

Modul 3 – Android App Entwicklung mit Kotlin

Room



Gliederung

- Room
- Database, DAO und Entities
- Beispiel

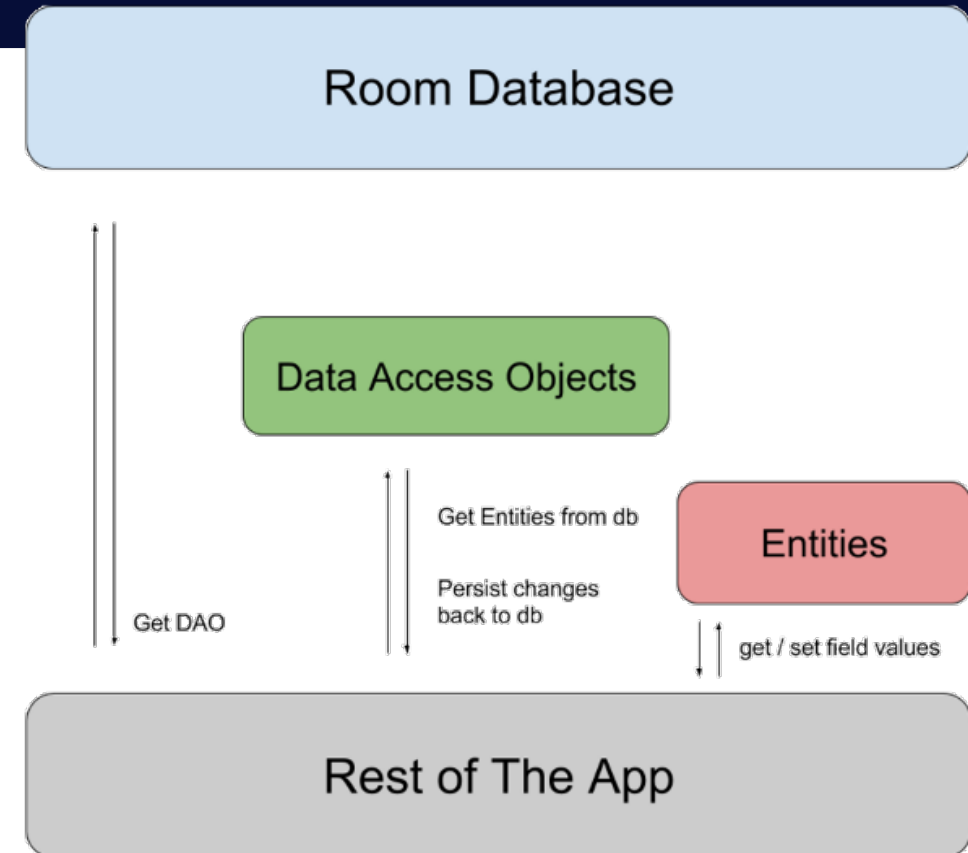


Quelle: <https://fesmag.com/images/stories/2020-08/functional-dry-storage.jpg>

Room

Dient als Abstraktionsschicht und vereinfacht die Kommunikation mit **SQLite**
(ähnlich wie retrofit für API Calls)

