageManager)` wird sichergestellt, dass es auf dem Gerät eine App gibt, die diesen Intent verarbeiten kann (z. B. einen Webbrowser).

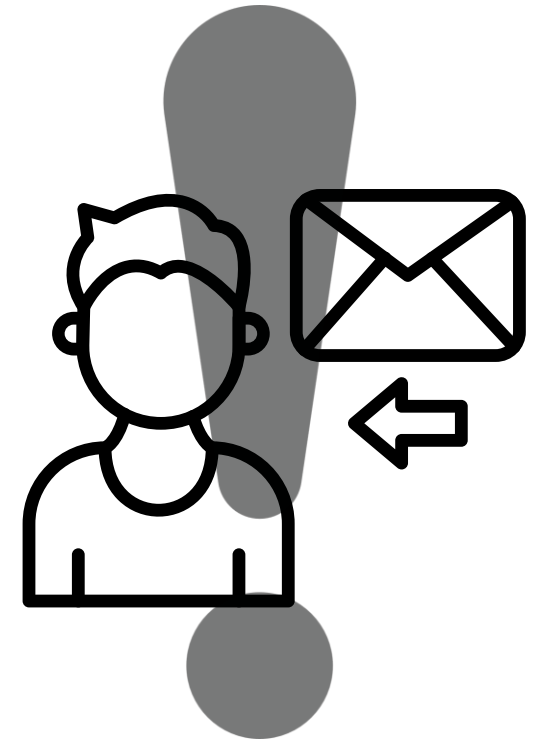


IMPLIZIERTER INTENT



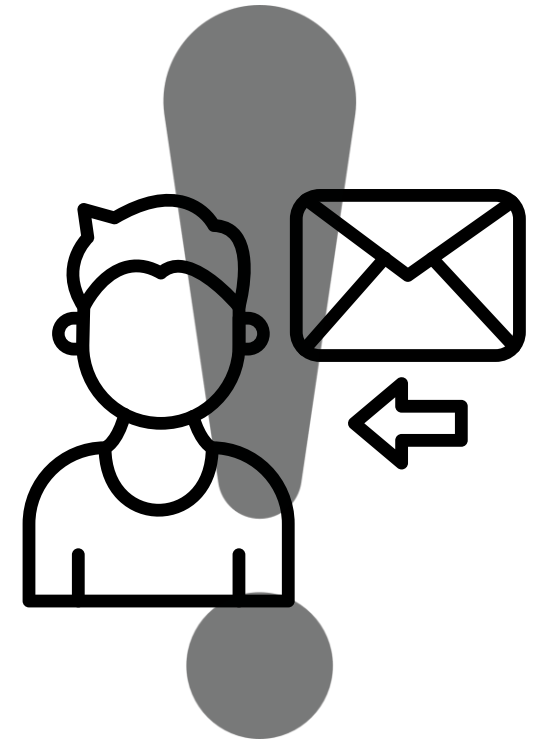
Was ist ein expliziter Intent?

- Ein expliziter Intent gibt eine genaue Komponente (z. B. eine Aktivität oder einen Service) an, die gestartet werden soll.
- Häufig innerhalb derselben App verwendet, um zwischen Aktivitäten zu wechseln.



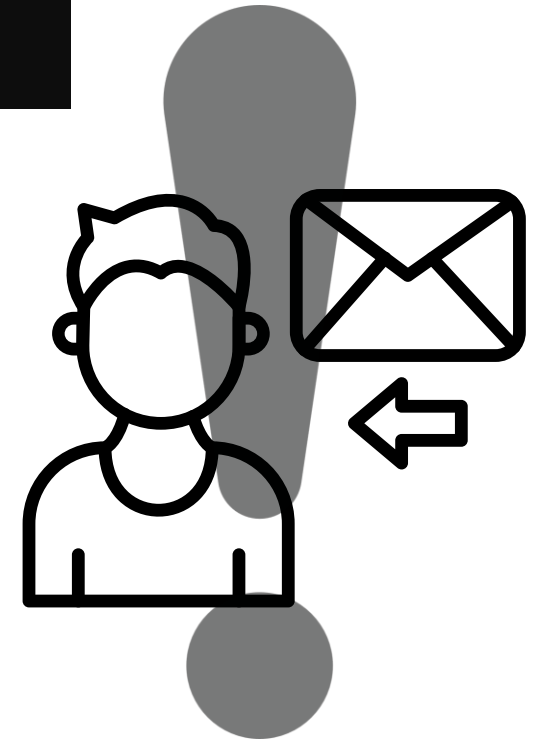
Anwendungsfälle:

- Starten einer bestimmten Aktivität innerhalb der App.
- Kommunikation zwischen Services und Komponenten innerhalb derselben App.

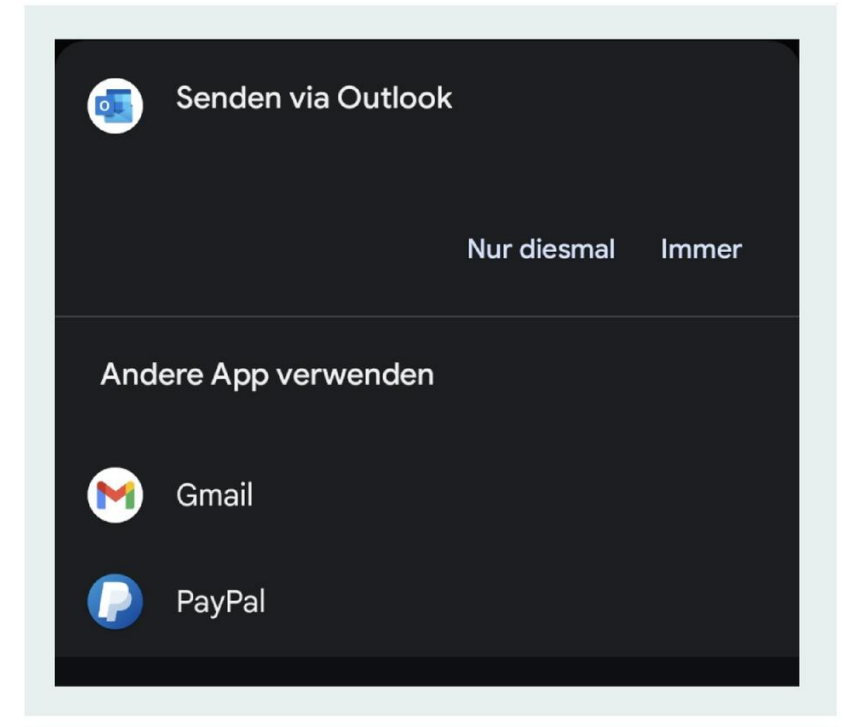


```
fun navigateToTargetActivity(context: Context, message: String) {  
    val intent = Intent(context, TargetActivity::class.java).apply {  
        putExtra("key", message)  
    }  
    context.startActivity(intent)  
}
```

