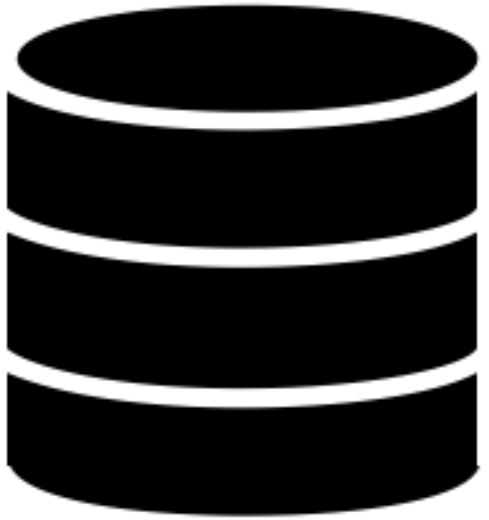
ROOM

Persistance Library

Part of Google Architecture Components

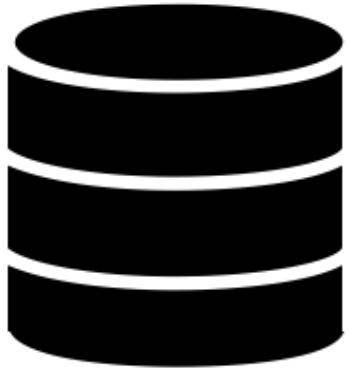
Was ist Room

- Abstraktion über SQLite
- Einfacher Datenbank Zugriff
- Hilft bei der Implementation eines Cache in einer Android Applikation



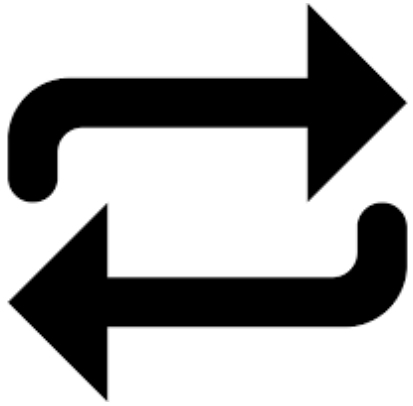
Room Komponenten

Database



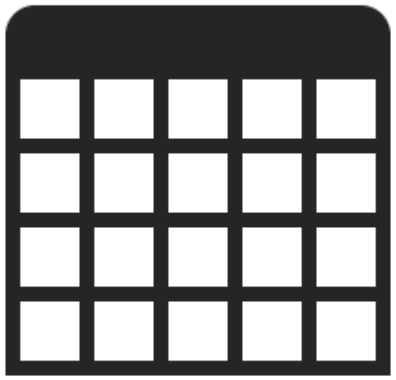
- Haupt-Zugriffspunkt für die Verbindung
- Enthält eine Liste der Entities
- Erweitert “RoomDatabase”-Klasse
- Hat eine Funktion für die Rückgabe eines **DAO**

DAO



Data Access Object enthält Funktionen für Zugriff auf **Entities** und Persistierung

Entity



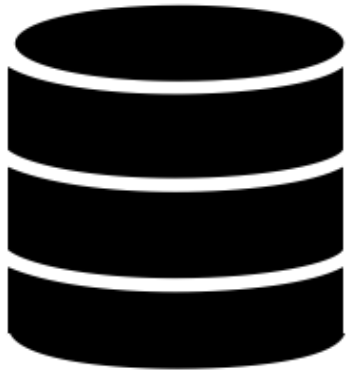
Repräsentiert die Tabelle mit get/set Feldern

Database

Gib die Entities an für die Tabellen

Die Datenbank Version

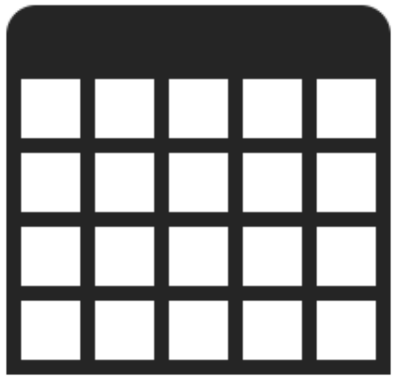
```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase: RoomDatabase() {
    abstract fun userDao() : UserDao
}
```