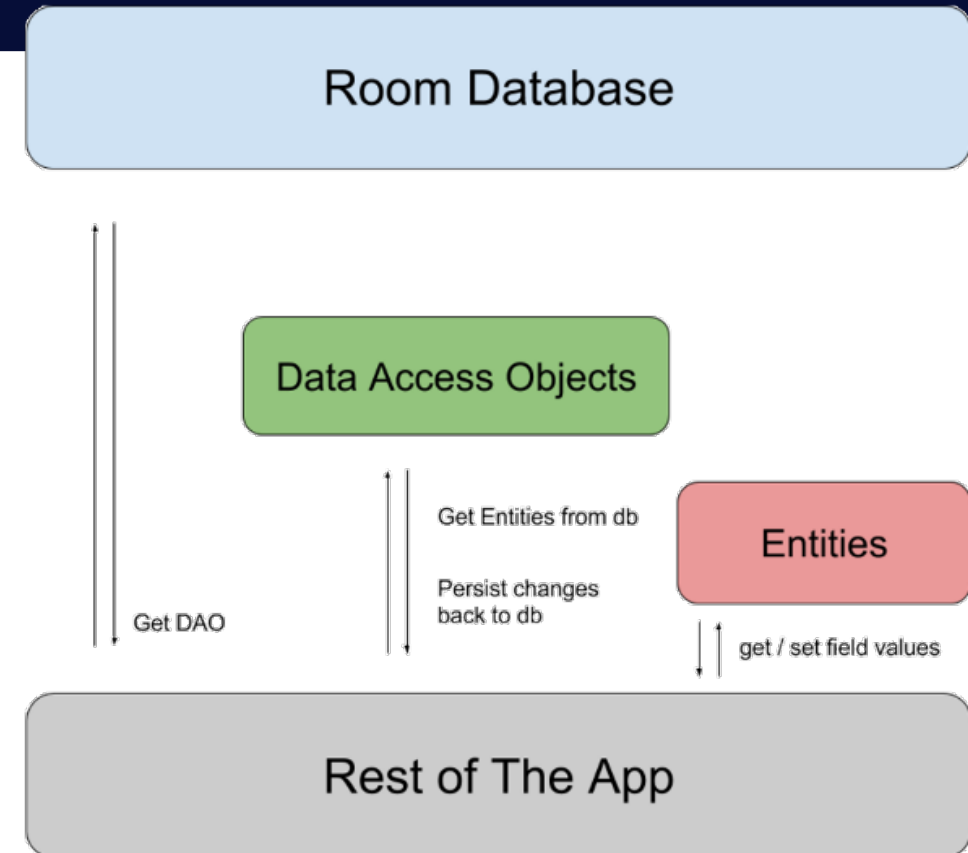
- Überprüft SQL Abfragen
- erspart mittels **Annotations** viel Boilerplate Code
- Optimierte Datenbankmigrationspfade



Quelle: https://developer.android.com/images/training/data-storage/room_architecture.png

Room

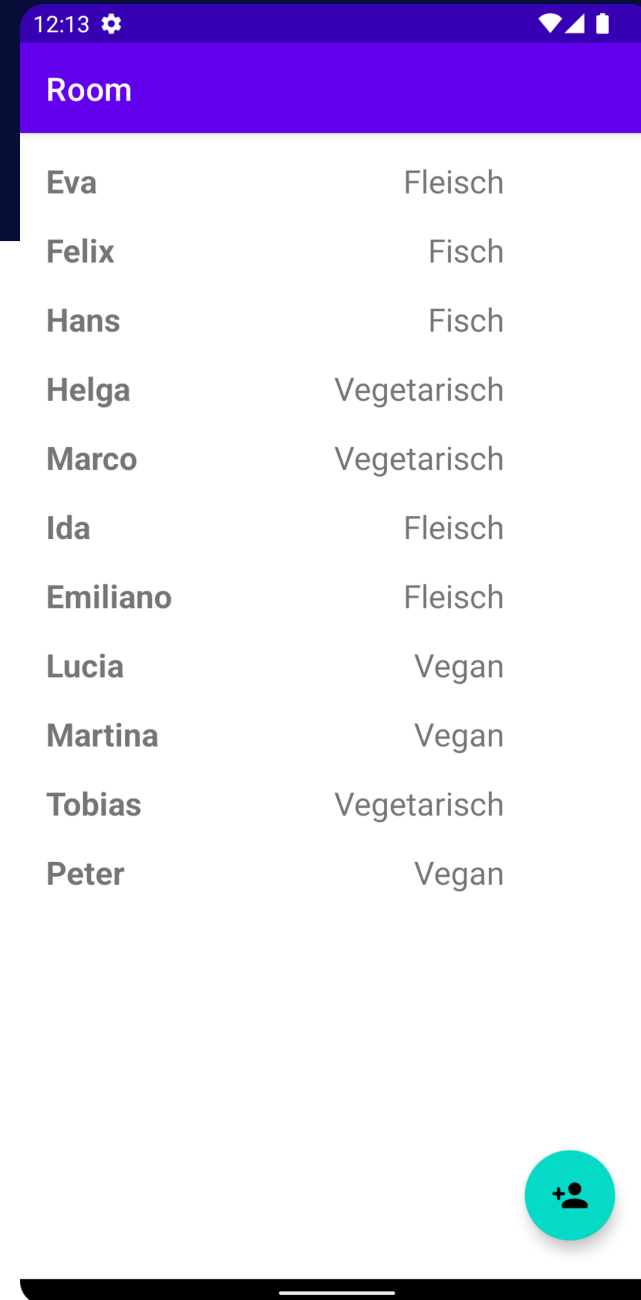
- Datenbank-Klasse
 - enthält die Datenbank und dient als Hauptzugriffspunkt
- Entities (Daten Klassen)
 - stellen Tabellen der Datenbank dar
- DatabaseAccessObject (DAO)
 - stellt Methoden für Datenbankoperationen bereit



