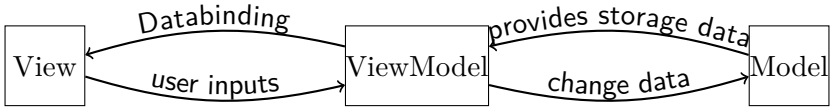
Architektur

Von
Felix Winkler
Florian Vierkant
Marie Biethahn
Max Henning Junghans
Sebastian Guhl
Steffen Marbach

In diesem Dokument wird die Softwarearchitektur des Ortungssimulator Projekts, vorgestellt und erklärt.

Inhaltsverzeichnis

1 Die MVVM Architektur



1.1 Erklärung der Architektur

Die Model View ViewModel (MVVM) Architektur ist eine weitverbreitete Architektur, die eine Software grob in 3 Schichten unterteilt: Views, ViewModels und Models.
Die Models enthalten die Daten, die am Ende vom ViewModel modifiziert werden können. Die Daten der Models sind unabhängig von den anderen Schichten und können daher von unterschiedlichen View Implementierungen verwendet werden. Es kann auch Geschäftslogik in den Models liegen, wie das Einlesen von Daten aus dem Speicher.
Die ViewModels nehmen Eingaben der UI über die Views entgegennehmen und modifizieren ggf. die Daten der Models. Sie bieten den Views einen State der Models, auf den die Views bei Änderungen ohne weiteres Eingreifen reagieren können und die UI verändern.
Die Views reagieren auf Änderungen der ViewModels und kümmern sich um die Darstellung der Daten im User Interface. Die Views enthalten keine Geschäftslogik.

1.2 Warum haben wir uns für die MVVM Architektur entschieden

Für die Darstellung im User Interface nutzen wir Jetpack Compose, welches dazu ausgelegt ist, mit einem ViewModel zu arbeiten. Google empfiehlt die Nutzung eines ViewModels in Zusammenhang mit Jetpack Compose.

> **Jetpack Compose**
>
> When using Jetpack Compose, ViewModel is the primary means of exposing screen UI state to your composables. In a hybrid

Quelle: Android Developer Documentation [1]

Weitere Gründe sind, durch die klare Trennung der unterschiedlichen Schichten kann die Geschäftslogik unabhängig von der Darstellungslogik bearbeitet werden. Dadurch erhöht sich die Testbarkeit des Projekts, da wir die Businesslogik ohne UI implementieren und daher einfach testen können. Komplexe UI Tests sind daher für das Testen der Geschäftslogik nicht nötig.

1.3 Data Binding

Um die UI mit Daten versorgen zu können, verwenden wir die in Jetpack Compose integrierten States. Die States enthalten die Daten, die die jeweilige View anzeigen soll.

Das State selbst wird in dem zugehörigen ViewModel erzeugt und der View bereitgestellt. Das ViewModel ändert den State und die View reagiert automatisch darauf. Die Aktualisierungslogik müssen wir nicht selbst entwerfen, Jetpack Compose kümmert sich darum.

2 Ressourcen zur Einarbeitung

Wir empfehlen allen, die sich in unserer Architektur einarbeiten wollen, zusätzlich zu diesem Dokument folgendes Video `https://www.youtube.com/watch?v=8YPXv7xKh2w`. Dort werden detailliert alle Inhalte der Architektur erklärt. Wir haben uns zum größten Teil an der Architektur des Beispiels aus dem Video orientiert, sodass das Video gut zum Erreichen eines vertieften Verständnisses unserer Architektur geeignet ist.

