Android

# Zielsetzung und Funktionen

1. **Bildaufnahme und -übertragung**:  
   Die App ermöglicht es den Benutzern, Bilder direkt mit ihrem Gerät aufzunehmen. Diese Bilder werden anschließend sicher an das Backend gesendet, um dort verarbeitet und gespeichert zu werden.
2. **Anzeige von Bildern und Ergebnissen**:  
   Die App stellt eine benutzerfreundliche Oberfläche zur Verfügung, um die aufgenommenen Bilder sowie die damit verbundenen Ergebnisse anzuzeigen.
3. **Sicherheit und Datenschutz**:  
   Da die App mit sensiblen Daten arbeitet, wird besonderer Wert auf Sicherheitsmaßnahmen gelegt.
4. **Benutzeranleitung**:  
   Die App enthält eine integrierte Anleitung, die den Benutzern hilft, qualitative Fotos zu schießen.
5. **ansprechende Benutzeroberfläche:**Die App stellt eine ansprechende und benutzerfreundliche Oberfläche bereit, die intuitive Nutzung ermöglicht.

# Sprache

Programmiersprache: Kotlin

Zu Anfang des Projekts stand die Technologiewahl im Vordergrund, davon ist eine zentrale Komponente die Programmiersprache, auf der das ganze Projekt basiert. Zur Verfügung standen folgende Optionen:

* Java
* Kotlin
* Dart
* C++

Die Wahl der Sprache war eine einfache, aus folgenden Gründen:

Im Unterricht wurde bereits Java behandelt, jedoch ist Java anfälliger für Exceptions, als es bei Kotlin der Fall ist. Ein weiter Grund für Kotlin ist die Ähnlichkeit zu Java, da Kotlin letztlich eine Weiterentwicklung von Java ist, und auch unter Android-Applikationen die Nr. 1 Programmiersprache ist. Dart, sowie C++, waren keine Option, da sich diese nicht in unserem Techstack befinden, und der Lernaufwand dadurch nochmal drastisch steigen würde.

# Architektur

Bei Model-View-ViewModel (MVVM) handelt es sich um eine Architektur, die die Benutzeroberfläche von der Geschäftslogik und den Daten trennt. Es sorgt dafür, dass die App leicht wartbar, testbar und erweiterbar bleibt. Die Architektur besteht aus drei Hauptkomponenten:

* Model
* View
* ViewModel



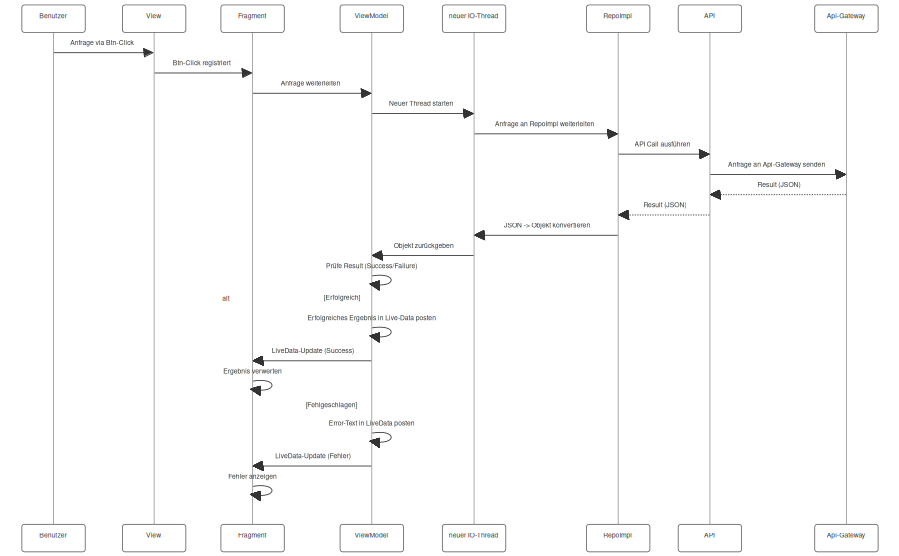
1. **Model**:  
   Das Model repräsentiert die Daten und Geschäftslogik der Anwendung. Es enthält alles, was mit der Datenhaltung und -verarbeitung zu tun hat. Das Model weiß jedoch nichts über die Benutzeroberfläche oder wie die Daten angezeigt werden. Die Model als auch die ViewModel bestehen aus Kotlin-Dateien.
2. **View**:  
   Die View ist für die Darstellung der Benutzeroberfläche verantwortlich. Sie zeigt die Daten an, die vom ViewModel bereitgestellt werden, und empfängt Benutzerinteraktionen. Sie umfasst die XML-Dateien.
3. **ViewModel**:  
   Das ViewModel fungiert als Vermittler zwischen der View und dem Model. Es stellt die Daten bereit, die die View benötigt, und verarbeitet Benutzerinteraktionen. Es kümmert sich darum, dass die View mit den richtigen Informationen versorgt wird, ohne selbst direkt mit dem Model oder der View zu interagieren. Dabei darf ein ViewModel keine direkten Referenzen von View-Elementen, noch den Kontext von Models halten, da in diesem Fall Probleme mit dem Lifecycle der einzelnen Elemente auftreteon können.

Zusätzlich zu dieser Architektur wurden Repositorys implementiert, die mittels Dependeny-Injection in die jeweiligen ViewModel eingebunden werden. Die Auslagerung sämtlicher Serveraufrufe in diese Klassen verbessert das Separation of Concerns (SoC) des Projekts erheblich, da die Logik für die Kommunikation mit dem Server nun zentralisiert ist und ausschließlich in ViewModels initialisiert wird. Dadurch wird die Struktur klarer und der Code besser wartbar, wie das folgende Zitat verdeutlicht:

„**By separating the concerns of your app into Model, View, and ViewModel components, you can build robust and maintainable applications. This pattern, when combined with Android’s built-in features like data binding and LiveData, simplifies the development process, making it more efficient and scalable.**“ (Vanasi, 2023)

## Sequenzdiagramm – Weg zum Backend

Dieses Diagramm zeigt den Ablauf einer Benutzerinteraktion, wenn der Benutzer Aktionen vornimmt, die Anfragen an das Backend senden.