Quelle: https://developer.android.com/images/training/data-storage/room_architecture.png

Beispiel: Gästeliste

- Daten Klasse - Guest
 - mit Variablen für ID, Namen und bevorzugtem Essen
- Datenbankklasse
 - mit Singletonpattern damit es auch wirklich nur eine Datenbank in der App gibt
- DatabaseAccessObject (DAO)
 - mit Lese- und Schreibfunktionen um mit der Datenbank zu arbeiten



Room	
Eva	Fleisch
Felix	Fisch
Hans	Fisch
Helga	Vegetarisch
Marco	Vegetarisch
Ida	Fleisch
Emiliano	Fleisch
Lucia	Vegan
Martina	Vegan
Tobias	Vegetarisch
Peter	Vegan

build.gradle (App Modul)

ganz oben

In den plugins muss (falls nicht vorhanden)

der AnnotationProcessor **kotlin-kapt** hinzugefügt werden

bei den **dependencies**

werden die Room Libraries hinzugefügt



```
plugins {  
    id 'com.android.application'  
    id 'org.jetbrains.kotlin.android'  
    id 'androidx.navigation.safeargs.kotlin'  
    id 'kotlin-kapt'  
}
```

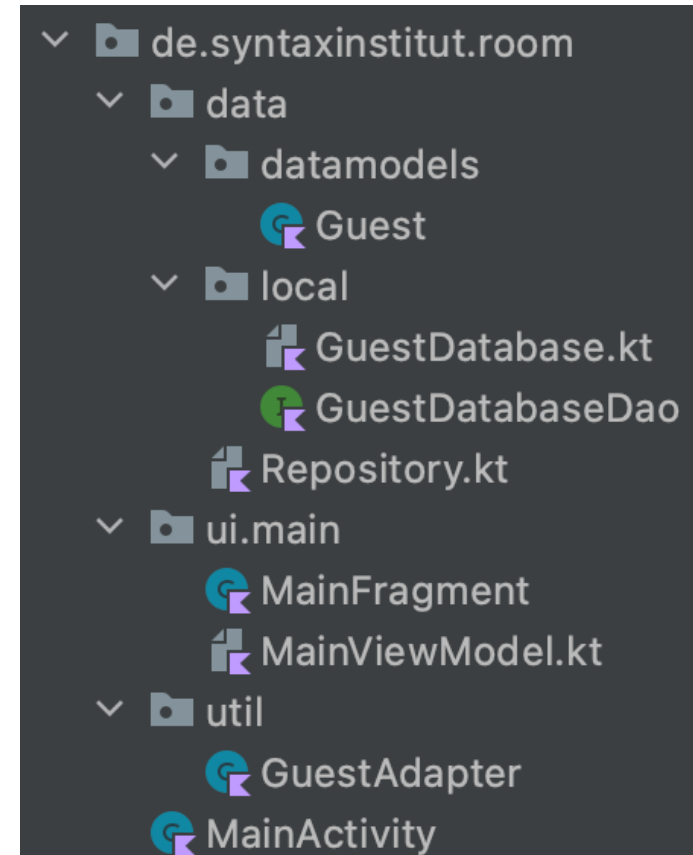
```
// Room Dependencies  
implementation "androidx.room:room-runtime:$room_version"  
kapt "androidx.room:room-compiler:$room_version"  
implementation "androidx.room:room-ktx:$room_version"
```

Struktur

MVVM Struktur

im `data.local` Package befinden sich **Datenbank** und **DAO**

die **Guest Datenklasse** ist wie gewohnt im `datamodels` Package



```
de.syntaxinstitut.room
├── data
│   ├── datamodels
│   │   └── Guest
│   └── local
│       ├── GuestDatabase.kt
│       ├── GuestDatabaseDao
│       └── Repository.kt
├── ui.main
│   ├── MainFragment
│   └── MainViewModel.kt
└── util
    ├── GuestAdapter
    └── MainActivity
```

The screenshot displays the project structure of an Android application. The root package is `de.syntaxinstitut.room`. It contains three main sub-packages: `data`, `ui.main`, and `util`. The `data` package is further divided into `datamodels` and `local`. The `datamodels` package contains the `Guest` class. The `local` package contains the `GuestDatabase.kt`, `GuestDatabaseDao`, and `Repository.kt` files. The `ui.main` package contains the `MainFragment` and `MainViewModel.kt` files. The `util` package contains the `GuestAdapter` and `MainActivity` files.

Guest.kt

Jeder Gast in der Datenbank soll eine automatisch generierte Id als Long, einen **Namen** als String und ein bevorzugtes **Essen** als String besitzen.