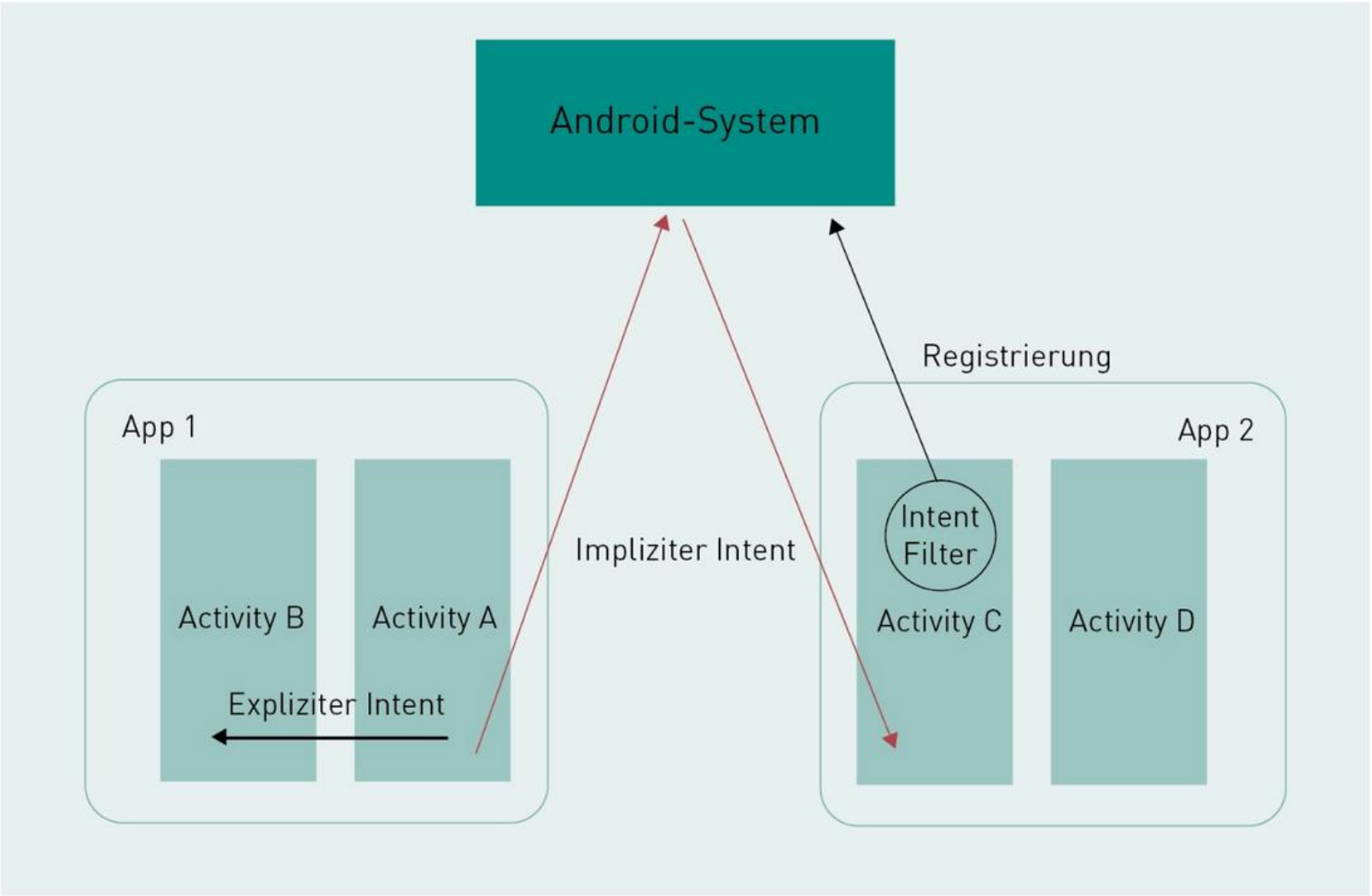
- Der Intent wird explizit für die TargetActivity erstellt, indem diese als Zielklasse angegeben wird (TargetActivity::class.java)
- Mit putExtra wird eine zusätzliche Information (message) an die Zielaktivität gesendet. Die Zielaktivität kann diese Daten über intent.getStringExtra("key") abrufen.
- Die Methode startActivity(intent) wird aufgerufen, um die TargetActivity zu starten



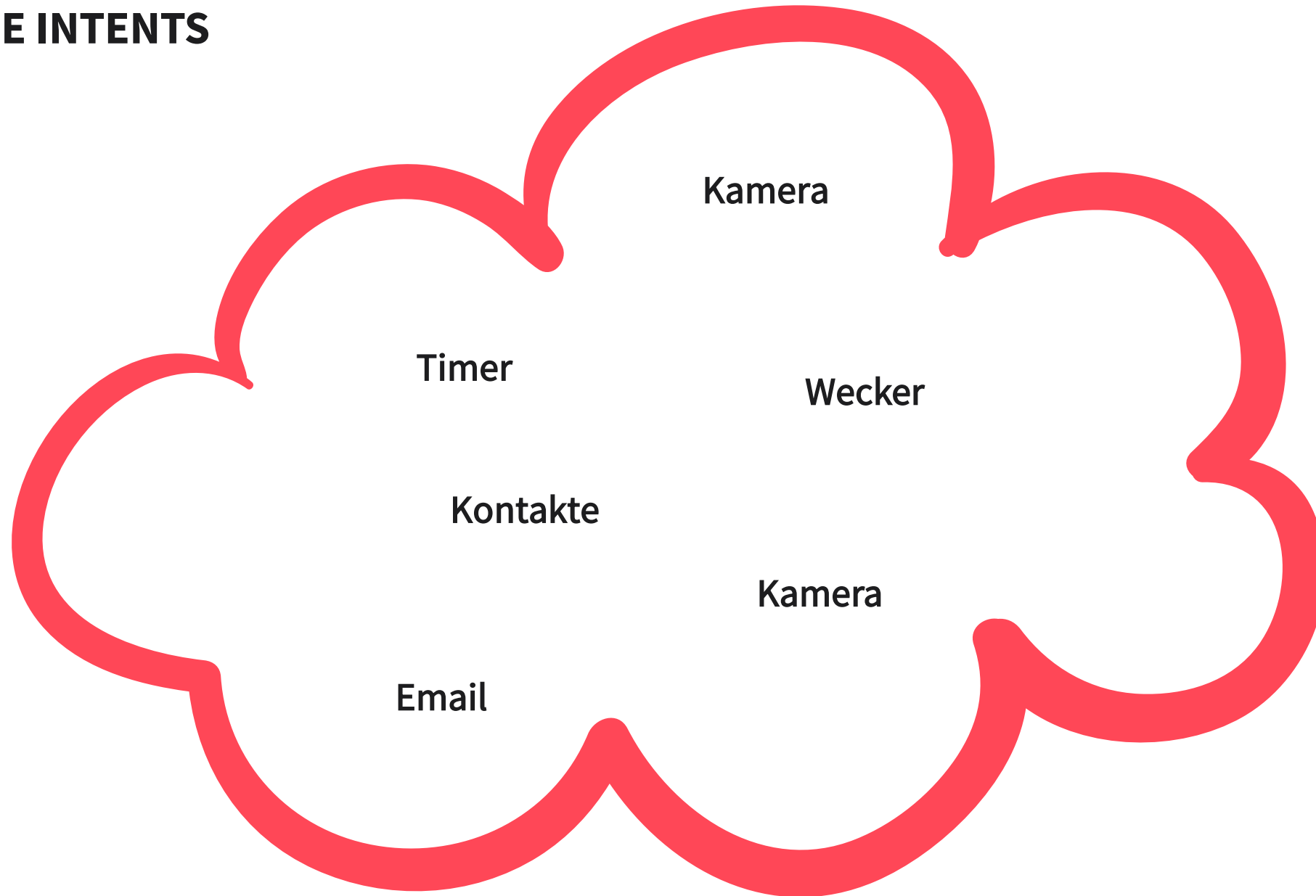
- Ein Intent-Filter ist ein Ausdruck in der Manifestdatei einer App, der den Typ der Intents angibt, die die Komponente empfangen soll.
- Wenn Sie beispielsweise einen Intent-Filter für eine Aktivität deklarieren, können andere Apps Ihre Aktivität direkt mit einer bestimmten Art von Intent starten.
- Wenn mehrere Intent-Filter kompatibel sind, wird ein Dialogfeld angezeigt, in dem der Nutzer auswählen kann, welche App verwendet werden soll.



Quelle: erstellt im Auftrag der IU, 2015.



HÄUFIGE INTENTS

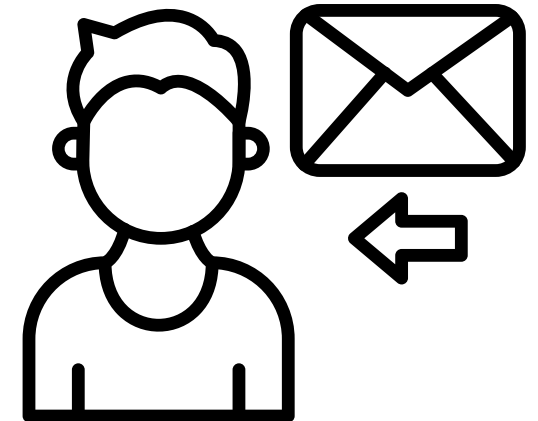


- Funktionskopf bestehend aus drei Parameter: message, hour und minutes
- Ein neuer Intent wird mit der Aktion `AlarmClock.ACTION_SET_ALARM` erstellt. Diese Aktion signalisiert dem Android-System, dass ein Alarm erstellt werden soll.
- innerhalb des apply-Block werden zusätzlich Daten an das Intent “gehängt”: Beschreibung des Alarms, Stunde des Alarms und die Minuten des Alarms