3 Projektstruktur

3.1 Ordnerstruktur

data

Hier befindet sich die Implementierung der Datenbank. Zusätzlich findet man hier auch die StorageManager.

di

Dieser Ordner besteht derzeit nur aus dem AppModule für die Data Injection.

domain

In dem Ordner domain findet man die Klassen des Models. Darunter die Klasse Sound und Vibration, sowie ihre Superklasse und der zugehörige Converter für die JSON-Strings. Auch die Datenbank Schnittstellen befinden sich hier, wie die UseCases der Datenbank.

presentation

In dem presentation Ordner bekommt jeder Screen einen Unterordner(select, run, edit ...). Jeder dieser Unterordner enthält die zum Screen zugehörigen ViewModel, View und die Events, die die View auslöst. Bei Bedarf kann ein weiterer Unter-Ordner 'components' erstellt werden, hier können zugehörige compose Klassen abgelegt werden. Im presentation-Ordner befindet sich außerdem ein weiterer Unter-Ordner 'universalComponents', hier können compose-Klassen abgelegt werden, die in mehreren Views verwendet werden.

ui

Der ui-Ordner enthält das Theme der App und speichert die verwendeten Farben für den Dark- und Light-Mode.

3.2 Events

Jeder Screen-Ordner in dem presentations Ordner hat eine Event-Klasse. Diese Events werden bei der Kommunikation von View zu ViewModel benutzt. Die Funktion 'onEvent' der ViewModel erwartet ein Event von der View. in den Event Klassen ist die Kommunikation von View zu ViewModel beschrieben, hier kann man sich ein Überblick der Funktionen des Screens schaffen.

3.3 Storage Manager

Die Storage Manager bieten eine Schnittstelle zum internen Speicher des Geräts und können Daten aus dem Speicher lesen und schreiben.

Configuration

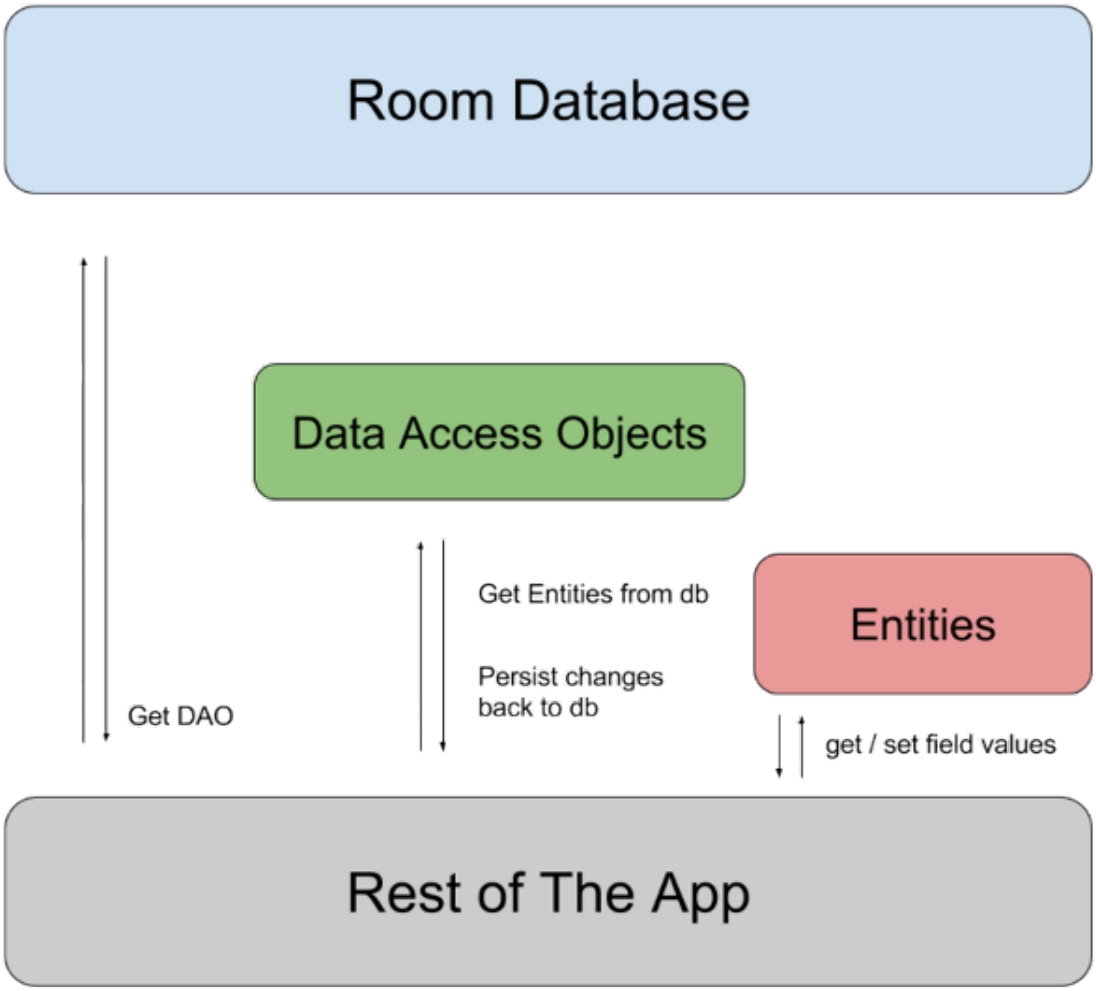
Der 'ConfigurationStorageManager' speichert und lädt Konfigurationen aus dem Speicher. Hier wurde die Import- und Exportfunktion der App implementiert. Dafür werden JSON-Strings benutzt, diese werden hier erstellt und umgeformt