1. **Benutzerinitiiert Anfrage**: Der Benutzer klickt auf einen Button in der View, was das Fragment dazu bringt, die Anfrage an das ViewModel weiterzuleiten.
2. **Thread und Datenabruf**: Das ViewModel startet einen neuen Hintergrund-Thread, um die Anfrage asynchron zu verarbeiten, ohne die Benutzeroberfläche zu blockieren. Dieser Thread leitet die Anfrage an das Repository weiter.
3. **API-Anfrage**: Das Repository wiederum bedient sich an der Klasse API, die die Anfragen letztlich versendet.
4. **Verarbeitung der Antwort**: Die Antwort des Server kommt anschließend auch in das Repository, in der der JSON-String in ein Objekt konvertiert wird.
5. **Ergebnisprüfung und Update**: Das ViewModel prüft das Ergebnis und aktualisiert das LiveData. Bei Erfolg wird die Antwort in das entsprechende Live-Data gepostet und entsprechend weiterverarbeitet. Schlägt etwas fehl, wird die Fehlernachricht des Servers in ein eigenständiges Live-Data gepostet, das ausschließlich für die Fehlerdarstellung zuständig ist und den Benutzer über das Problem informiert.

### Fehlerbehandlung

In der Architektur der Android-Anwendung spielt die Fehlerbehandlung eine entscheidende Rolle, um eine benutzerfreundliche Benutzeroberfläche zu gewährleisten. Sowohl in **ViewModels** als auch in **Fragments** werden Fehler auf strukturierte Weise erfasst, behandelt und an die Benutzeroberfläche weitergegeben.

### Fehlerbehandlung im ViewModel

Ein **ViewModel** dient als Bindeglied zwischen der **UI** und der **Datenquelle**. Es sorgt dafür, dass UI-bezogene Daten erhalten und Fehler sicher behandelt werden, ohne dass die Benutzeroberfläche in das ViewModel selbst eingebunden wird. In der Fehlerbehandlung innerhalb eines **ViewModels** erfolgt die Kommunikation über LiveData-Objekte, die Fehlerereignisse und Datenänderungen an das Fragment weitergeben, insofern dieser einen Observer für die gewünschte Variable implementiert.

### Fehlerbehandlung im Fragment

Ein **Fragment** ist die UI-Komponente, die mit dem **ViewModel** kommuniziert und die Fehler, die im ViewModel auftreten, observiert und verarbeitet. Das Fragment beobachtet die LiveData-Objekte des ViewModels und reagiert entsprechend der Daten die es erhält. Sollte das ViewModel eine negative Antwort vom Backend erhalten, wird die Fehlermeldung in die Variable „message“ gepostet. Diese wird im Fragment überwacht und löst eine **Toast-Message** aus, wenn sich der Wert ändert. Im Falle einer positiven Nachricht wird das Ergebnis in einer dedizierten Variable gespeichert, die je nach Funktion weitere Ereignisse auslösen kann.

# Datenquellen

## Datenbank

Grundsätzlich ist das ganze System so designed, dass es möglich wäre alle Daten ausschließlich in der Datenbank zu speichern, ohne das Dateisystem des Geräts zu verwenden. Allerdings fiel im Laufe des Entwicklungsprozesses auf, dass das ständige Laden der Bilder und Diagnosen zu einer langen Wartungszeit geführt hat. Um das Netzwerk zu entlasten, werden die Bilder sowie Diagnosen nun im lokalen Dateisystem hinterlegt.

## lokales Dateisystem

Um die Privatsphäre der Nutzerdaten zu maximieren, wird für alle gespeicherten Daten „Scoped Storage“ verwendet – ein Feature, das seit Android 10 existiert. Es sorgt dafür, dass der Speicherzugriff auf die Daten einzelner Apps streng limitiert wird, sodass jede App nur auf ihren eigenen Speicher zugreifen kann. Neben den Bildern befindet sich auch eine JSON-Datei im Dateisystem, die die Diagnosen der einzelnen Bilder speichert. Die JSON-Datei speichert das Modell „Diagnosis“, das sowohl die Vorhersage als auch den Pfad des jeweiligen Bildes im Speicher enthält. Da die Dateinamen so generiert werden, dass sie eindeutig sind, kann jede Diagnose eindeutig einem einzelnen Bild zugeordnet werden. Wenn ein Benutzer ein Foto aufnimmt, wird es zuerst im **lokalen Dateisystem** gespeichert und anschließend in der **Datenbank** abgelegt. Eine Synchronisierung der Bilder ist nur erforderlich, wenn der Benutzer Bilder aus dem lokalen Speicher löscht oder das Gerät verliert. In diesem Fall kann er die Bilder sowie die zugehörigen Diagnosen aus der Datenbank zurück in den lokalen Speicher herunterladen.

# Koin

Koin ist ein Dependency-Injection-Framework, das sich besonders für Android-Apps eignet, welche auf Kotlin zurückgreifen.

## Alternativen

Neben Koin hätte sich noch das Framework Hilt angeboten, welches nach einer detaillierten Evaluierung allerdings verworfen wurde, da Koin folgende Vorteile bereitstellt, die Hilt oder Dagger nicht besitzen:

1. **Einfachheit**

Koin ist einfacher und flexibler, da es auf einer Kotlin-DSL basiert und keinen Code-Generierungsaufwand benötigt, sowie keine Verwendung von Annotationen nötig sind.

1. **Performance**

Koin benötigt keine Code-Generierung, was den Build-Prozess beschleunigt. Stattdessen erfolgt die Abhängigkeitsauflösung zur Laufzeit, was den Build-Prozess deutlich vereinfacht.

Um Koin zu implementieren, benötigt es mehrere Komponenten:

1. **MyApplication.kt**: Startet die Dependency-Injection-Instanz.