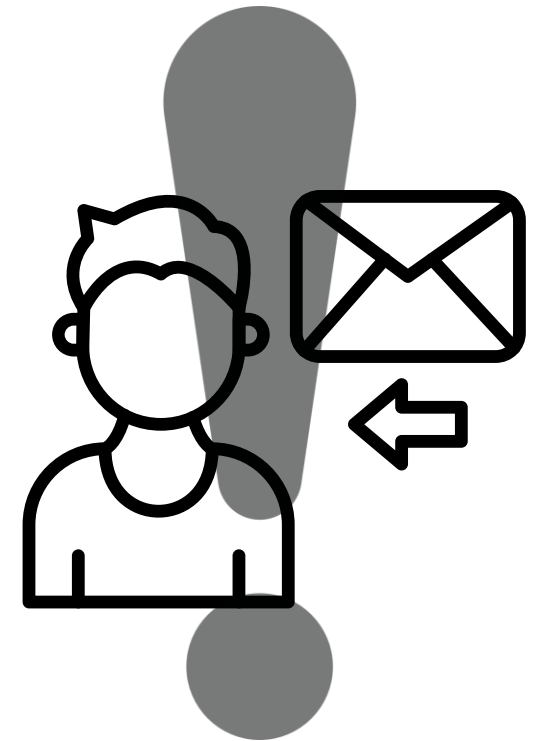
```
fun createAlarm(message: String, hour: Int, minutes: Int) {  
    val intent = Intent(AlarmClock.ACTION_SET_ALARM).apply {  
        putExtra(AlarmClock.EXTRA_MESSAGE, message)  
        putExtra(AlarmClock.EXTRA_HOUR, hour)  
        putExtra(AlarmClock.EXTRA_MINUTES, minutes)  
    }  
    if (intent.resolveActivity(packageManager) != null) {  
        startActivity(intent)  
    }  
}
```

REMEMBER

- Intents sind ein zentraler Mechanismus in Android, um Kommunikation zwischen Komponenten (z. B. Aktivitäten, Services) oder zwischen Apps zu ermöglichen.

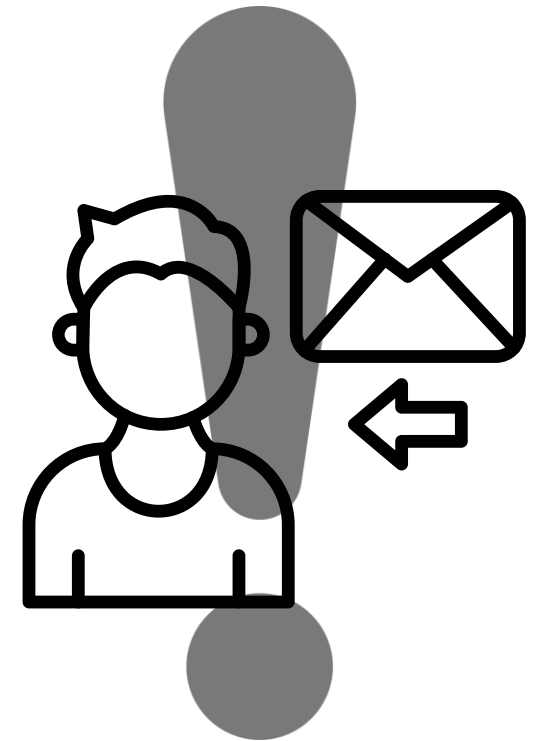


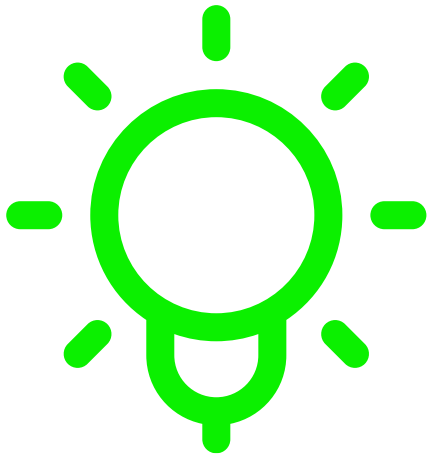
- Implizite Intents
 - Beschreiben eine allgemeine Aktion (z. B. "Webseite öffnen").
 - Das System wählt eine passende App aus.
 - Anwendungsfälle: Öffnen eines Browsers, Teilen von Inhalten, Starten eines Anrufs.



REMEMBER

- Explizite Intents
 - Geben eine spezifische Komponente an (z. B. "Start MainActivity").
 - Häufig innerhalb derselben App verwendet.
 - Anwendungsfälle: Navigation zwischen Aktivitäten oder Kommunikation mit Services.





- Recherchiere und beschreibe mindestens **drei** verschiedene vordefinierte Intent-Aktionen und ihre typischen Anwendungsfälle.
- Beantworte dabei folgende Fragen:
 - Was macht diese Aktion?
 - Welche zusätzlichen Daten (Extras) werden häufig benötigt?
 - Gibt es bestimmte Kategorien (z. B. Intent.CATEGORY_DEFAULT), die oft mit dieser Aktion verwendet werden?

20 Minuten

05

INTERAKTION ZWISCHEN ANWENDUNGSKOMPONENTEN SERVICES UND BROADCAST RECEIVE

WARUM SERVICES?

- Activities bieten eine Benutzerschnittstelle, die vollständig sichtbar ist und interaktiv genutzt werden kann. Allerdings sind nicht alle Funktionen auf eine solche sichtbare UI angewiesen oder sinnvoll, insbesondere bei langlaufenden Operationen wie:
 - Hochladen von Bildern
 - Synchronisierung von Daten
 - Musikkwiedergabe im Hintergrund
- Diese Aktionen können effizient im Hintergrund ausgeführt werden, sodass der Benutzer gleichzeitig andere Tasks in der App durchführen kann.



- Services sind spezielle Komponenten in Android, die Hintergrundoperationen ausführen.
- Sie benötigen keine sichtbare Benutzerschnittstelle.
- Typische Interaktionen:
 - Starten der Aktion (z. B. über einen Button in der UI).
 - Abschlussmeldung (z. B. über eine Benachrichtigung).



- Android bietet verschiedene Arten von Services, die sich in ihren Eigenschaften und Verhalten unterscheiden:
 1. Prozesszugehörigkeit: Läuft der Service im gleichen Prozess wie die Anwendung, oder wird ein eigener Prozess gestartet?
 2. Ressourcenmanagement: Was passiert mit dem Service, wenn Ressourcen knapp werden? Kann er beendet oder geschützt werden?



BROADCAST INTENTS

- Systemnachrichten, die vom Android-System gesendet werden, um Anwendungen über Änderungen im Systemzustand zu informieren werden **Broadcast Intents** genannt.
- Beispiele:
 - Geringer Akkustand
 - Installation einer neuen App
 - Bevorstehendes Herunterfahren des Systems



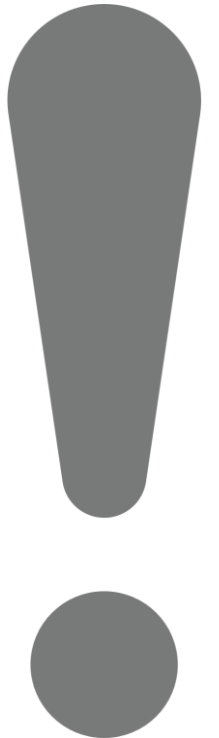
- Android Anwendungen können auf solche Systemnachrichten über sogenannte

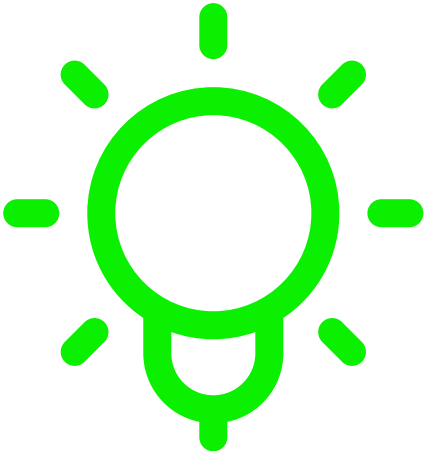
Broadcast Receiver reagieren:

- Akkuwarnung: Energiesparmodus aktivieren
- System herunterfahren: App-Zustand sichern

Statische Broadcast Receiver	Dynamische Broadcast Receiver
<ul style="list-style-type: none">• Registrierung im Android-Manifest• Läuft unabhängig von der App.• Kann Systemnachrichten empfangen, auch wenn die App nicht aktiv ist.	<ul style="list-style-type: none">• Registrierung während der Laufzeit in einer Komponente• Nur aktiv, während die registrierende Komponente läuft.• Registrierung und Abmeldung erfolgen programmatisch.