Sound

Der 'SoundStorageManager' kann Sounds aus dem Speicher lesen und schreiben. Dafür wird der private Ordner der App benutzt. Auf diesen Ordner hat der User kein Zugriff.

3.4 Datenbank

Für das Speichern der Konfigurationen nutzen wir eine SQL-Lite Datenbank, die von Androids Room bereitgestellt wird. Unsere Datenbank beschränkt sich auf eine Tabelle mit einer Entity.



Quelle: Android Developer Documentation [2]

In domain/model/Configuration.kt findet man die Entity der Configuration:

```
@Entity
data class Configuration(
    val name: String,
    val description: String,
    val randomOrderPlayback: Boolean,
    val components: List<ConfigComponent>,
    val isFavorite: Boolean = false,
    @PrimaryKey val id: Int? = null
){
...
}
```

Der Primary Key bzw. die ID wird mit @PrimaryKey gekennzeichnet, so übernimmt Room die Zuweisungen der IDs automatisch. 'components' bildet eine Liste aus Vibrationen und Sounds. Alle Felder müssen als 'val' gekennzeichnet werden, so muss bei Änderungen eines Eintrags in der Datenbank die ganze Entity neu zugewiesen werden. Dieses Vorgehen ist von Jetpack Compose festgelegt, so können Änderungen an der Datenbank automatisch von Jetpack Compose erkannt und die UI entsprechend angepasst werden.

In data/source/ConfigurationDao findet man das Data Access Object. Hier finden die Abfragen der Datenbank mit üblichen SQL-Befehlen statt. Zum Beispiel:

```
@Query("SELECT * FROM configuration WHERE id = :id")
suspend fun getConfigurationById(id: Int): Configuration?
```

Dieses Dao bietet eine Schnittstelle zwischen App und Datenbank. Sämtliche Interaktion mit der Datenbank findet über die Funktionen in dieser Klasse statt.

Neue Einträge und Änderungen an der Datenbank werden durch folgende Funktion realisiert:

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insertConfiguration(configuration: Configuration)
```

Bereits bestehende Konfigurationen, also Konfigurationen, die die gleiche ID haben, stellen ein Konflikt dar. Durch den Parameter OnConflictStrategy.REPLACE wird die entsprechende Konfiguration mit der gleichen ID gefunden und ersetzt.

Den ViewModels wird über die Data-Injection die configurationUseCases übergeben. Über diese kommunizieren die ViewModels mit der Datenbank. Die Implementierung der UseCases findet man unter: domain/useCase. In diesen Klassen kann weitere Logik implementiert und z. B. eine Exception geworfen werden, wenn versucht wird, unerwünschte Einträge in die Datenbank zu laden.

Data-Injection

Die Data Injection wird von Androids Dagger-Hilt übernommen [3]. Implementiert wurde die Logik hier: di/AppModule.kt. Durch dieses Vorgehen bekommen die ViewModels automatisch Zugriff auf die Datenbank über die UseCases. So muss keine Initialisierung und laden der Datenbank, sowie Übergabe an die ViewModels von uns erstellt werden. Einbinden kann man dies, indem man das ViewModel mit @HiltViewModel kennzeichnet und anschließend mit @Inject constructor(...) die UseCases übergibt. Hier gezeigt am Beispiel vom SelectViewModel:

