Uniwersytet Wrocławski

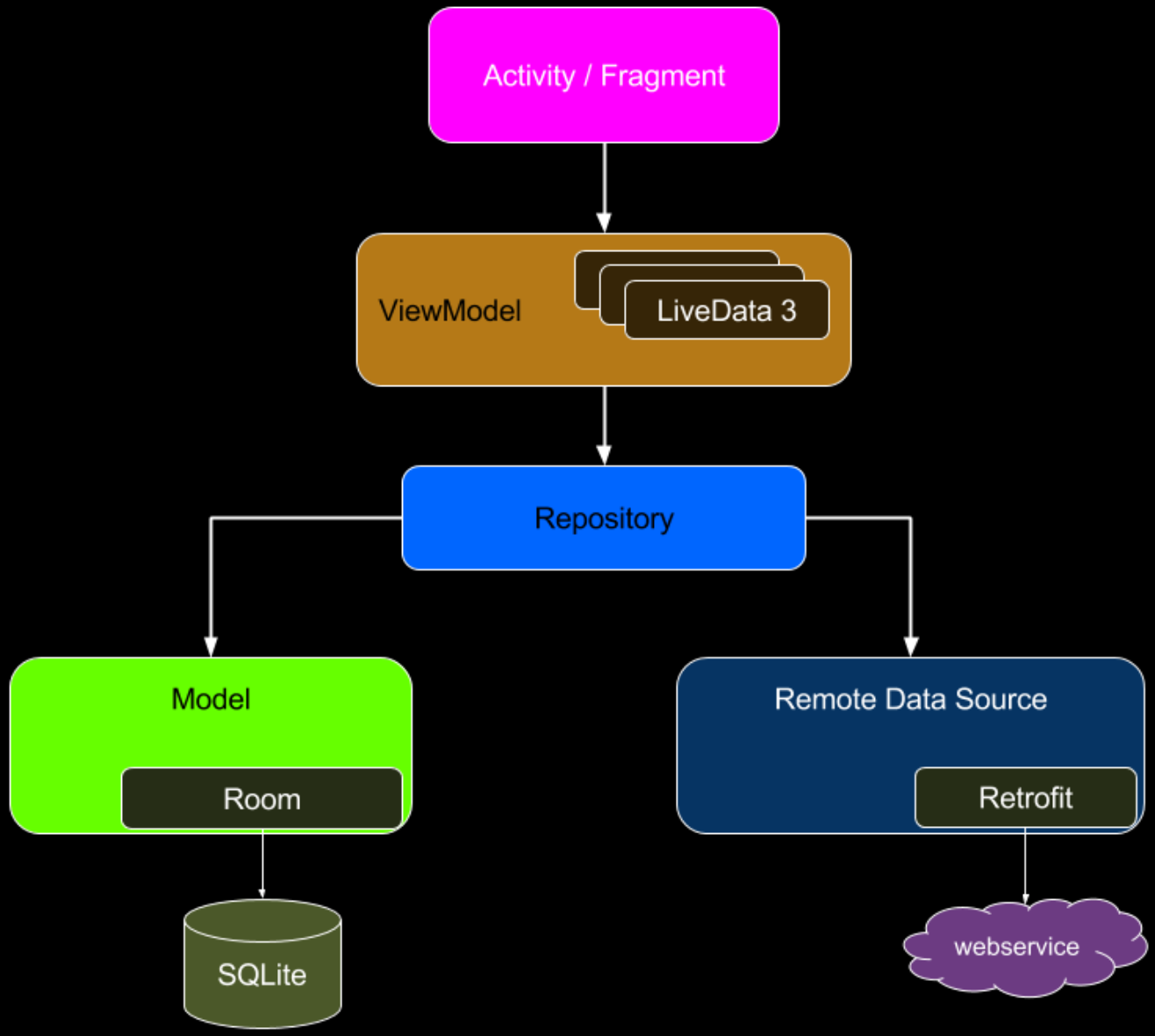
Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