1. **AppModule.kt**: Definiert das Koin-Modul, das Abhängigkeiten bereitstellt. Es registriert drei Singletons:
   1. LoginRepoImpl
   2. ImageRepoImpl
   3. UserRepoImpl
   4. AdminRepoImpl
   5. ModelRepoImpl



Die Klassen LoginRepoImpl, ImageRepoImpl, ModelRepoImpl und AdminRepoImpl tätigen mittels der Klasse API die Api-Calls an das Backend. UserRepoImpl hingegen verwaltet die Benutzerdaten nach der Authentifizierung und stellt sie in den anderen Komponenten des Frontend bereit und ist somit unabhängig der Lifecycle der ViewModels. Zusätzlich wird der Klasse den Kontext im Konstruktor übergeben.

Eingebunden werden die Dependencys anschließend ausschließlich in den ViewModels auf folgende Art und Weise:



# Component-Diagramm

Das Diagramm beschreibt die Kommunikation mit dem Backend und dient als Frontend für die Login/Registrierung und Fotoaufnahme. Beide Repos nutzen POST/GET-Anfragen, die Daten im JSON-Format verarbeiten. Die Anfragen werden an eine Python-basierte Gateway API weitergeleitet, die als Vermittler zum Backend dient.

Ein Bild, das Text, Diagramm, Reihe, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

# Klassen

## Modelle

Für den Datenaustauch mit dem Gateway werden die Klassen vor dem Transfer, sowohl vom Client als auch vom Backend, in ein JSON-Objekt konvertiert und anschließend auf der Empfängerseite, je nach API-Aufruf, in das entsprechende Datenmodell konvertiert. Insgesamt kommen mehrerer Datenmodelle zum Einsatz, aus Relevanz-Gründen sind aber nur die drei wichtigsten näher beschrieben:

1. Prediction:



1. User

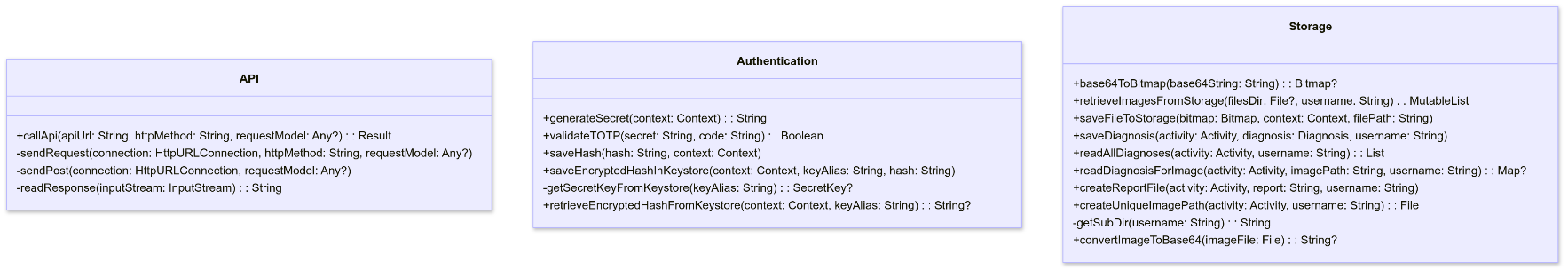


1. Diagnosis

## Helferklassen

Diese Klassen erfüllen jeweils spezifische Funktionen, ohne eigene Datenfelder zu besitzen. Ihre Methoden sind zudem statisch implementiert, sodass sie ohne Instanziierung der Klasse direkt aufgerufen werden können.

1. API.kt: Sendet GET/POST-Anfragen an das API-Gateway.
2. Storage.kt: Verwaltet die aufgenommenen Bilder im Speicher des Geräts.
3. Authentication.kt: Erzeugt den Key für 2FA und validiert das eingegebene TOTP.



# Fragments

## Gallery

Um die vom Benutzer geschossenen Bilder anzuzeigen, war eine Implementierung einer Gallery notwendig. Sobald der Benutzer diese Seite aufruft, wird diese programmatisch mittels der Methode „*fillView(images: List<File>)“* mit den geschossenen Bildern befüllt, sobald diese aus dem Speicher geladen wurden. Um ein geordnetes Design zu schaffen, spielen dabei mehrere Elemente zusammen.

Der Ablauf sieht folgendermaßen aus:

1. Der Benutzer ruft die Seite auf, indem er auf „Gallery“ in der Nav-Bar drückt.

Nun beginnt der erste Schritt des Prozesses, indem zuallererst das Directory definiert wird, und anschließend die Bilder aus diesem geladen werden.



1. Methoden-Aufruf im ViewModel: LoadImages(filesDir)

Um zu verhindern, dass der Main-Thread einfriert, wird der eigentliche Methoden-Aufruf aus der Klasse „*Storage*“ im ViewModel aufgerufen. Alle Threads werden aus Best-Practice-Gründen ausschließlich in ViewModels verwaltet.



Nach dem Abrufen der Bilder werden diese als List<File> in ein MutableLiveData-Objekt gespeichert.



1. Observer wird ausgelöst

MutableLiveData ist ein Datentyp, der es ermöglicht, Änderungen zu beobachten. Um diese Beobachtung zu implementieren ist folgender Teil im Fragment notwendig. Er wird immer dann ausgeführt, wenn sich die Variable „*images*“, die sich im ViewModel befindet, verändert wird.



1. Laden der Bilder aus dem Speicher

Zurück zu Punkt 2. Dort wird mittels der Hilfsklasse „*Storage“* alle Dateien mit der Dateienendung „jpg“ als Liste von Files zurückgeben. Davor wird allerdings ein Unterverzeichnis angelegt, welches sich mit dem Benutzernamen zusammensetzt, um eine klare Trennung von Bildern/Usern zu gewährleisten.



1. Eigentliche befüllen der View

In Schritt 3 erfolgt die eigentliche Erstellung und Befüllung der Seite mit den geladenen Bildern.

1. Thumbnail erstellen

Zunächst wird der Container geleert, um potentielle Fehler im Vorhinein aus dem Weg zu gehen. Im Anschluss wird über jedes geladene Bild iteriert und ein Bitmap aus diesen erzeugt, mit den Maßen 300px \* 200px, welches als Thumbnail dient.