```
@HiltViewModel
class SelectViewModel @Inject constructor(
    private val configurationUseCases: ConfigurationUseCases
) : ViewModel() {
...
```

```
}
```

Dabei sollte man beachten, dass in der View das ViewModel als hiltViewModel() erzeugt werden muss:

```
fun SelectScreen(
    viewModel: SelectViewModel = hiltViewModel(),
    ...
) {
...
}
```

3.5 Models

Die zugehörigen Klassen des Models sind zu finden unter: domain/model. Die Verwendung der Klassen wird im Folgenden beschrieben.

Vibration und Sound

Die Datenklassen Vibration und Sound beinhaltet alle Informationen, die für Nutzung und Einstellen in der Timeline notwendig sind, zusätzlich eine id. Die pro Konfiguration einmalige ID wird verwendet, da ein Vergleich in Kotlin(===) zwei Komponenten als gleich bewertet hat, wenn diese den gleichen Inhalt haben, aber unterschiedlich sind. So wurden z. B. zwei Vibrationen, die die gleiche Stärke, Pause, Duration und Name hatten als gleich angesehen. Dadurch war ein Verschieben in der Timeline nicht möglich.

Beide Klassen bieten eine Funktion 'myCopy(...)'. Diese Funktion gibt eine Kopie des Objekts aus, dabei können Parameter übergeben werden, die Eigenschaften des ausgegebenen Objekts verändern. Dies war nötig, um Änderungen in einem Sound oder Vibration vorzunehmen und werden für die Slider im 'EditTimelineViewModel' benutzt

ConfigComponent

Die Klasse 'ConfigComponent' ist die Superklasse von Vibration und Sound. Diese wurde eingeführt, um eine Liste aus ConfigComponent zu erstellen, diese bildet die Timeline.

Configuration

Die Klasse Configuration wurde im Kapitel 'Datenbanken' angesprochen. Diese Klasse ist eine Entity und ihr Inhalt wird als Eintrag in der Datenbank benutzt. Unter weiteren Einstellungsmöglichkeiten, wie name und description wird hier die Liste aus ConfigComponent gespeichert, welche die Timeline Abbildet.

Converter

Sound Der Sound Converter kann ein aus einer Sounddatei eingelesenes ByteArray in ein String umwandeln, indem es das Base 64 Encoding nutzt. Ebenso ein Base64-String

in ein ByteArray. Dieser Converter wird benutzt für das Importieren und Exportieren von Konfigurationen als txt-Datei.

Configuration Component Dieser Converter wandelt die Liste aus 'ConfigComponent' in ein JSON-String um, sowie auch ein JSON-String in die Liste. Dies wurde für das Importieren und Exportieren von Konfigurationen als Txt-Datei benutzt. Da ein ConfigComponent nur eine Superklasse ist, wurden die Klassen als @Serializable gekennzeichnet und die Vibration und Sound Klasse ein 'SerialName' bekommen. Durch die Kennzeichnung wird die Convertierung in/von JSON automatisch abgeschlossen.

Range Die Serviceklasse benötigt für das Abspielen der Vibrationen und Sounds Wertebereiche, die nicht nutzerfreundlich sind. Der Converter wandelt die in den Slidern benutzten Intervalle in die für den Service benötigten Werte um.

3.6 JSON

Als Format für das Speichern von Konfigurationen haben wir JSON festgelegt. JSON wird von Kotlin unterstützt, indem die erforderlichen Funktionalitäten gestellt werden. Der JSON String wird im ConfigurationStorageManager zusammen mit den Converter Klassen umgewandelt. Dafür wurde die Serialisierbarkeit benutzt und Sounds als Base64-String gespeichert, mehr dazu im Kapitel Converter.

3.7 ViewModel