- **Intents als Kommunikationsmechanismus**
 - Intents ermöglichen die lose gekoppelte Kommunikation zwischen Komponenten in Android.
 - Sie dienen als "Absichtserklärung", um eine bestimmte Aktion von einer anderen Komponente auszuführen.
- **Broadcast Intents**
 - Eine spezielle Art von Intents, mit denen Systemnachrichten an mehrere registrierte Empfänger (Broadcast Receiver) gesendet werden.
- **Services für Hintergrundoperationen**
 - Langlaufende Prozesse ohne UI-Schnittstelle (z. B. Hochladen von Bildern oder Systemaktualisierungen) sollten in Services ausgeführt werden.
 - Services sorgen dafür, dass UI-interaktive Tasks und Hintergrundprozesse getrennt bleiben.





- Recherchiere die drei Haupttypen von Services in Android und beantworte folgende Fragen für jeden Service-Typ:
 - Was ist die Hauptfunktion dieses Service-Typs?
 - In welchen Szenarien wird dieser Service-Typ typischerweise verwendet?
 - Wie unterscheidet sich der Lebenszyklus dieses Services von den anderen Typen?
- Die drei zu untersuchenden Service-Typen sind:
 - Started Service
 - Bound Service
 - Foreground Service

06

FORTGESCHRITTENE TECHNIKEN

WORUM GEHT ES IN DIESEM KAPITEL?

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- aus welchen Bestandteilen ein Android-System besteht.
- welche dieser Komponenten für die Software-Entwicklung von Bedeutung sind.
- welche Netzwerk- und Kommunikationstechnologien für die mobile Android-Welt relevant sind.

06

FORTGESCHRITTENE TECHNIKEN

ARCHITEKTUR

Was ist MVVM (Model-View-ViewModel)?

- MVVM ist ein Architektur-Muster, das vor allem in der Entwicklung von Benutzeroberflächen, z. B. in Android, eingesetzt wird.
- Ziel: Trennung der Benutzeroberfläche (View) von der Geschäftslogik (ViewModel) und den Daten (Model).

Ziele des MVVM-Musters

- Trennung der Verantwortlichkeiten:
 - UI, Logik und Daten bleiben klar voneinander getrennt.
- Testbarkeit:
 - Business-Logik und Daten können unabhängig von der UI getestet werden.
- Separation of Concerns (SoC):
 - Jede Schicht ist für einen bestimmten Zweck zuständig.

Die Rollen im MVVM-Muster

Model:

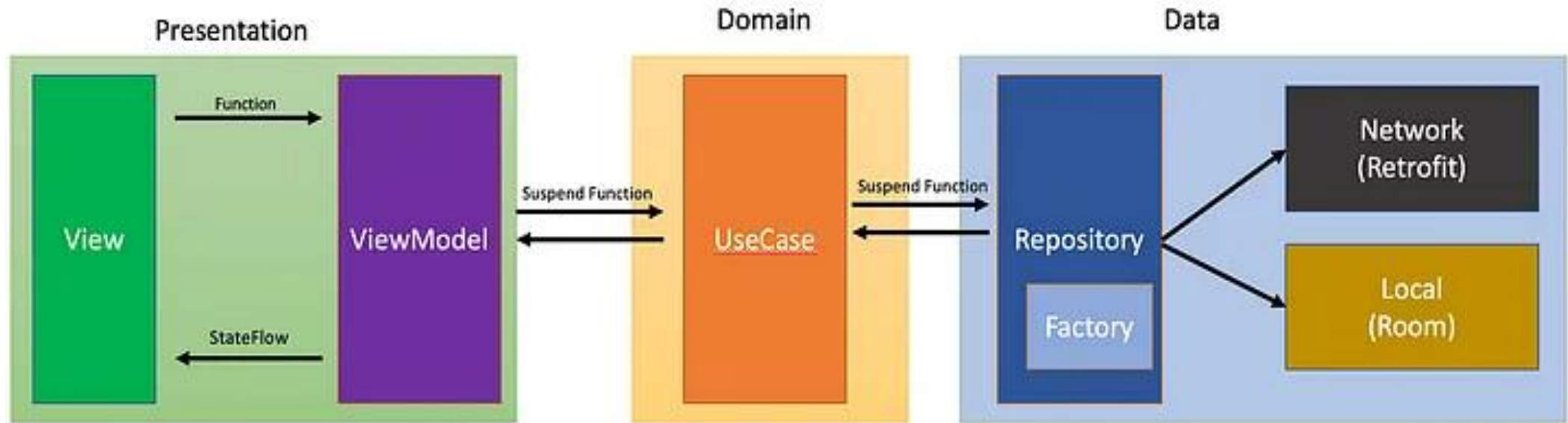
- Verwalten von Daten und Geschäftslogik
- Liefert Daten an das ViewModel

ViewModel:

- Vermittler zwischen Model und View
- Bereitet Daten auf, die die View benötigt
- Enthält keine Informationen zur Benutzeroberfläche

View

- Präsentation der Benutzeroberfläche
- Bindet sich an die Daten und Befehle des ViewModels
- Reagiert auf Benutzerinteraktionen



MVVM - PRESENTATION

Die Presentation Layer bildet die Schnittstelle zwischen dem Benutzer und der Anwendung.

Benutzereingaben verarbeiten:

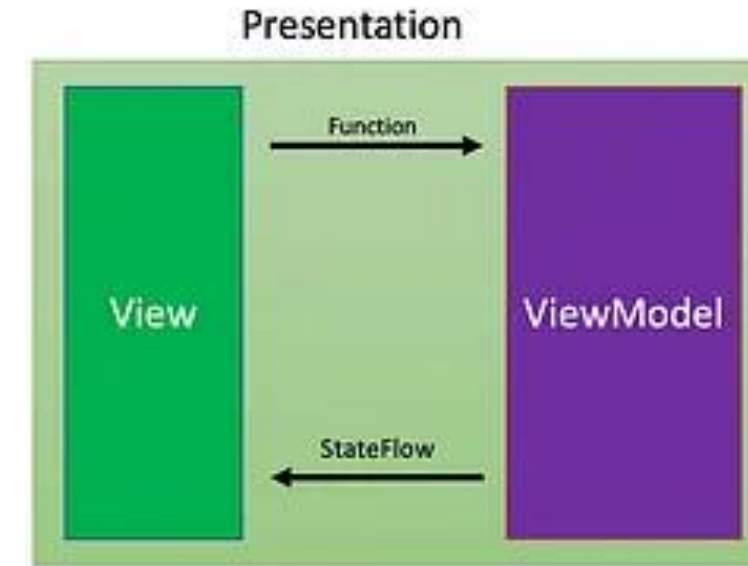
- Interaktionen des Benutzers entgegennehmen und entsprechende Aktionen auslösen

Ergebnisse anzeigen:

- Verarbeitete Daten und Ergebnisse aus der Domain Layer dem Benutzer präsentieren.

Trennung der Logik sicherstellen:

- Falls Geschäftslogik in den Views vorhanden ist, sollten diese Klassen refaktoriert werden, um die Trennung der Verantwortlichkeiten zu wahren.



Geschäftslogik verwalten:

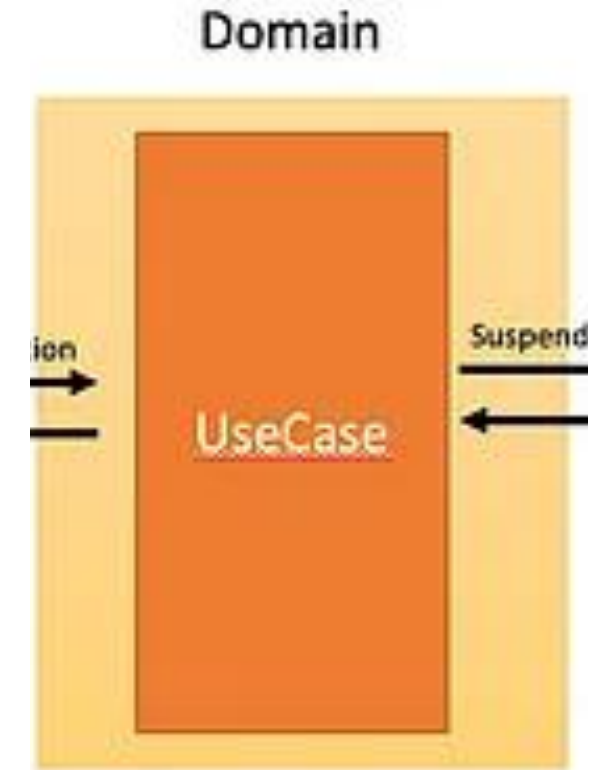
- Enthält die zentralen Regeln und Prozesse der Anwendung.
- Steuert komplexe Abläufe und Entscheidungen.