1. horizontalen Container und Event Listener konfigurieren

Im weiteren Verlauf wird ein Container erstellt, welcher horizontal ausgerichtet ist, und die Funktion erfüllt das Thumbnail, sowie einen Text unterzubringen. Da das angezeigte Thumbnail runterskaliert wurde, ist noch ein Event-Listener von Wichtigkeit, der beim Anklicken des Containers die Aktivität „*ResultActivity*“ startet und dabei den Pfad des jeweiligen Bildes mitsendet. Diese Aktivität erfüllt den Zweck, das Bild nochmal in voller Qualität darzustellen, sowie die Diagnose als Text darunter.



1. Konfiguration der inneren Container

Die zwei weiteren Container, die sich aus dem Thumbnail und dem erwähnten Text zusammensetzen, benötigen ebenso eine Konfiguration. Wie alle anderen Container auch, wird „*LayoutParams*“ auf „*WRAP\_CONTENT*“ gesetzt, mit der Funktion, dass der Container nur so viel Platzt einnimmt, wie nötig. Für den Image-Container wird noch das Thumbnail als Inhalt gesetzt, und für den Text-Container den Namen der Datei.



1. Übergeordnetem Container hinzufügen

Abschließend werden die beiden inneren Views dem horizontalen Container hinzugefügt, und dieser wiederum wird dem übergeordneten Container hinzugefügt.



## Admin

Die Admin-Seite stellt verschiedene Funktionen bereit, die ausschließlich für Administratoren verfügbar ist. Dies geht aus dem Funktionsumfang hervor, da dieser die zusätzlich zu den gewohnten Funktionen, noch die Möglichkeit hat, KI-Modelle erneut zu trainieren, sowie Reports anzufordern, die zusätzliche Daten preisgeben. Bei beiden Funktionen hat der Administrator die Wahl, ob er alle KI-Modelle trainiert oder Reports für alle Modelle anfordert, oder nur ein spezifisches Modell oder ein spezifischen Report.

## Accountinfo

# Verwaltung von Textressourcen

In der Android-Anwendung werden alle Benutzeroberflächentexte und API-Endpunkte zentral in XML-Dateien verwaltet. Dies erleichtert die Wartbarkeit, ermöglicht eine einfache Anpassung von Texten sowie eine einfache Möglichkeit Texte zu übersetzen, sollte dies in Zukunft ein Ziel sein.

1. **Zentrale Verwaltung von Textressourcen**

Die Textressourcen der Anwendung sind in der Datei „*strings.xml*“ zusammengefasst. Diese Datei enthält alle für die Benutzeroberfläche relevanten Texte, wie z.B. Titel, Beschriftungen von Buttons. Durch die zentrale Verwaltung dieser Ressourcen kann eine einfache Wartung und Anpassung der Texte gewährleistet werden. Änderungen an Texten müssen nicht direkt im Code vorgenommen werden, sondern können direkt in der “*strings.xml*” erfolgen.

1. **Verwaltung von API-Endpunkten**

Die links.xml-Datei enthält URLs und API-Endpunkte, die in der Anwendung verwendet werden. Diese zentrale Verwaltung von URLs macht es einfach, die Endpunkte zu ändern, ohne dass Änderungen im gesamten Code vorgenommen werden müssen. Bei Änderungen an der Backend-URL reicht es aus, die “*links.xml*” zu aktualisieren, was die Wartung der Anwendung erheblich vereinfacht. Ein praktisches Beispiel für den Einsatz dieser Datei in diesem Projekt ist die Einführung des API-Gateways zu einem späteren Zeitpunkt. Als das API-Gateway entwickelt wurde, konnten die URLs in der links.xml-Datei einfach angepasst werden, ohne den bestehenden Code ändern zu müssen.

# Scherheit

Da die Diagnosen und Bilder der Benutzer hochsensible Daten enthalten, ist es von Notwendigkeit, für dafür zu sorgen, dass auch nur autorisierte Benutzer Zugriff darauf haben. Aus diesem Grund wurden zwei Sicherheitsmechanismen implementiert:

## Inputvalidierung

Um zu gewährleisten, dass es sich bei den Benutzerdaten um eine Email und ein starkes Passwort handelt, wurde dies clientseitig implementiert. Der folgende Snippet stellt sicher, dass der Benutzer beim Registrieren sein Passwort zwei Mal eingeben muss, um Tippfehler zu vermeiden, sowie dass eine Mindestlänge von 8 Zeichen eingehalten wird:



## 2FA – erste Herangehensweise

Um einen Key für einen Authenticator zu erstellen wurde die Dependency „*dev.samstevens.totp*“ verwendet, da sie eine simple Lösung bereitstellt. Der Key wurde generiert und anschließend mithilfe der Methode „*saveHash(secret,context)“* im KeyStore geseichert.



Wenn der Benutzer sich schließlich anmeldet, wird das eingegebene TOTP validiert. Damit eine Validierung möglich ist, braucht die Methode den zuvor generierten Key, sowie die aktuelle Systemzeit, um den zu erwartenden Code mittels dem Verifier zu überprüfen.



### Problem

Wie bereits erwähnt wird der generierte Key im KeyStore des Geräts gespeichert, was bedeutet, dass dieser nur lokal auf dem Gerät verfügbar ist, mit dem sich der Benutzer initial registriert hat. Sollte der Benutzer sein Gerät verlieren, ist auch der Key verloren und er hat keine Möglichkeit mehr sich mit diesem Account anzumelden. Aus diesem Grund musste die Implementierung so geändert werden, dass eine Synchronisierung zwischen Geräten möglich ist, ohne den Key auch in jener Datenbank zu speichern, in der die E-Mail und das Passwort hinterlegt sind.

# Verzeichnisse

Innerhalb des Projekts arbeiten viele verschieden Dateien zusammen. Dabei gibt es verschieden Verzeichnisse für verschiedene Funktionen.