Abstrakte Funktionen um die DAO’s aufzurufen, werden hier angegeben

Entity

```
@Entity
data class User (
    @PrimaryKey
    private var uid: Int = 0,


    @ColumnInfo(name = "first_name")
    private val firstName: String? = null,


    @ColumnInfo(name = "last_name")
    private val lastName: String? = null
)
```

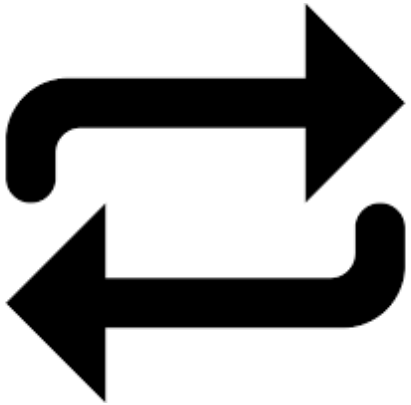


DAO– Data Access Object

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first
AND " + "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertAll(users: List<User>)

    @Delete
    fun delete(user: User)
}
```

Ein Parameter in die Query einfliessen lassen

Optional



Es kann auch ein einzelner User mitgegeben werden

Relationen definieren

```
@Entity(foreignKeys = @ForeignKey(entity = User.class,
        parentColumns = "id",
        childColumns = "user_id"))
class Book {
    @PrimaryKey
    public int bookId;


    public String title;


    @ColumnInfo(name = "user_id")
    public int userId;
}
```

Ein kombiniertes Objekt abfragen

```
@Dao
interface MyDao {
    @Query("SELECT user.name AS userName, pet.name AS petName "
            + "FROM user, pet "
            + "WHERE user.id = pet.user_id")
    fun loadUserAndPetNames(): <List<UserPet>>


    // You can also define this class in a separate file, as long as  //
       // you  add the "public" access modifier.
    class UserPet {
        var userName: String? = null
        var petName: String? = null
    }
}
```

Room Exercise 1

- Git checkout
- Suche alle “TODO: room_exercise1”
- Füge dort die nötigen Room annotations hinzu

Tag: 03_room_exercise1_TAG
Branch: step_03_room_exercise1

RX.Java and LiveData

- DAO’s können auch RxJava Objekte oder LiveData zurückgeben
- Automatisch Änderungen in der Datenbank erkennen

DB- Änderungen

Automatisches UI Update

RxJava und Room

- Maybe
  1. Kein Eintrag in der DB → Keine Reihe wird zurückgegeben
  2. Ein Eintrag ist in der DB → Das Resultat kommt im onSuccess zurück
  3. Bei einer Aktualisierung → Es geschieht nichts