Datenverarbeitung:

- Konvertiert Rohdaten aus der Data Layer in ein nutzbares Format
 - Filtern von irrelevanten Informationen
 - Sortieren von Daten nach bestimmten Kriterien
 - Zusammenführen mehrerer Datenquellen

Bereitstellung für die Präsentationsschicht:

- Transformiert und vereinfacht Daten, damit sie von der Präsentationsschicht einfach genutzt werden können.



MVVM – DATA LAYER

Datenfluss verwalten:

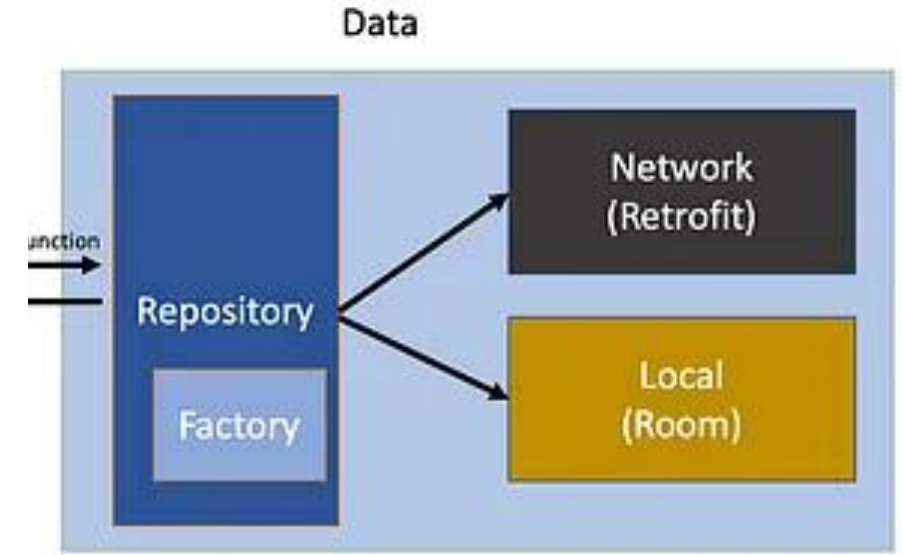
- Organisiert den Datenfluss innerhalb der Anwendung

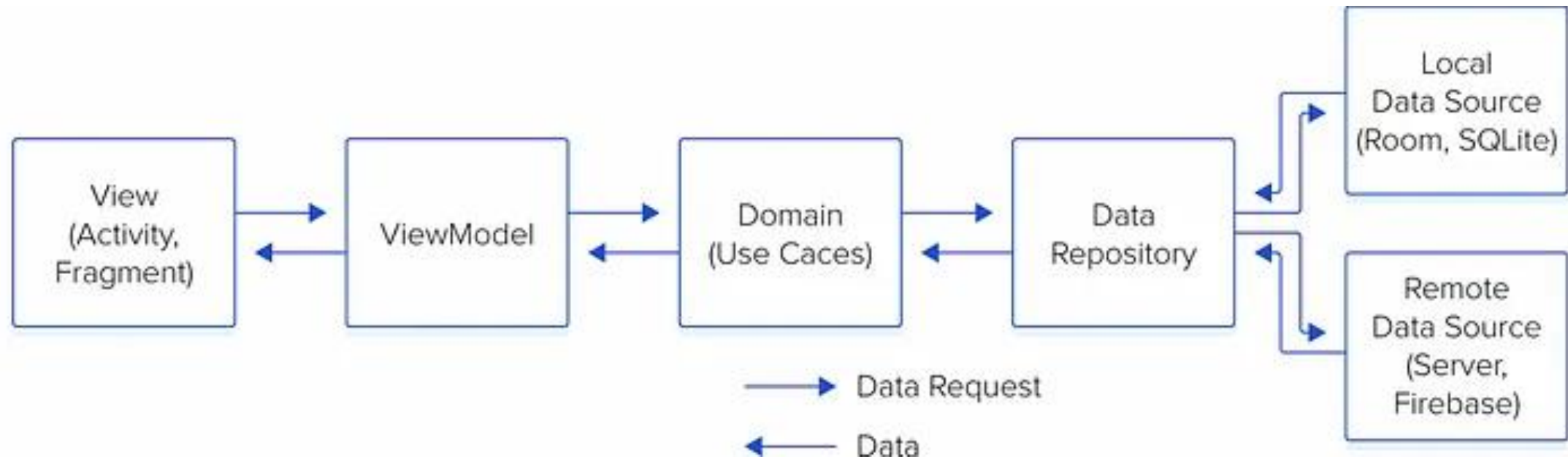
Datenpersistenz gewährleisten:

- Speichert und verwaltet Daten dauerhaft.

Abstraktionsschicht:

- Dient als Schnittstelle zwischen der Anwendung und den Daten.
- Schützt die restliche Anwendung vor direktem Kontakt mit den Datenquellen.





- **UI (View):**
 - Verwalter der View-Komponenten wie Activities und Fragments.
 - Aufgaben:
 - Anzeige und Steuerung der Benutzeroberfläche (Elemente zeigen/verstecken, Klick-Listener einrichten etc.).
 - Beobachtet Datenänderungen im ViewModel und passt die UI-Komponenten entsprechend an.

- **ViewModel:**
 - Eintrittspunkt in die Datenzugriffsschicht:
 - Sammelt Daten, indem es Flows startet und verwaltet.
 - Verantwortlich für den Lebenszyklus der Datenströme.
 - Bindeglied zwischen View und Model:
 - Stellt der UI die benötigten Daten bereit.
 - Lebenszyklusbewusst: Verhindert Speicherlecks, indem es sich an den Lebenszyklus der View hält.

- **Model:**
 - Besteht aus Klassen, die sich auf die Datenzugriffsschicht beziehen:
 - Domain-Datenmodelle, Repositorys oder APIs.

06

FORTGESCHRITTENE TECHNIKEN

THREADING

- Was ist Threading?
 - Threads sind Ausführungseinheiten innerhalb eines Prozesses.
 - Threads teilen sich Ressourcen eines Prozesses, arbeiten aber unabhängig voneinander.
 - Parallele Programmierung nutzt Threads, um mehrere Aufgaben gleichzeitig auszuführen.
- Vorteile von Threading:
 - Bessere Nutzung von Multi-Prozessorsystemen.
 - Schnellere Umschaltung und Datenaustausch im Vergleich zu Prozessen.



- **Inkonsistenter Datenzugriff:**

- Gemeinsamer Zugriff auf globale Variablen kann zu Fehlern führen.

- **Deadlocks:**

- Zyklische Abhängigkeiten zwischen Threads können zu einem Stillstand führen.
- Beispiel: Thread A wartet auf Ressource Y, die von Thread B gesperrt ist, während Thread B Ressource X benötigt, die von Thread A gesperrt ist.



- **Speicherprobleme:**
 - Freigabe von Ressourcen durch einen Thread, die noch von einem anderen verwendet werden, kann Abstürze verursachen.
- **Schwieriges Debugging:**
 - Fehler können unvorhersehbar und schwer reproduzierbar sein.



- **Was ist ein Deadlock?**
 - Zustand, in dem mindestens zwei Threads aufeinander warten und keine Fortschritte möglich sind.
 - Beispiel für Deadlock:
 - Thread A hat Ressource X und benötigt Ressource Y.
 - Thread B hat Ressource Y und benötigt Ressource X.
- Beide Threads blockieren sich gegenseitig und bleiben im Wartezustand.



- **Was ist ein Deadlock?**
 - Zustand, in dem mindestens zwei Threads aufeinander warten und keine Fortschritte möglich sind.
- **Lösungsideen:**
 - Algorithmen zur Deadlock-Vermeidung (z. B. Ressourcen in einer festen Reihenfolge anfordern).
 - Timeouts für Ressourcenzugriffe.



06

FORTGESCHRITTENE TECHNIKEN

ANWENDUNGSSPEICHER

Seit Android 6 (API 23) ist der **Zugriff auf Speicher mit Laufzeitberechtigungen** gesichert.

- **Interner Speicher (Internal Storage):** Speicherplatz, der nur von der App selbst genutzt werden kann.
→ Daten wie Einstellungen, Datenbanken oder temporäre Dateien werden hier abgelegt. Wird gelöscht, wenn die App deinstalliert wird!

- **Externer Speicher (External Storage):** Speicherplatz außerhalb der App, z. B. die SD-Karte.
→ Kann von anderen Apps oder dem Benutzer direkt eingesehen werden. Für persistente Daten, z. B. Bilder, Dokumente, Medien. Ab Android 10 eingeschränkt durch das Konzept Scoped Storage (wird mittelfristig zum Standard).