Die wichtigsten Verzeichnisse sind folgende:

„Android\app\src\main\java\com\example\dermaai\_android\_140\ui“

Dieses beeinhaltet weitere Unterverzeichnisse, die die Klassen (Fragment und ViewModel) zur zugehörigen XML-Datei beeinhaltet.

„Android\app\src\main\java\com\example\dermaai\_android\_140\myClasses“

Hier befinden sich die Helper-Klassen, wie „Storage“, „AppModule“ oder „RequestCallback“.

„Android\app\src\main\java\com\example\dermaai\_android\_140\repo“

Enthält die Interfaces für die Klassen in folgendem Verzeichnis:

“Android\app\src\main\java\com\example\dermaai\_android\_140\repoImpl”

Enthält den Code, der letztendlich den Aufruf zur API tätigt.

„Android\app\src\main\res“

Hier befinden sich weitere Unterverzeichnisse, die ausschließlich XML-Dateien beinhalten, und sich erneut jeweils von ihrer Funktion unterscheiden. Die wichtigsten Dateien befinden sich dabei im Verzeichnis:

„Android\app\src\main\res\layout”

Grundlegende Layout-Dateien, wie die Homepage oder Login-Page, sind hier hinterlegt.

# Erstellen der APK

Mittels des Commands „gradlew assembleRelease” kann die APK erstellt werden. Bei der APK handelt es sich um jenes Dateiformat, mit welchem Apps auf Android-Geräten installiert werden können. Sie beinhaltet alle notwendigen Dateien, einschließlich Dependencys, Codedateien, und Bilder. Beim Erstellen der APK werden mehrere Schritte abgearbeitet, so wird im Schritt „app:stripReleaseDebugSymbols“ Debug-Symbole aus den Bibliotheken entfernt, um die App-Größe zu reduzieren. Anschließend wird der Code compiliert. Die APK ist letztlich hier zu finden:

“Android\app\build\outputs\apk\release\app-release.apk”A screenshot of a computer program

Description automatically generated

# UI

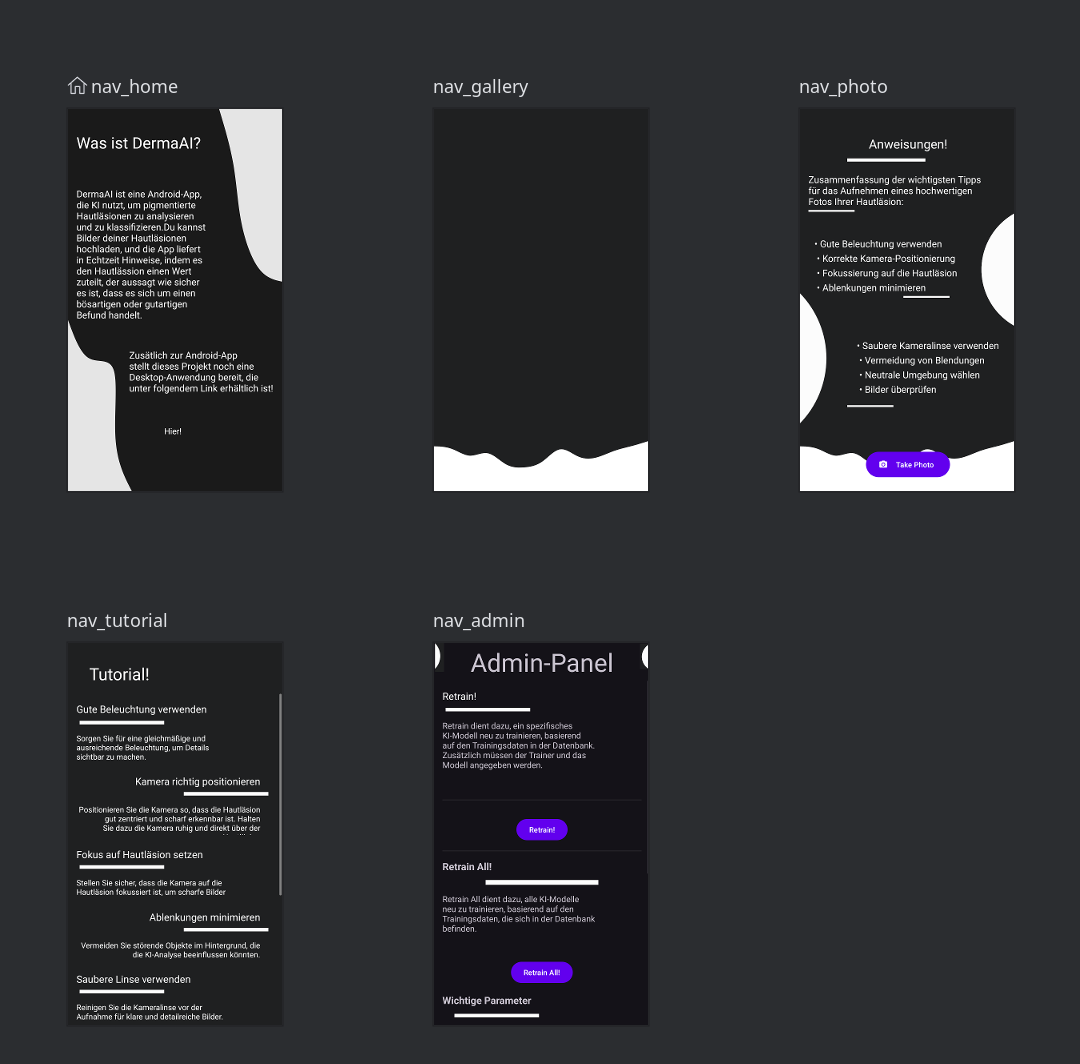
## Design

Für das Design der App wurde Material Design 3 Dark als Designsystem gewählt, da es eine moderne und benutzerfreundliche Ästhetik bietet, die sich in den Icons und der vordefinierten Farben äußert.