Die ViewModels dienen als Bindeglied zwischen Views und Models/ Datenbank. Jeder Screen hat ein ViewModel, diese werden von der View im Constructor erstellt. Die ViewModel bieten der View einen State, mit den Daten, die die View anzuzeigen hat. Außerdem findet man in den ViewModels die Geschäftslogik. Die entsprechenden Funktionen lassen sich mittels Events von der View aufrufen. Änderungen, die vorgenommen wurden, werden im State gespeichert, die View kann daraufhin auf die Änderungen automatisch reagieren.

3.8 View

Die Views werden in der MainActivity durch den NavController erstellt. Die View erzeugt darauf hin ihr ViewModel im Constructor. Vom ViewModel bekommt sie einen State, diese Daten werden in der View dargestellt. Sämtliche Geschäftslogik wird an das ViewModel abgegeben, in dem ein Event an das ViewModel mit den passenden Daten gesendet wird. Dies wird z. B. bei Klick auf ein Button verwendet.

Jetpack Compose

Wir haben uns in diesem Projekt für Jetpack Compose entschieden. Dieses ist gängiger Standard in Android App Projekten und passt perfekt in die MVVM Architektur. Mit

Jetpack Compose konnten wir auch automatisierte User Story Tests durchführen, die Klicks automatisch ausführen kann.

3.9 Infinity Service

Zum Abspielen der Konfigurationen wird ein Service verwendet. Dieser wird mittels eines Intent gestartet, durch das auch der Dateipfad des Ordners, in dem die Sounddateien gespeichert sind, und die abzuspielende Konfiguration als json String kodiert übergeben werden. Auch zum Beenden des Service wird ein entsprechendes Intent verwendet. Während der Service die übergebene Konfiguration abspielt, wird in dem Service ein Wake Lock verwendet, um sicherzustellen, dass der Service nicht davon beeinflusst wird, wenn das System in den Doze-Modus wechselt. Außerdem wird beim Starten des Service ein Flag gesetzt, das dafür sorgt, dass der Bildschirm sich nicht mehr automatisch deaktiviert, da ein ausgeschalteter Bildschirm vereinzelt bei Geräten zu Problemen führen kann. Sollte dies auf dem verwendeten Gerät nicht der Fall sein, kann der Screen einfach beim Starten des Service manuell ausgeschaltet werden. Um Konflikte mit den Battery-Optimizern des Systems zu vermeiden, wird der User im HomeScreen darum gebeten, die App von der Nutzung der Battery-Optimizer auszuschließen. Dies ist nötig, da es ansonsten auf verschiedenen Geräten das Problem gibt, dass diese den Service als unnötigen Energieverbrauch registrieren und vorzeitig beenden.

3.10 Navigation

Die Logik der Navigation findet man in der MainActivity. Hier konnten wir jedem Screen einen Namen geben und zu diesem von jeder Stelle aus navigieren, indem der NavController an die Views übergeben wurde. Mit der Navigation können auch Daten zwischen Screens ausgetauscht werden, z. B. teilt der Select Screen dem run Screen mit, welche Konfiguration der User ausgewählt hat.

3.11 Exceptions

Exceptions waren meist nicht nötig, weil der UserInput über Slider geschieht. Bei ungültigen Nutzereingaben wurden Buttons als inaktiv markiert. Zusätzlich Strings, die vom User eingegeben werden können, um z. B. Name und Beschreibung zu ändern. Hier haben wir den Usern volle Freiheit gelassen, diese so zu benennen wie sie wollen oder sogar leer lassen. Den Fall des Leerlassens möchte man möglicherweise in zukünftigen Versionen abfangen. Dafür gibt es eine Exception in den UseCases der Datenbank bzw. in der Klasse Configuration, deren Funktion einfach implementiert werden kann.