- **Cache-Speicher:** Temporärer Speicher zur Performance-Optimierung.
→ Kann automatisch oder manuell gelöscht werden.

GRUNDLAGEN ZUM ANWENDUNGSSPEICHER

- Datenbank-Speicher: SQLite-Datenbanken, z. B. für strukturierte Daten.
→ Liegt im internen Speicher; Zugriff via SQLiteOpenHelper.
- Shared Preferences: Schlüssel-Wert-Paare zur Speicherung kleiner Konfigurationsdaten.
→ Auch im internen Speicher; Beispiel: Login-Status oder Nutzereinstellungen.

06

FORTGESCHRITTENE TECHNIKEN

JETPACK COMPOSE

Was ist Jetpack Compose?

- Jetpack Compose ist das moderne, deklarative UI-Toolkit von Android.
- Es ermöglicht die Erstellung von benutzerfreundlichen Oberflächen mit weniger Code und besserer Wartbarkeit.
- Es ersetzt XML-basierte Layouts durch eine Programmierschnittstelle in Kotlin.
- Deklarativer Ansatz: UI wird durch Funktionen definiert, die direkt den Zustand der App widerspiegeln.
- Weniger Boilerplate-Code: Weniger komplex und leicht verständlich.
- Flexibilität: Einfaches Erstellen von dynamischen, anpassbaren Designs.
- Integration: Vollständig kompatibel mit bestehenden Android-Apps.



WAS IST JETPACK COMPOSE



<https://www.youtube.com/watch?v=HLGvwXpv4X0>

- **Composable Funktionen:**
 - Jede UI-Komponente wird durch eine @Composable-annotierte Funktion erstellt.
- **State-Management:**
 - UI wird automatisch aktualisiert, wenn sich der Zustand ändert.
- **Layout-System:**
 - Flexibles Anordnen von Elementen mit vorgefertigten Layouts wie Column, Row, oder Box.



- **Warum Jetpack Compose verwenden?**
 - Effizient: Einfachere Implementierung von dynamischen und reaktiven Oberflächen.
 - Modularität: Jede UI-Komponente ist wiederverwendbar und testbar.
 - Zukunftssicher: Aktiver Support von Google und stetige Weiterentwicklung.



Was ist "State" in einer App?

- Der Status ist ein Wert, der sich im Laufe der Zeit ändern kann
- Er kann alles umfassen – von einer Datenbank (Room) bis hin zu einer einfachen Variablen
- Alle Android-Apps zeigen Status für den Nutzer an

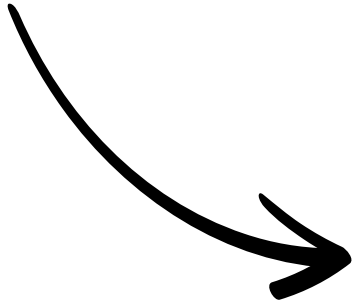
Beispiele für Status in Android-Apps

- Eine Snackbar, die angezeigt wird, wenn keine Netzwerkverbindung besteht.
- Ein Blogpost mit Kommentaren, die dynamisch geladen werden
- Wellenanimationen auf Buttons, die nach einem Klick abgespielt werden
- Sticker, die Nutzer auf ein Bild platzieren können



Jetpack Compose ermöglicht eine klare Kontrolle darüber, wo und wie Status gespeichert wird

- Es stellt APIs bereit, um einfacher mit dynamischem UI-Status zu arbeiten
- Status kann in Composable-Funktionen gespeichert und weitergegeben werden

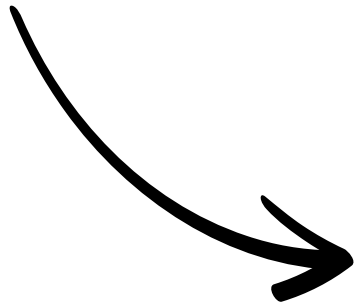


- Jetpack Compose ist deklarativ, was bedeutet, dass UI-Elemente nur durch erneutes Aufrufen des Composables mit neuen Argumenten aktualisiert werden
- ein Composable muss explizit über den neuen Status informiert werden, damit es sich aktualisieren kann
- Änderungen im Status müssen gezielt weitergegeben werden



Jetpack Compose ermöglicht eine klare Kontrolle darüber, wo und wie Status gespeichert wird

- Es stellt APIs bereit, um einfacher mit dynamischem UI-Status zu arbeiten
- Status kann in Composable-Funktionen gespeichert und weitergegeben werden



- Jetpack Compose ist deklarativ, was bedeutet, dass UI-Elemente nur durch erneutes Aufrufen des Composables mit neuen Argumenten aktualisiert werden
- ein Composable muss explizit über den neuen Status informiert werden, damit es sich aktualisieren kann
- Änderungen im Status müssen gezielt weitergegeben werden



Speicherung von Objekten im Arbeitsspeicher

- Zusammensetzbare Funktionen können die remember API nutzen, um Werte über Kompositionszyklen hinweg im Arbeitsspeicher zu speichern
- Ein mit remember berechneter Wert wird bei der ersten Komposition gespeichert und bei einer Neukomposition wiederverwendet



Verwendung von remember

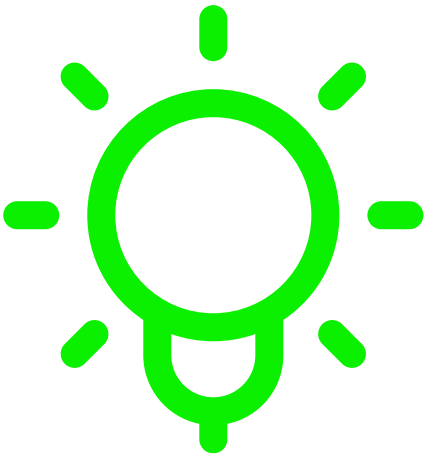
- remember kann sowohl für veränderliche als auch für unveränderliche Objekte genutzt werden
- *mutableStateOf* erstellt ein Observable MutableState<T>, das mit der Compose-Laufzeit verbunden ist und bei Änderungen automatisch eine Neuzusammensetzung auslöst

```
@Composable
private fun MyApp() {

    val mutableState = remember { mutableStateOf( value: "" ) }
    var value by remember { mutableStateOf( value: "" ) }
    val (text, setText) = remember { mutableStateOf( value: "" ) }

    TextField(
        value = text,
        onValueChange = { setText(it)},
        label = { Text( text: "Gib etwas ein" ) }
    )
}
```

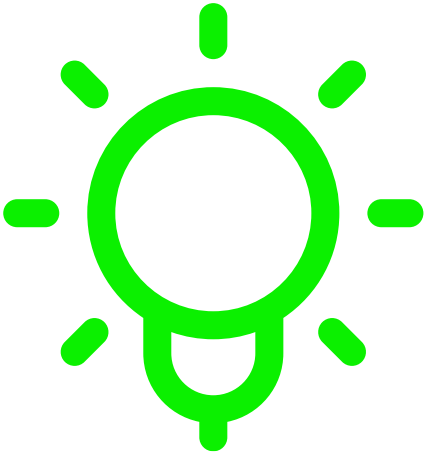




Aufgabe 1: Zähler mit remember und mutableStateOf

In dieser Aufgabe lernst du, wie du mit remember und mutableStateOf einen einfachen interaktiven Zähler erstellst.