Um ein ansprechendes Design für die App zu gestalten, wurden simple Vektorgraphiken mit der Website haikei.app erstellt. Beim Hinzufügen eines neuen Vektor-Assets in Android Studio erfolgt automatisch eine Konvertierung in eine XML-Datei, was die Integration in die App erheblich vereinfacht, da sich diese an alle Bildschirmgrößen anpasst.

## Navigation

Um der Anwendung zusätzliche Seiten hinzuzufügen, die nicht Teil des Ablaufs einer bereits bestehenden Seite sind, sondern eine eigenständige Funktionalität bieten, müssen diese in der Datei „*mobile\_navigation.xml*“ eingetragen werden. Bei den einzelnen Seiten ist wichtig zu beachten, dass nur Fragments verwendet werden können.



### Vorteile

Das Host-Element all dieser Elemente ist die MainActivity. Dieses Konzept wurde aus folgenden Gründen umgesetzt:

1. **Wiederverwendbarkeit**

**Fragments** ermöglichen es, Teile der Benutzeroberfläche und Logik einer App wiederverwendbar und modular zu gestalten. So können einzelne Komponenten wie UI-Elemente und Funktionen in verschiedenen Kontexten innerhalb der App verwendet werden, ohne dass eine neue Activity für jede Seite erstellt werden muss.

1. **Übersichtlichkeit**

Eine Activity stellt einen einzelnen Bildschirm dar, und die Verwaltung von vielen **Activities** wird schnell unübersichtlich.

1. **Lebenszyklusmanagement**

Der Lebenszyklus von **Fragments** ist eng mit dem der **Activity** verbunden, was bedeutet, dass beim Wechsel zwischen Fragmenten der Zustand der App effizienter verwaltet werden kann, ohne die gesamte **Activity** neu starten zu müssen. Diese enge Integration des Lebenszyklus führt zu einer besseren Performance, da nur die Fragmente verwaltet werden, anstatt die komplette **Activity** neu zu laden.

# Manifest

Das Manifest mit dem Namen „AndroidManifest.xml“ spielt eine wichtige Rolle bei der Konfiguration des Projekts und wird in jeder Android-Applikation implementiert und erfüllt unter anderem folgende Aufgaben:

* Definiert Berechtigungen
* Erklärt Intents
* Implementiert den File-Provider

## Probleme

Wenn der Benutzer die Activity wechselt, um beispielsweise die Seite zu wechseln, stürzt die App ab. Um diesen Fehler zu korrigieren, muss die Activity im Vorfeld im Manifest folgendermaßen registriert werden:



## Permissions

Aufgrund der Funktionalität, die die Applikation bietet, benötigt diese verschiedenen Berechtigungen, um ordnungsgemäß arbeiten zu können.



Innerhalb dieses Blocks wird definiert, dass die Applikation die Funktionalität bereitstellt, Fotos schießen zu können. Diese Zeile Code berechtigt die App nicht automatisch zur Verwendung der Kamera, diese muss weiterhin explizit vom Benutzer genehmigt werden. Damit dies überhaupt erst möglich ist, muss es jedoch im Manifest wie im Code-Abschnitt angegeben werden.



Auch hier wird eine Berechtigung definiert, die es dem Programm erlaubt, in den externen Speicher zu schreiben. „ScopedStorage“ ist dabei eine Limitierung des Pfades, in der die Applikation schreiben kann, die in Android 10 eingeführt wurde. Jede Applikation erhält seinen eigenen Ordner im Android-Verzeichnis. Der Pfad dieser Applikation sieht dabei folgendermaßen aus:

“/storage/emulated/0/Android/data/com.example.dermaai\_android\_140”

Zu guter Letzt benötigt die Anwendung noch Internetzugriff, um die Api-Requests zu tätigen.



# Activitys

Die Aktivität ist eine zentrale Komponente in einer Android App. Zusammen mit Fragments, sind sie der größte Teil im Bereich der UI. Kleinere UI-Elemente wie Knöpfe oder Bilder werden in diesen Elementen gehostet, daher sind ihre Lebenszyklen eng miteinander verknüpft. Die Aktivität ist dabei das Host-Element aller Unterelemente und stellt immer ein neues Fenster da.

## Activity on Startup

Standardmäßig wird beim Starten der App die MainActivity ausgeführt, die dann wiederum die UI befüllt. Da unsere App jedoch einen Account benötigt, indem die Ergebnisse gespeichert werden, soll die Login-Seite jene sein, die zuerst geladen werden soll. Dazu sind folgende Änderungen im Manifest nötig:



Mit dem <intent-filter> wird dem Betriebssystem mitgeteilt, wie es auf eine bestimmte Aktion reagieren soll. In diesem Fall wird unter <action> definiert, dass es sich bei der „LoginActivity“ um die Hauptaktivität handelt, und unter <category>, dass ein Launcher der App erzeugt wird, mit dem man die App starten kann.

# Fragments

Wie bereits erwähnt handelt es sich bei Fragments um eine weitere wichtige Komponente. Fragments sind modulare Bestandteile, die wenn benötigt, häufiger im Projekt verwendet werden können. Gehostet werden Fragmente in „FragmentContainerView“

## Probleme

Nach dem Erstellen der Fragments entstand das Problem, dass diese zu klein ausgefallen sind. So wurden anfangs einzelne Fragmente in der Login-Page für die Buttons und den User-Input erstellt. Dies hatte den Vorteil, dass die einzelnen Elemente wiederverwendet werden können, allerdings auch den Nachteil, dass es nicht möglich war Daten zwischen den einzelnen Fragmenten auszutauschen. Daher bestand die Lösung darin, nur zwei Fragmente für die Login-Page und Register-Page zu erstellen, da hier keine Daten zwischen den Klassen fließen.