@Entity

gibt an, dass aus dieser Klasse eine Tabelle für die Datenbank generiert werden kann

@PrimaryKey(autoGenerate = true)

gibt an, dass die nachfolgende Variable als Primärschlüssel dient

```
@Entity
data class Guest(
    @PrimaryKey(autoGenerate = true)
    val id: Long = 0,
    val name: String,
    val food: String
)
```


GuestDatabaseDao.kt

@Dao

Gibt an, dass es sich um ein `DataAccessObject` handelt

@Query()

Gibt die SQL Anfragen an welche von der Funktion ausgelöst werden soll

@Insert und @Update

Sind von Room vorgefertigte Anfragen zum Einfügen und Aktualisieren

```
@Dao
interface GuestDatabaseDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(guest: Guest)

    @Update
    suspend fun update(guest: Guest)

    @Query(value: "SELECT * FROM Guest")
    fun getAll(): LiveData<List<Guest>>

    @Query(value: "SELECT * from Guest WHERE id = :key")
    fun getById(key: String): LiveData<Guest>

    @Query(value: "DELETE from Guest")
    suspend fun deleteAll()
}
```

GuestDatabase.kt

```
@Database(entities = [Guest::class], version = 1)
```

gibt an dass diese Klasse eine Datenbank ist mit der Tabelle Guest und sich in der Version 1 befindet

die Version muss geändert werden wenn man das Schema der Datenbank ändert (z.B.: eine weitere Klasse(Tabelle) einfügt oder weitere Spalten(Variablen))

guestDatabaseDao

verknüpft die Datenbank mit dem DAO Interface

```
@Database(entities = [Guest::class], version = 1)
abstract class GuestDatabase : RoomDatabase() {

    abstract val guestDatabaseDao: GuestDatabaseDao
}

private lateinit var INSTANCE: GuestDatabase

// if there's no Database a new one is built
fun getDatabase(context: Context): GuestDatabase {
    synchronized(GuestDatabase::class.java) {
        if (!::INSTANCE.isInitialized) {
            INSTANCE = Room.databaseBuilder(
                context.applicationContext,
                GuestDatabase::class.java,
                name: "guest_database"
            ).build()
        }
    }
    return INSTANCE
}
```

GuestDatabase.kt

getDatabase(context: Context)

Erstellt eine neue Datenbank wenn noch keine in `INSTANCE` gespeichert wurde.

Somit wird sichergestellt dass es nur eine einzige Datenbank gibt.

```
@Database(entities = [Guest::class], version = 1)
abstract class GuestDatabase : RoomDatabase() {

    abstract val guestDatabaseDao: GuestDatabaseDao
}

private lateinit var INSTANCE: GuestDatabase

// if there's no Database a new one is built
fun getDatabase(context: Context): GuestDatabase {
    synchronized(GuestDatabase::class.java) {
        if (!::INSTANCE.isInitialized) {
            INSTANCE = Room.databaseBuilder(
                context.applicationContext,
                GuestDatabase::class.java,
                name: "guest_database"
            )
                .build()
        }
    }
    return INSTANCE
}
```

Repository.kt

guestList

Speichert das Ergebnis der `getAll()` Anfrage des DAO

insert(guest: Guest)

Versucht mittels der `insert` Funktion des DAO einen neuen Gast in die Datenbank einzufügen

```
const val TAG = "Repository"

class Repository(private val database: GuestDatabase) {

    val guestList: LiveData<List<Guest>> = database.guestDatabaseDao.getAll()

    suspend fun insert(guest: Guest) {
        try {
            database.guestDatabaseDao.insert(guest)
        } catch (e: Exception) {
            Log.d(TAG, msg: "Failed to insert into Database: $e")
        }
    }
}
```

MainViewModel.kt

`getDatabase(application)`

ruft Datenbank auf und erstellt falls noch keine vorhanden ist eine neue mittels `application(context)`

mit Hilfe der `database` wird ein `repository` erstellt

In der `guestList` Value wird der Wert der `guestList` des `Repositories` gespeichert
(dabei handelt es sich immer noch um `LiveData`)

```
class MainViewModel(application: Application) : AndroidViewModel(application) {  
  
    private val database = getDatabase(application)  
    private val repository = Repository(database)  
  
    val guestList = repository.guestList  
  
    fun insertGuest(guest: Guest) {  
        viewModelScope.launch { this: CoroutineScope  
            repository.insert(guest)  
        }  
    }  
}
```