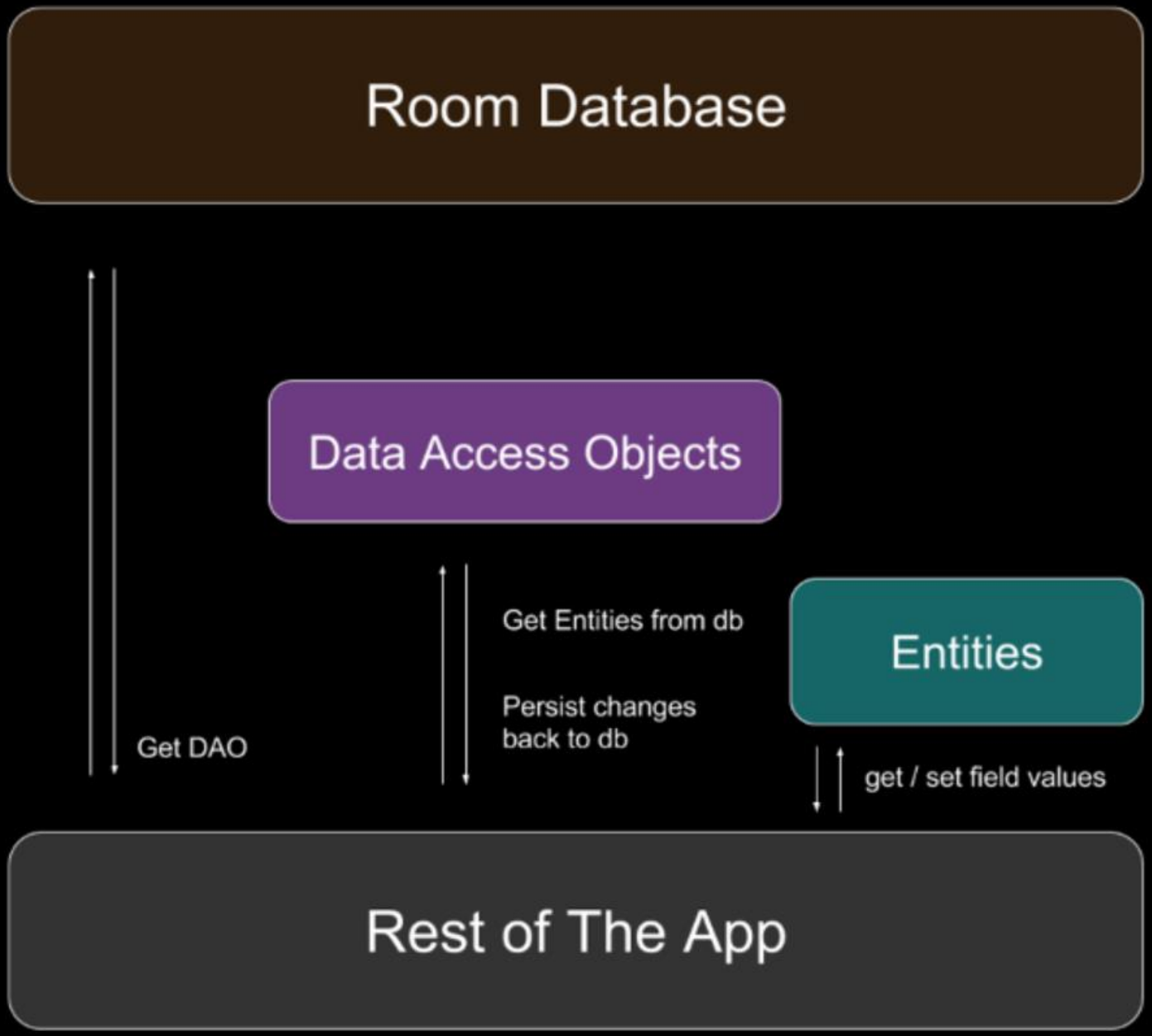
WYKŁAD 9

- ROOM

MVVM

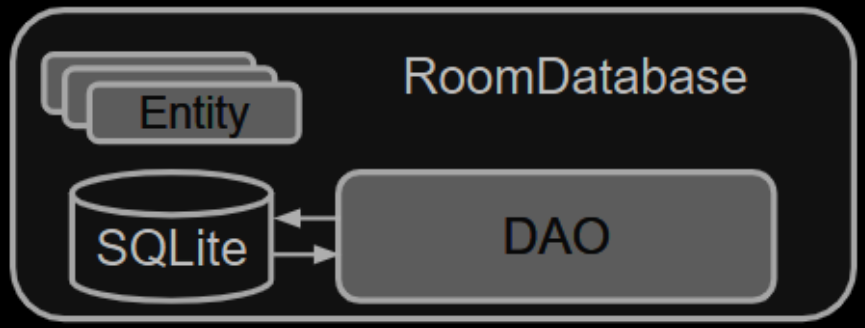

Activity / Fragment
ViewModel
LiveData 3
Repository
Model
Room
Remote Data Source
Retrofit
SQLite
webservice

ROOM


Room Database
Data Access Objects
Get DAO
Get Entities from db
Persist changes back to db
Entities
get / set field values
Rest of The App