4 Automatische Tests

Für eine ausführliche Einarbeitung bietet sich folgendes Video an: `https://www.youtube.com/watch?v=nDCCwyS0_MQ`. Dort werden die Unit und End-to-End Tests an einer Beispielapp ausführlich aufgesetzt und erklärt. Die Architektur der Beispielapp ist auch

nahezu identisch mit unserer umgesetzten MVVM-Architektur, sodass sich die Inhalte sehr einfach übertragen lassen.

4.1 Unit-Tests

Die Unit-Tests werden mit JUnit 4 durchgeführt und ermöglichen ein isoliertes Testen einzelner Klassen und Methoden. In unserem Projekt wurden sie vor allem eingesetzt, um Konvertierungs- und Speicherfunktionen zu testen, da die Anwendung sonst sehr UI-dominant ist und es wenig komplizierte Modell-Logik gibt, die isoliert getestet werden muss. Die Unit-Tests befinden sich in der Projektstruktur unter *app/java/-com.ispgr5.locationsimulator(test)*

4.2 End-to-End Tests

End-to-End Tests simulieren eine Benutzereingabe, die dann auf einem Emulator oder Smartphone automatisch ausgeführt und getestet wird. Wir hab dieses benutzt, um unsere User-Story-Tests so weit es sinnvoll ist, automatisch zu testen (Siehe die User-Story-Test-Dokumentation für weitere Informationen). Die End-to-End Tests befinden sich in der Projektstruktur unter *app/java/com.ispgr5.locationsimulator(androidTest)*

Initiales Set-Up

Beim initialen Set-up der End-to-End Tests wurde ein neues TestAppModule hinzugefügt (*app/java/com.ispgr5.locationsimulator(androidTest)/di/TestAppModule*), welches im Unterschied zum AppModule eine In-Memory-Database verwendet, sodass beim Testen keine langfristigen Daten in Datenbanken abgelegt werden, die Funktion der Datenbank jedoch realitätsnah getestet werden kann. Des Weiteren musste zum Laufen ein HiltTestRunner erstellt werden *app/java/com.ispgr5.locationsimulator(androidTest)/di/HiltTestRunner*. Diese Schritte mussten nur einmalig durchgeführt werden.

Test-Klassen

Jede Testklasse für End-to-End Test muss folgende Annotationen enthalten:

- @HiltAndroidTest, damit die richtigen Hilt-Injections automatisch gesetzt werden.
- @UninstallModules(AppModule::class), damit dat TestAppModule, statt dem Test-Module verwendet wird.

Des Weiteren muss eine Hilt-Rule und Compose-Rule gesetzt werden und es sollte in einer SetUp Methode die Hilt-Rule mit *hiltRule.inject()* injected werden und der Compose Content muss gesetzt werden. Dies ist beispielsweise in der Klasse *com.ispgr5.locationsimulator.UserStoryTes* zu finden und kann so für weitere Klassen übernommen werden.

Test-Methoden

Die Compose-Rule kann verwendet werden, um Aktionen und Tests auf Compose-Elementen anzuwenden. Zum Identifizieren von Compose-Elementen werden Test-Tags verwendet, die bei den meisten Compose-Elementen über *Modifier.testTag(tagString)* gesetzt werden. Die TestTags werden als Konstanten in der Klasse *com.ispgr5.locationsimulator.core.util.TestTags* gesammelt. So können in den Methoden komplexe Nutzereingaben simuliert werden. Für weitere Informationen siehe die Android-Entwickler Dokumentation [4].

Literatur

[1] `https://developer.android.com/topic/libraries/architecture/viewmodel#jetpack-compose`

[2] `https://developer.android.com/training/data-storage/room`

[3] `https://developer.android.com/training/dependency-injection/hilt-android`

[4] `https://developer.android.com/jetpack/compose/testing`