MainViewModel.kt

`insertGuest(guest: Guest)`

startet die `insert` Funktion des Repositories in einer Coroutine um den UI Thread nicht zu blockieren

```
class MainViewModel(application: Application) : AndroidViewModel(application) {  
  
    private val database = getDatabase(application)  
    private val repository = Repository(database)  
  
    val guestList = repository.guestList  
  
    fun insertGuest(guest: Guest) {  
        viewModelScope.launch { this: CoroutineScope  
            repository.insert(guest)  
        }  
    }  
}
```

MainFragment.kt

Die `guestList` aus dem Repository liefert `LiveData` direkt von der Datenbank und durch `observe` wird bei jeder Änderung auch der Inhalt der `RecyclerView` geändert

Bei klick auf den `addGuestButton` wird ein neuer Gast namens Peter in die Datenbank geschrieben

```
override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

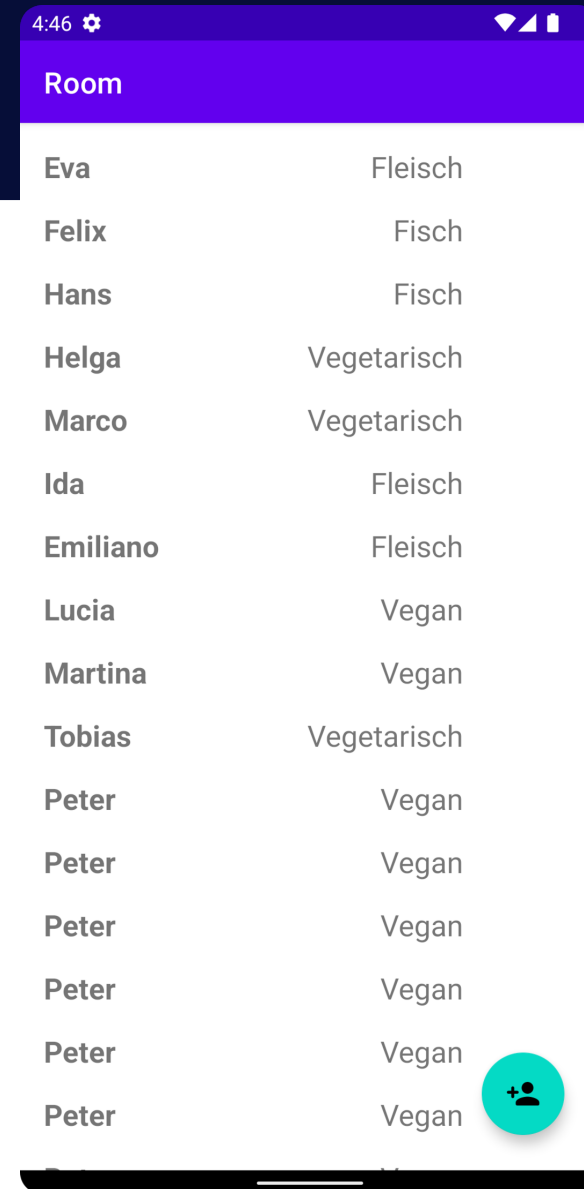
    val recyclerView = binding.guestlist

    viewModel.guestList.observe(
        viewLifecycleOwner,
        Observer { it: List<Guest>!
            recyclerView.adapter = GuestAdapter(it)
        }
    )

    binding.addGuestButton.setOnClickListener { it: View!
        viewModel.insertGuest(
            Guest(
                name = "Peter",
                food = "Vegan"
            )
        )
    }
}
```


Peters

Per Knopfdruck wird ein weiterer Peter in die Datenbank geschrieben und dank LiveData wird die Änderung sofort angezeigt



The screenshot shows an Android app interface with a purple header bar labeled "Room". Below the header is a list of names and their corresponding food preferences. The list is as follows:

Room	
Eva	Fleisch
Felix	Fisch
Hans	Fisch
Helga	Vegetarisch
Marco	Vegetarisch
Ida	Fleisch
Emiliano	Fleisch
Lucia	Vegan
Martina	Vegan
Tobias	Vegetarisch
Peter	Vegan
Peter	Vegan
Peter	Vegan
Peter	Vegan
Peter	Vegan
Peter	Vegan
Peter	Vegan

At the bottom right of the list, there is a red circular button with a white plus sign and a person icon, indicating an option to add a new entry.

Wiederholung

Wiederholung - Was haben wir heute gelernt?

1	Room
2	Database, DAO, Entities
3	Beispiel: Gästeliste

Viel Spaß!



Quelle: <https://cheapandcheerfulcooking.com/veganes-schnitzel-mit-kartoffelsalat/>