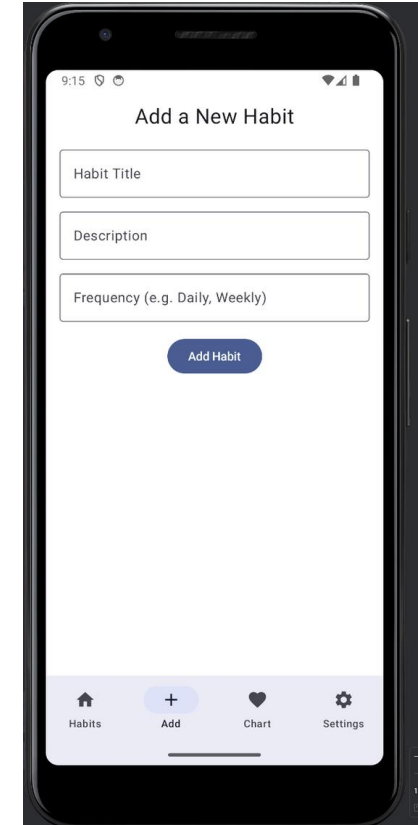
- Erstelle eine Column, die einen Text und einen Button enthält
- Der Text zeigt eine Zahl an, die mit dem Button erhöht wird
- Speichere den Zählerwert mit remember { mutableStateOf(0) }
- Bei jedem Klick auf den Button soll sich der Wert im Text aktualisieren



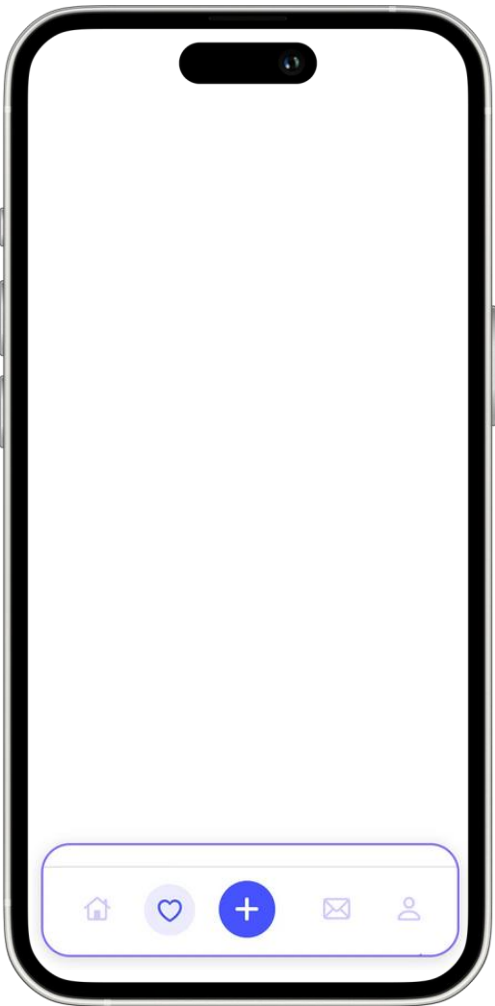
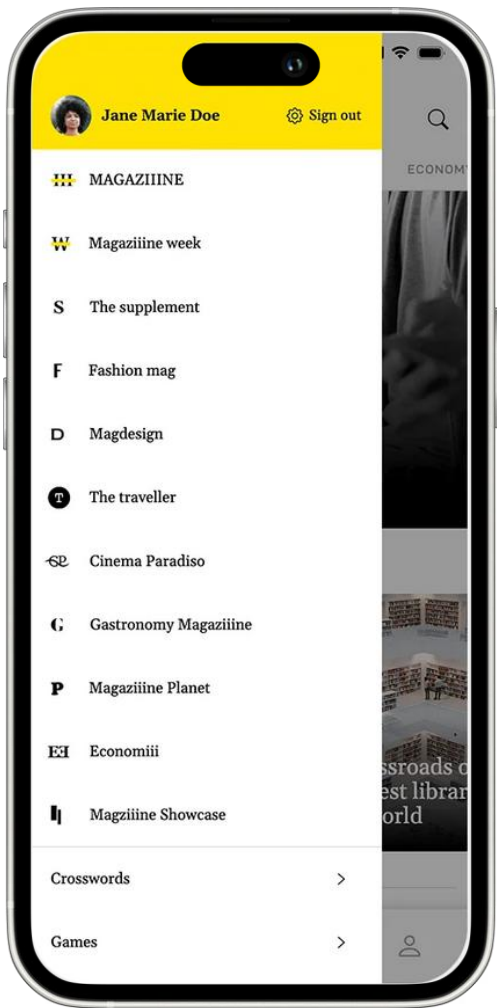
Aufgabe 2: Eingabefeld mit dynamischer Anzeige

Hier lernst du, wie `remember` genutzt wird, um Nutzereingaben in einem `TextField` zu speichern und anzuzeigen.

- Erstelle ein `TextField`, in das der Nutzer etwas eingeben kann
- Die eingegebene Zeichenfolge soll in einem `Text`-Element darunter angezeigt werden
- Verwende `remember` und `mutableStateOf`, um den eingegebenen Text zu speichern
- Der Text soll sich sofort aktualisieren, wenn sich die Eingabe ändert




JETPACK COMPOSE - NAVIGATION



Navigation Controller

- Der zentrale Koordinator für die Verwaltung der Navigation zwischen Zielen
- Der Controller bietet Methoden für die Navigation zwischen Zielen, die Verarbeitung von Deeplinks, die Verwaltung des Back-Stacks und vieles mehr



```
val items = listOf(NavDestination.Home, NavDestination.New, NavDestination.Graphics, NavDestination.Settings)
val navController = rememberNavController()
var selectedIndex by remember { mutableIntStateOf(value: 0) }
```

Navigation Host

- Ein UI-Element, das das aktuelle Navigationsziel enthält. Das heißt, wenn Nutzende durch eine App navigieren, tauscht die App im Grunde die Ziele in und aus dem Navigationshost aus
- innerhalb des Navigation Host kann ebenfalls der Startbildschirm (*startDestination*) festgelegt werden

```
@Composable
private fun NavigationHost(
    navController: NavHostController,
    innerPadding: PaddingValues
) {
    NavHost(
        navController = navController,
        startDestination = "home_screen",
        modifier = Modifier
            .background(color = Color.White)
            .padding(innerPadding)
    ) {
        composable(route = "home_screen") {
            HabitTrackerScreen()
        }
        composable(route = "new_screen") {
            AddScreen()
        }
        composable(route = "settings_screen") {
            SettingsScreen()
        }
        composable(route = "chart_screen") {
            ChartScreen()
        }
    }
}
```

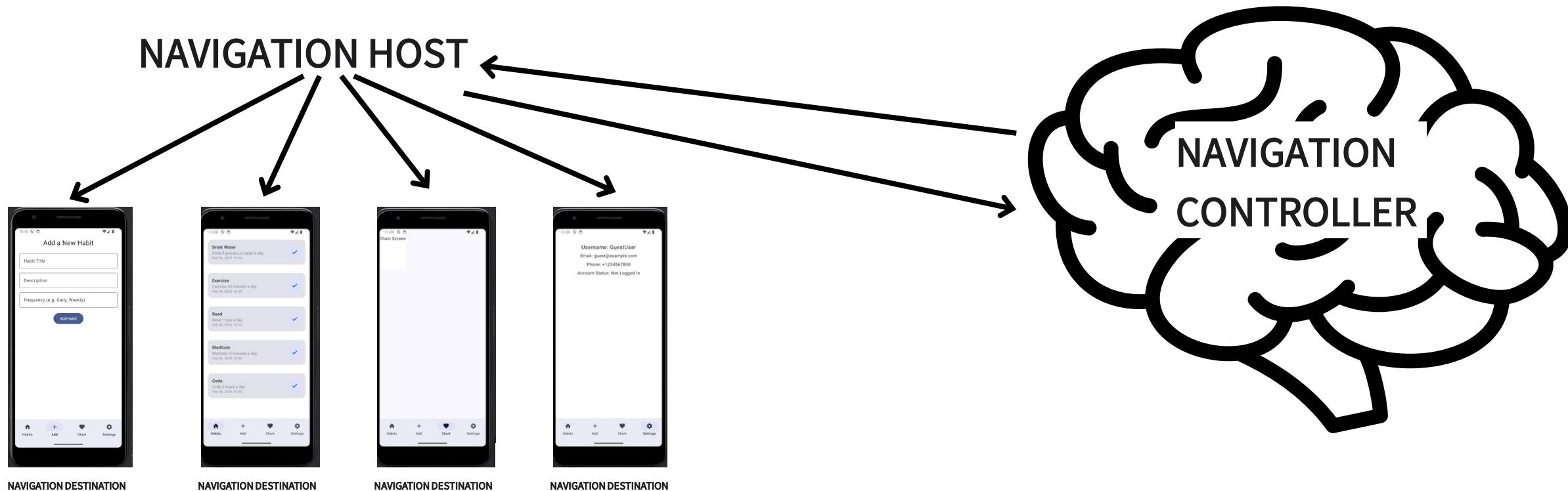
Navigation Destination

- Ein Knoten im Navigationsdiagramm. Wenn der Nutzer diesen Knoten aufruft, zeigt der Host seinen Inhalt an.
- Wird normalerweise beim Erstellen des Navigationsdiagramms erstellt.
- Enthält in unserem Beispiel zusätzlich Informationen wie “title” und “icon”

```
sealed class NavDestination(val title: String, val route: String, val icon: ImageVector) {  
    object Home: NavDestination(title = "Habits", route = "home_screen", icon = Icons.Filled.Home)  
    object New: NavDestination(title = "Add", route = "new_screen", icon = Icons.Filled.Add)  
    object Settings: NavDestination(title = "Settings", route = "settings_screen", icon = Icons.Filled.Settings)  
    object Graphics: NavDestination(title = "Chart", route = "chart_screen", icon = Icons.Filled.Favorite)  
}
```