# Technologien

## Sprachen / IDE / Architektur / Build-Tool

Programmiersprache: Kotlin

IDE: Android Studio

Architektur: MVVM

Build-Tool: Gradle Kotlin DSL

## Design

Designsystem: Material Design 3 Dark

Hintergrundbilder: erstellt mit der Platform haikei.app

## Libraries / Dependencys

## Gradle (SDKs/JDKs)

### Probleme

Dadurch, dass wir das Projekt über GitHub über mehrere Geräte synchronisieren und verändern, entstand folgendes Problem:

A screenshot of a computer error

Description automatically generated

Da nun die Pfade nicht mehr zusammenpassen, müssen diese geändert werden, damit die JDKs vom Projekt gefunden und verwendet werden können. Dazu war es nötig folgende Systemvariable bei Windows hinzuzufügen:

Zusätzlich musste die Systemvariable ANDROID\_HOME gesetzt werden, welche üblicherweise automatisch bei der Installation von Android Studio gesetzt werden sollte:



## Build-Time

# Optimierung

Im Verlauf des Projekts, nach Abschluss der Implementierung der Galerie, stellte sich heraus, dass das Laden der Bilder aus dem Speicher erheblich Zeit beanspruchte und den Haupt-Thread blockierte. Ein ähnliches Problem trat auch bei den API-Aufrufen auf. Um dieses Problem zu beheben und die Performance zu verbessern, wurden Threads implementiert. Des Weiteren wird Proguard verwendet, da dies eine weitere Möglichkeit bietet, um Optimierungen vorzunehmen.

1. **Threads**

Zur Auslagerung von IO-Aufgaben, sowie API-Aufrufen kommt der IO-Tread zum Einsatz, der wie unteren Beispiel definiert ist. Weitere Threads werden im Sinne der Architektur ausschließlich in ViewModels erstellt.



1. **Proguard**

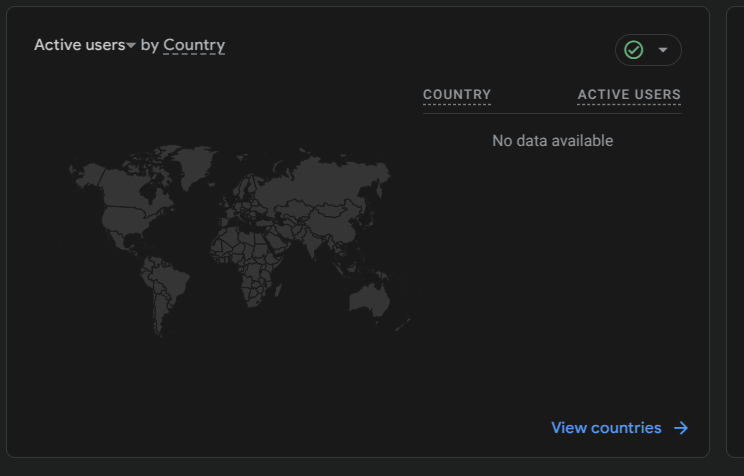
Proguard bietet die Möglichkeit, die Größe der Applikation zu reduzieren und die Performance zu verbessern, indem die Bytecodes optimiert werden. Das bedeutet Proguard entfernt automatisch redundanten Code und verwendet Inlinig: Code wird an der Stelle direkt eingefügt, an der sie aufgerufen wird. Dies reduziert den Overhead des Methodenaufrufs, während die Wartbarkeit und Lesbarkeit des Codes unverändert bleiben.

Ebenso erschwert es Reverse Engineering, indem es den Code verschleiert: Methoden, Klassen und Variablen werden so verändert, so dass ihr ursprünglicher Zweck nicht mehr erkennbar ist.



1. **Analytics**

Neben der programmatischen Optimierung bietet die Analyse des Nutzerverhaltens eine wertvolle Möglichkeit, die Performance weiter zu verbessern. **Firebase Analytics** ermöglicht die Erfassung und Auswertung verschiedener Daten, darunter **App-Interaktionen** sowie **das Land, aus dem die App genutzt wird**. Die Herkunft der Nutzer ist beispielsweise ein wichtiger Faktor, da sie Aufschluss über regionale Unterschiede im Nutzungsverhalten gibt. Beispielsweise können **unterschiedliche Netzgeschwindigkeiten** in verschiedenen Ländern die Ladezeiten beeinflussen, was gezielte Optimierungen wie **reduzierte Bildgrößen** erforderlich machen könnte.



# Kamera – erste Herangehensweise

Auswahl:

* CameraX
* Camera2
* Camera

Die Wahl viel zuallererst auf: Camera

Android liefert bereits eine Kamera, zu der der Benutzer vollen Zugriff hat. Da die Kamera-Funktionen, die von unserer App verwendet werden, auf Bilder beschränkt sind, ist keine komplexere Library wie CameraX nötig, da diese zusätzlich Videos und Audios aufnehmen kann. Ein Umstieg auf die komplexere CameraX Library wäre jedoch zu einem späteren Zeitpunkt nützlich, da sie zusätzlich Funktionen zur Bildanalyse bereitstellt, die nützlich sind, sollten wir das KI-Modell direkt am Android Device bereitstellen. So bietet CameraX Funktionen direkt zur Übergabe an ML Kit, womit wir mit TensorFlow Lite oder PyTorch Edge unser eigenes Modell integrieren und einbinden können.

## Funktionsweise

Bevor die Kamera geöffnet wird, wird überprüft, ob die erforderliche Berechtigung zur Nutzung der Kamera erteilt wurde. Wenn die Berechtigung noch nicht gewährt wurde, wird eine Anfrage zur Erteilung der Kamera-Berechtigung gesendet. Sobald die erforderlichen Rechte erteilt werden, wird die Methode „openCamera()“ aufgerufen, in der ein Intent erstellt wird.



„MediaStore.ACTION\_IMAGE\_CAPTURE“ gibt dabei an, dass ein Foto gemacht werden soll. Anschließend wird überprüft, ob eine Kamera-App existiert und anschließend wird die Aktivität mit „startActivityForResult()“ gestartet.

Nachdem der Benutzer das Foto geschossen und akzeptiert hat, wird der Benutzer in die Methode „onActivityResult()“ geleitet, in der das Foto via der Klasse „Storage“ gespeichert wird.