```
@Query("SELECT * FROM Users WHERE id = :userId")
Maybe<User> getUserById(String userId);
```

RxJava und Room

- Single
  1. Kein Eintrag in der DB
     → Keine Reihe wird zurückgegeben und onError(EmptyResultSetException.class) wird aufgerufen
  2. Ein Eintrag ist in der DB
     → Das Resultat kommt zurück und onSuccess wird aufgerufen
  3. Bei einer Aktualisierung nachdem Single.onComplete aufgerufen wurde
     → Es geschieht nichts

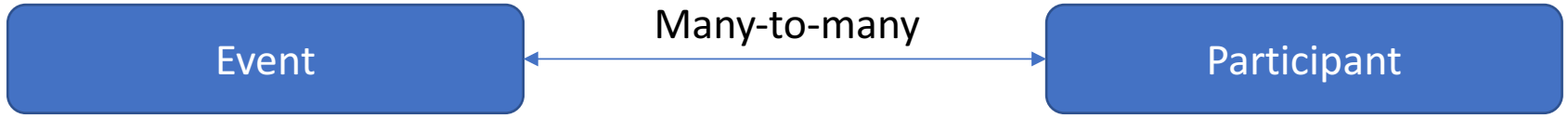
```
@Query("SELECT * FROM Users WHERE id = :userId")
Single<User> getUserById(String userId);
```

RxJava und Room

- Flowable
  1. Kein Eintrag in der DB
     → Keine Reihe wird zurückgegeben, weder onNext noch onError wird aufgerufen
  2. Ein Eintrag ist in der DB
     → Das Flowable ruft onNext auf
  3. Bei einer Aktualisierung
     → Es wird jedes Mal onNext aufgerufen, so kann das UI aktualisiert werden

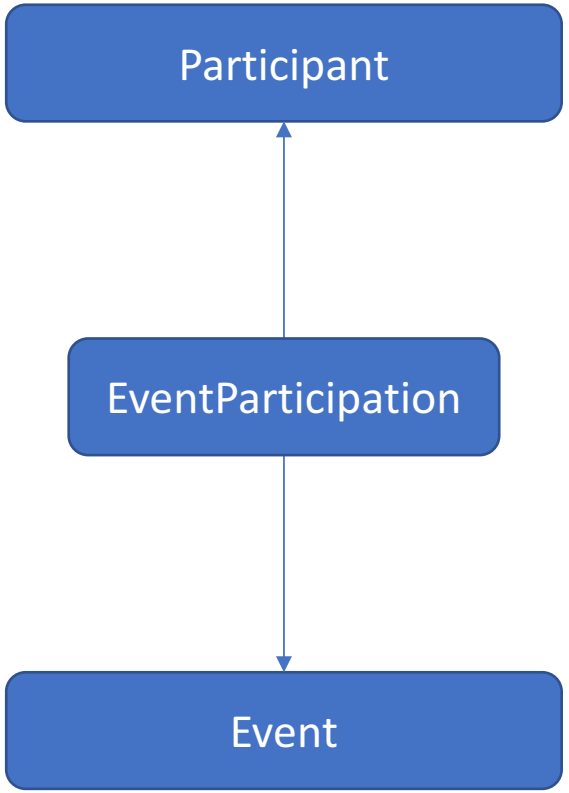
```
@Query("SELECT * FROM Users WHERE id = :userId")
Flowable<User> getUserById(String userId);
```

Many-to-Many Relationen



- Room wird keine Magic-Tabellen erstellen.
- Es muss für jede Tabelle eine Entity explizit angegeben werden
- -> **EventParticipation-Entity** muss erstellt werden

Many-to-many Room




```
@Entity(foreignKeys = {
        @ForeignKey(entity = Participant.class,
                parentColumns = "participantId",
                childColumns = "participant_id"),
        @ForeignKey(entity = Event.class,
                parentColumns = "eventId",
                childColumns = "event_id")})
public class EventParticipation {
    @PrimaryKey(autoGenerate = true)
    public long id;
    @ColumnInfo(name="event_id")
    public long eventId;
    @ColumnInfo(name="participant_id")
    public long participantId;
    public EventParticipation(){}
}
```

Schlussfolgerung

- Einfach zu benutzen
- Verringert BoilerplateCode
- Benutzt eine speziellen SQL-Cursor -> viel viel schneller!!
- Many-to-many Relationen müssen explizit angegeben werden
- DB-Änderungen können mit RX.java oder LiveData beobachtet werden

Quellen

- https://medium.com/google-developers/room-rxjava-acb0cd4f3757
- https://developer.android.com/topic/libraries/architecture/room.html
- https://medium.com/google-developers/7-steps-to-room-27a5fe5f99b2

FAQ

- Wie kann man eine existierende Datenbank zu Room migrieren?
  - Mit Migrations, damit die bestehenden Daten bleiben. Die Migration-Datei ist aber leer, weil die Tabellen schon existieren. Mehr Details unter: https://medium.com/google-developers/7-steps-to-room-27a5fe5f99b2