ROOM

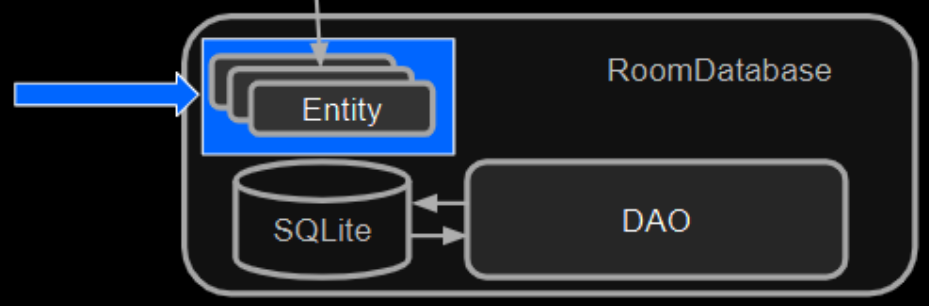
- **Entity**: Defines schema of database table.
- **DAO**: Database Access Object
  Defines read/write operations for database.
- **Database**:
  A database holder.
  Used to create or connect to database



Entity

- Entity instance = row in a database table
- Define entities as POJO classes
- 1 instance = 1 row
- Member variable = column name

```
public class Person {
    private int uid;
    private String firstName;
    private String lastName;
}
```



Entity

```
public class Person {
    private int uid;
    private String firstName;
    private String lastName;
}
```

uid	firstName	lastName
12345	Aleks	Becker
12346	Jhansi	Kumar

Entity

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

Entity

```
@PrimaryKey (autoGenerate=true)
```

- **`Entity`** class must have a field annotated as primary key
- You can auto-generate unique key for each entity

Entity

`@NonNull`

- Denotes that a parameter, field, or method return value can never be `null`
- Use for mandatory fields
- Primary key must use `@NonNull`

DAO

- Metody zdefiniowane w DAO zapewniają dostęp do bazy danych
- DAO musi być interfejsem lub klasą abstrakcyjną
- Implementacje metod są generowane przez Room

DAO

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

```
@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
    fun insertAll(vararg users: User)

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

ROOM

- Create public abstract class extending `RoomDatabase`
- Annotate as `@Database`
- Declare entities for database schema and set version number

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase extends RoomDatabase
```

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

ROOM

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase
                      extends RoomDatabase {

   public abstract WordDao wordDao();

   private static WordRoomDatabase INSTANCE;

   // ... create instance here
}
```

Entity defines DB schema

DAO for database

Create database as singleton instance

ROOM

```
static WordRoomDatabase getDatabase(final Context context) {
  if (INSTANCE == null) {
    synchronized (WordRoomDatabase.class) {
      if (INSTANCE == null) {
        INSTANCE = Room.databaseBuilder(
            context.getApplicationContext(),
            WordRoomDatabase.class, "word_database")
            .addCallback(sOnOpenCallback)
            .fallbackToDestructiveMigration()
            .build();
      }}}
  return INSTANCE;
```

Check if database exists before creating it

ROOM

```
@Database(
    entities = [
        Faculty::class,
        Dean::class,
    ],
    version = 1,
    exportSchema = false
)
abstract class FacultyRoomDatabase : RoomDatabase() {
    abstract val facultyDao: FacultyDao

    companion object{
        @Volatile
        private var INSTANCE: FacultyRoomDatabase? = null

        fun getInstance(context: Context): FacultyRoomDatabase{
            synchronized(this){
                return  INSTANCE ?: Room.databaseBuilder(
                    context.applicationContext,
                    FacultyRoomDatabase::class.java,
                    "kotlin_faculty_db"
                ).build().also {
                    INSTANCE = it
                }
            }
        }
    }
}
```

ROOM

```
private static RoomDatabase.Callback sOnOpenCallback =
  new RoomDatabase.Callback(){
   @Override
   public void onOpen (@NonNull SupportSQLiteDatabase db){
       super.onOpen(db);
       initializeData();
   }};
```

Relacje

```
@Dao
interface UserBookDao {
    @Query(
        "SELECT user.name AS userName, book.name AS bookName " +
        "FROM user, book " +
        "WHERE user.id = book.user_id"
    )
    fun loadUserAndBookNames(): LiveData<List<UserBook>>
}

data class UserBook(val userName: String?, val bookName: String?)
```

```
@Query(
    "SELECT * FROM user" +
    "JOIN book ON user.id = book.user_id"
)
fun loadUserAndBookNames(): Map<User, List<Book>>
```

Relacje