## Probleme

Beim Schießen der Fotos fiel auf, dass die Auflösung (256px\*192px) nicht hoch genug war, um potenzielle Hautläsionen zu erkennen oder angemessen zu bewerten. Grund dafür war, dass „MediaStore.*ACTION\_IMAGE\_CAPTURE*“ nur das Thumbnail zurückgibt.



Dieses Problem konnte mit der Implementierung eines „File Providers“ gelöst werden, der erneut im Manifest registriert werden muss.

# File Provider

Diese Komponente ist essentiell für die Speicherung der Bilder im hochauflösenden Format (4096px\*3072px). Dazu ist eine Implementierung im Manifest nötig, die folgendermaßen aussieht:



Die Bezeichnung des Providers wird mit dem „android:authorities“-Tag spezifiziert.

Zudem war die Implementierung der Datei „file\_paths.xml“ nötig, welche dem File Provider das Verzeichnis preisgibt, auf das er zugreifen darf. In diesem Fall wird, wie bereits erwähnt im Applikations-Verzeichnis:

“/storage/emulated/0/Android/data/com.example.dermaai\_android\_140”

ein Unterordner “Pictures” angelegt.



Durch die Implementierung des File Providers wird das hochauflösende Bild direkt im externen Speicher abgelegt, anstatt auf das niedrigaufgelöste Thumbnail zurückzugreifen.

# Kamera – finale Lösung

Im Laufe des Projekts fiel auf, dass es sinnvoll ist, eine grafische Hilfe einzubauen, die dem Benutzer hilft, ein zentriertes Bild der Hautlässion zu schießen. Da bei der vorherigen Implementierung, dies nicht möglich war, da dort auf die Kamera-App verwiesen wurde, musste zu CameraX gewechselt werden. Das Layout sieht dabei folgendermaßen aus:

Ein Bild, das Screenshot, Kreis, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

Um nun CameraX zu verwenden, muss der CameraProvider initialisiert werden, und an den Lifecycle gebunden werden, um anschließend eine Preview anzuzeigen.



Die Preview, sowie weitere Konfigurationen werden in folgender Funktion eingerichtet. Bei Preview handelt es sich im Code um eine PreviewView, die auf das XML-Element verweist. Im ersten Use-Case im Code werden die von der Kamera aufgenommenen Bilder auf diese View in Echtzeit übertragen. Beim zweiten Use-Case wird die Kamera näher konfiguriert. Aufgrund der Anforderungen der KI, wurde festgelegt, dass die Qualität im Vordergrund steht (CAPTURE\_MODE\_MAXIMIZE\_QUALITY) als eine geringere Latenz (*CAPTURE\_MODE\_MINIMIZE\_LATENCY).* Anschließend wird konfiguriert, dass ausschließlich die Hinterkamera zum Fotoschießen verwendet werden soll (LENS\_FACING\_BACK).

Im try-catch Block werden vorsichtshalber alle Use-Cases vom CameraProvider entfernt, bevor die gerade konfigurierten Use-Cases letztlich an das Objekt gebunden werden, um Exceptions zu vermeiden.



# Storage

## Speichern von Bildern

Alle für die zur Speicherung benötigten Methoden befinden sich in der Klasse „Storage“. Diese Klasse ist dafür zuständig, Bilder, die mit der Kamera aufgenommen wurden zu verwalten und zu speichern.

Der Speicherort für die Bilder, die der Benutzer schießt, ist folgendermaßen definiert: “/storage/emulated/0/Android/data/com.example.dermaai\_android\_140/files/Pictures/Photo\_User/<filename>.jpg”

# Ergebnisse

Das entwickelte Android-Projekt zielt darauf ab, den Benutzern eine effiziente und sichere Möglichkeit zur Aufnahme und Verarbeitung von Bildern zu bieten. Die App erfüllt ihre Hauptfunktionen wie die Bildaufnahme, -übertragung, und -anzeige sowie Maßnahmen für die Sicherheit und der Benutzerfreundlichkeit. Im Verlauf des Projekts traten verschiedene Herausforderungen auf, die jedoch erfolgreich gemeistert wurden:

1. **Neue Programmiersprache und Entwicklungsumgebung:**

Da ich zum ersten Mal mit Android und Kotlin gearbeitet habe, war die Einarbeitung in die Android-Entwicklung sowie die Programmiersprachen eine Herausforderung. Die Anpassung an die spezifischen Anforderungen der mobilen App-Entwicklung, wie z. B. das Lifecycle-Management und die effiziente Nutzung von Ressourcen, war anfangs ungewohnt. Allerdings konnte ich diese Hürden mit zunehmender Erfahrung erfolgreich überwinden.

1. **Neuankömmling im Bereich Android-Entwicklung:**

Als Anfänger in der mobilen Entwicklung war es eine spannende Herausforderung, alle nötigen Konzepte und Best Practices in Bezug auf Android-Architektur, UI-Design und Nutzerinteraktion zu erlernen. Besonders die Kommunikation zwischen den Komponenten (ViewModel, Model, und View) waren neue Konzepte, die jedoch im Laufe des Projekts gut gemeistert wurden.

# Ausblick

Aufgrund der strengen Einhaltung der Architektur, sollte es ein leichtes Spiel sein, die App zu einem späteren Zeitpunkt weiterzuentwickeln. Mögliche Arbeitspakete umfassen dabei folgende:

**Integrierte KI**: Eine mögliche Erweiterung könnte die Integration der KI direkt in die App sein. Dies könnte mit ML-Kit umgesetzt werden.

**Benutzeranalyse**:  
Um die App sowie die Genauigkeit der Ergebnisse weiter zu optimieren und die Benutzererfahrung zu verbessern, könnte ein integriertes Analyse-System entwickelt werden. Dies könnte Nutzerdaten sammeln, die dabei helfen die App, sowie die KI weiter zu verbessern. Zudem könnte dieses System uns dabei helfen zu erkennen, welche Fehler Benutzer beim Schießen der Fotos machen, und Tipps bezüglich deren Verbesserung geben.

# Quelle

https://android.googlesource.com/platform/sdk/+/refs/heads/main/files/proguard-android-optimize.txt