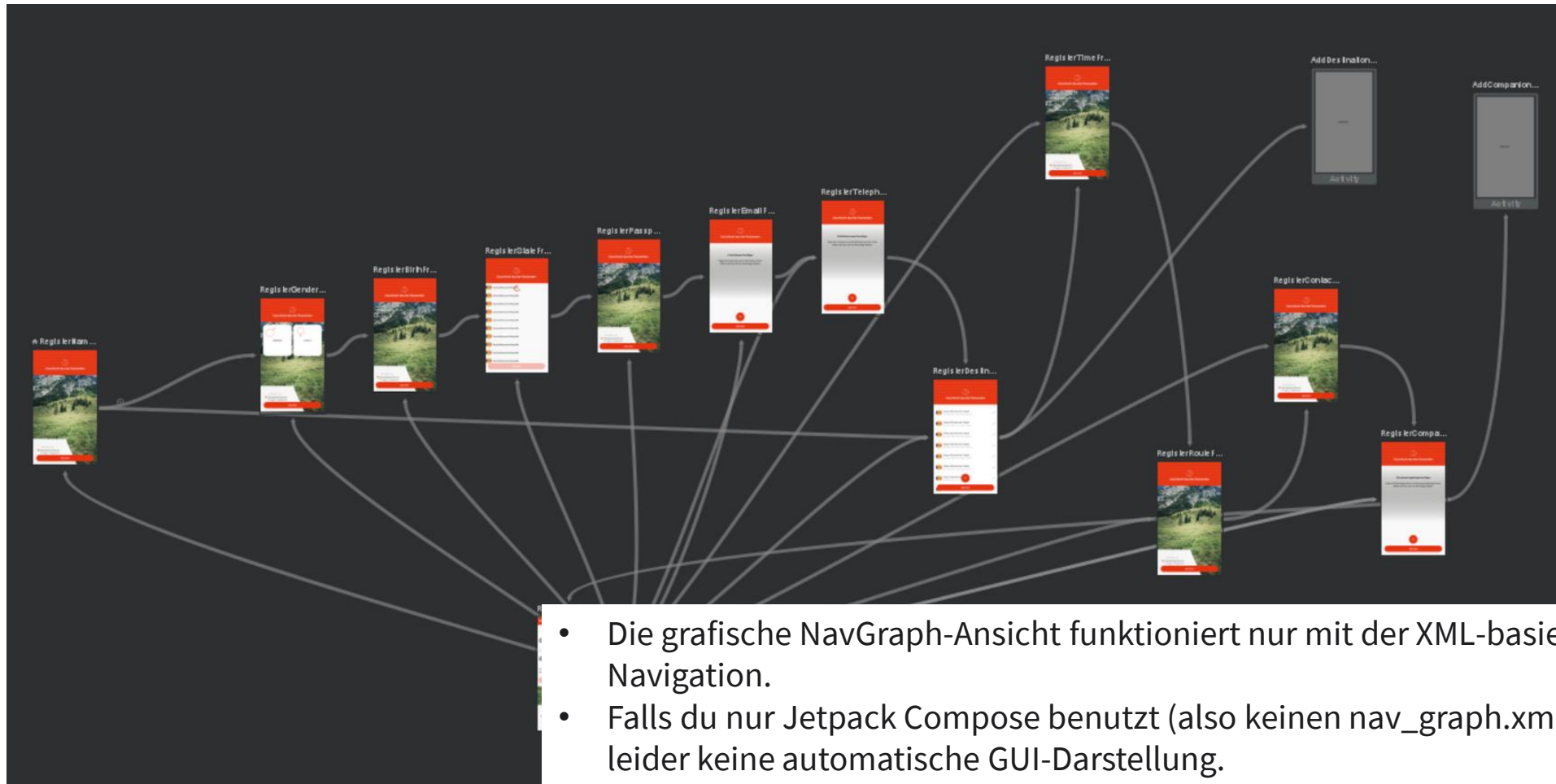
JETPACK COMPOSE - NAVIGATION

- Der NavController verwaltet den Navigationsstatus.
- Der NavHost verbindet den NavController mit den Destinations.
- Jede Destination ist eine Composable, die über den NavHost registriert wird und per `navController.navigate()` angesteuert werden kann.



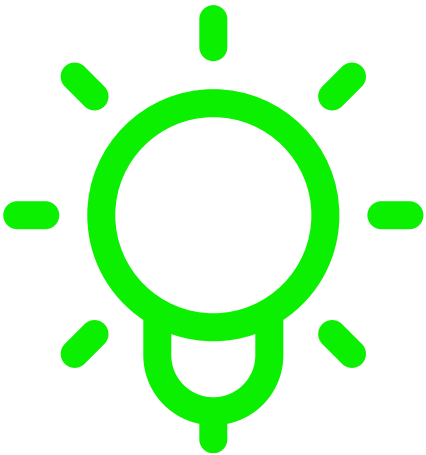
JETPACK COMPOSE - NAVIGATION





- Die grafische NavGraph-Ansicht funktioniert nur mit der XML-basierten Navigation.
- Falls du nur Jetpack Compose benutzt (also keinen `nav_graph.xml` hast), gibt es leider keine automatische GUI-Darstellung.
- Die Navigation in Jetpack Compose basiert rein auf Code (NavHost + NavGraph), weshalb Android Studio sie nicht automatisch visualisiert.

- **NavController** steuert die Navigation
- **NavHost** ist der Container, der alle Screens verwaltet
- **Navigation Destinations** sind die individuellen Composable-Screens
- Der **NavGraph** ist eine Sammlung aller möglichen Destinations
- Die Navigation kann mit Parametern und verschiedenen Steueroptionen erweitert werden (dynamische Routen)



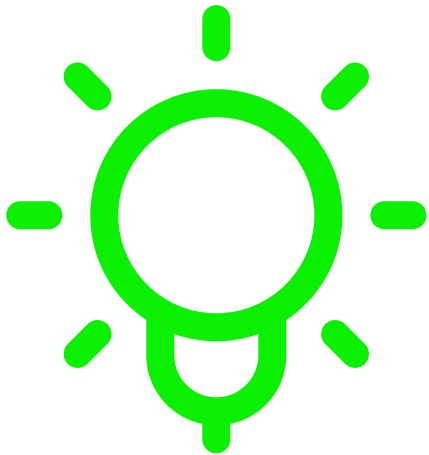
Aufgabe 1: Einstieg in die Navigation mit Jetpack Compose

Erstelle eine einfache Navigation zwischen zwei Screens in einer Jetpack Compose App. Dabei soll ein Startbildschirm einen Button enthalten, der zur zweiten Seite navigiert.

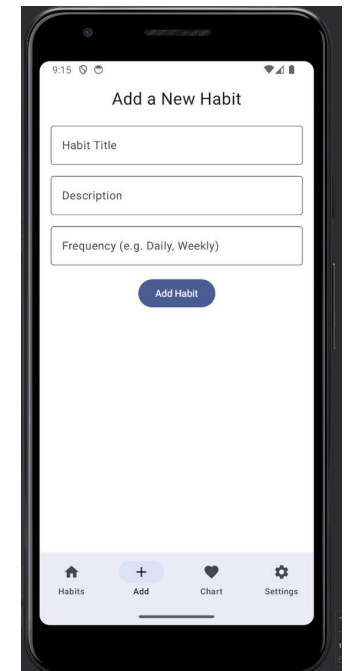
- Verwende `rememberNavController()`, `NavHost` und `composable`
- Definiere zwei Screens (`HomeScreen` und `DetailScreen`)
- Navigiere per Button von `HomeScreen` zu `DetailScreen` und zurück
- Nutze `popBackStack()`, um zum vorherigen Screen zurückzukehren

Aufgabe 2: Erstelle eine Tab-Navigation mit Jetpack Compose

Baue eine App mit einer Tab-Navigation, in der Nutzer zwischen verschiedenen Ansichten wechseln können.



- Verwende BottomNavigation mit NavHost
- Erstelle mindestens drei Tabs (Home, Profile, Settings)
- Setze Icons für die Tabs (Icons.Filled.X)
- Stelle sicher, dass der aktuell ausgewählte Tab hervorgehoben wird



DANKE