```
data class Address(
    val street: String?,
    val state: String?,
    val city: String?,
    @ColumnInfo(name = "post_code") val postCode: Int
)

@Entity
data class User(
    @PrimaryKey val id: Int,
    val firstName: String?,
    @Embedded val address: Address?
)
```

Relacje 1-1

```
@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Library(
    @PrimaryKey val libraryId: Long,
    val userOwnerId: Long
)
```

Relacje 1-1

```
data class UserAndLibrary(
    @Embedded val user: User,
    @Relation(
        parentColumn = "userId",
        entityColumn = "userOwnerId"
    )
    val library: Library
)
```

```
@Transaction
@Query("SELECT * FROM User")
fun getUsersAndLibraries(): List<UserAndLibrary>
```

Relacje 1-N

```
@Entity
data class User(
    @PrimaryKey val userId: Long,
    val name: String,
    val age: Int
)

@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val userCreatorId: Long,
    val playlistName: String
)
```

Relacje 1-N

```
data class UserWithPlaylists(
    @Embedded val user: User,
    @Relation(
        parentColumn = "userId",
        entityColumn = "userCreatorId"
    )
    val playlists: List<Playlist>
)
```

```
@Transaction
@Query("SELECT * FROM User")
fun getUsersWithPlaylists(): List<UserWithPlaylists>
```

Relacje M-N

```
@Entity
data class Playlist(
    @PrimaryKey val playlistId: Long,
    val playlistName: String
)

@Entity
data class Song(
    @PrimaryKey val songId: Long,
    val songName: String,
    val artist: String
)

@Entity(primaryKeys = ["playlistId", "songId"])
data class PlaylistSongCrossRef(
    val playlistId: Long,
    val songId: Long
)
```

Relacje M-N

```
data class PlaylistWithSongs(
    @Embedded val playlist: Playlist,
    @Relation(
        parentColumn = "playlistId",
        entityColumn = "songId",
        associateBy = Junction(PlaylistSongCrossRef::class)
    )
    val songs: List<Song>
)

data class SongWithPlaylists(
    @Embedded val song: Song,
    @Relation(
        parentColumn = "songId",
        entityColumn = "playlistId",
        associateBy = Junction(PlaylistSongCrossRef::class)
    )
    val playlists: List<Playlist>
)
```

Relacje M-N

```
@Transaction
@Query("SELECT * FROM Playlist")
fun getPlaylistsWithSongs(): List<PlaylistWithSongs>

@Transaction
@Query("SELECT * FROM Song")
fun getSongsWithPlaylists(): List<SongWithPlaylists>
```

Asynchroniczność

Query type	Kotlin language features	RxJava	Guava	Jetpack Lifecycle
One-shot write	Coroutines (suspend)	`Single<T>`, `Maybe<T>`, `Completable`	`ListenableFuture<T>`	N/A
One-shot read	Coroutines (suspend)	`Single<T>`, `Maybe<T>`	`ListenableFuture<T>`	N/A
Observable read	`Flow<T>`	`Flowable<T>`, `Publisher<T>`, `Observable<T>`	N/A	`LiveData<T>`

Asynchroniczność

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUsers(vararg users: User)

    @Update
    suspend fun updateUsers(vararg users: User)

    @Delete
    suspend fun deleteUsers(vararg users: User)

    @Query("SELECT * FROM user WHERE id = :id")
    suspend fun loadUserById(id: Int): User

    @Query("SELECT * from user WHERE region IN (:regions)")
    suspend fun loadUsersByRegion(regions: List<String>): List<User>
}
```

Asynchroniczność

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUsers(vararg users: User)

    @Update
    suspend fun updateUsers(vararg users: User)

    @Delete
    suspend fun deleteUsers(vararg users: User)

    @Query("SELECT * FROM user WHERE id = :id")
    suspend fun loadUserById(id: Int): User

    @Query("SELECT * from user WHERE region IN (:regions)")
    suspend fun loadUsersByRegion(regions: List<String>): List<User>
}
```

Asynchroniczność

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user WHERE id = :id")
    fun loadUserById(id: Int): Flow<User>

    @Query("SELECT * from user WHERE region IN (:regions)")
    fun loadUsersByRegion(regions: List<String>): Flow<List<User>>
}
```

```
@Query("SELECT * FROM word_table ORDER BY word ASC")
    fun getAlphabetizedWords(): LiveData<List<Word>>
```

Asynchroniczność

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(word: Word)
```

```
fun insert(word: Word) {
        viewModelScope.launch{
            db.wordDao().insert(word)
        }
    }
```