Uniwersytet Wrocławski

Rafał Lewandków

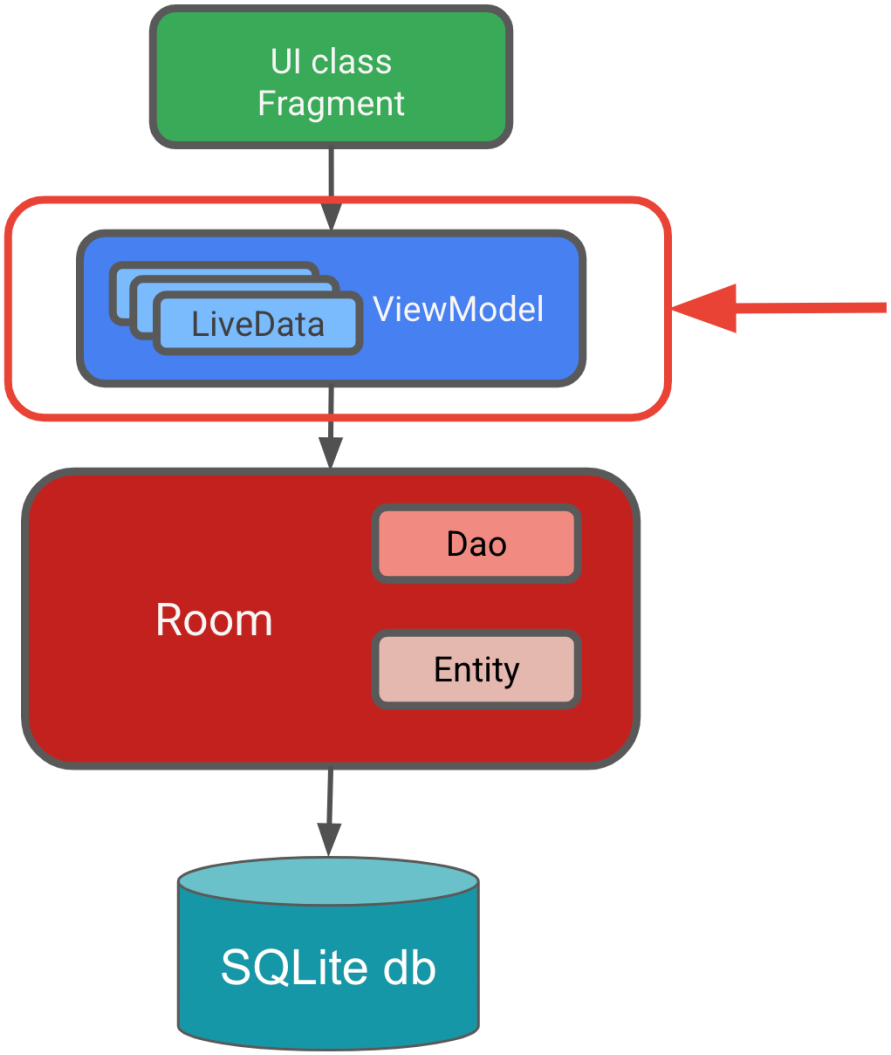
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

WYKŁAD 11

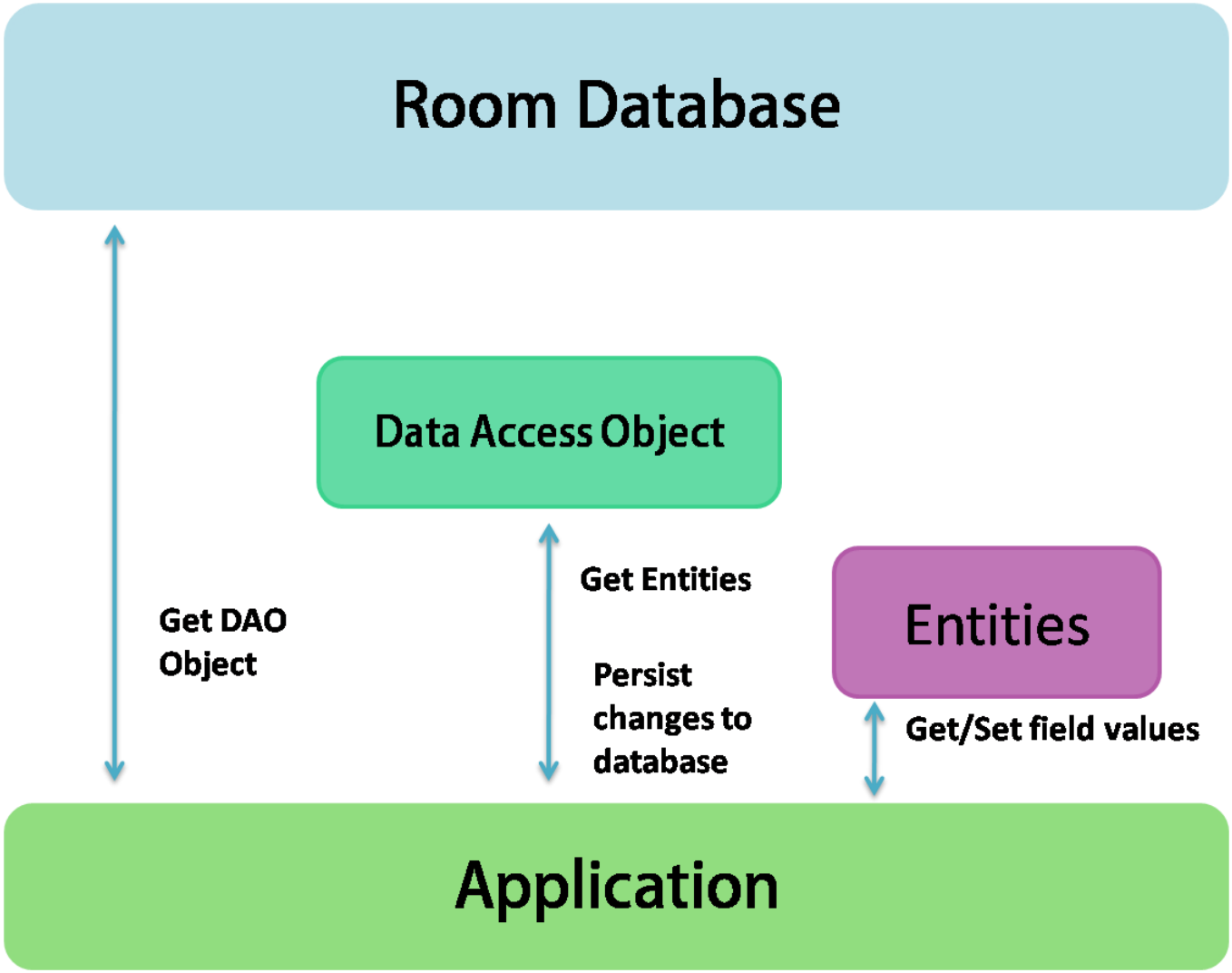
Bazy danych ROOM

- Dao
- Entity
- SQL

ROOM

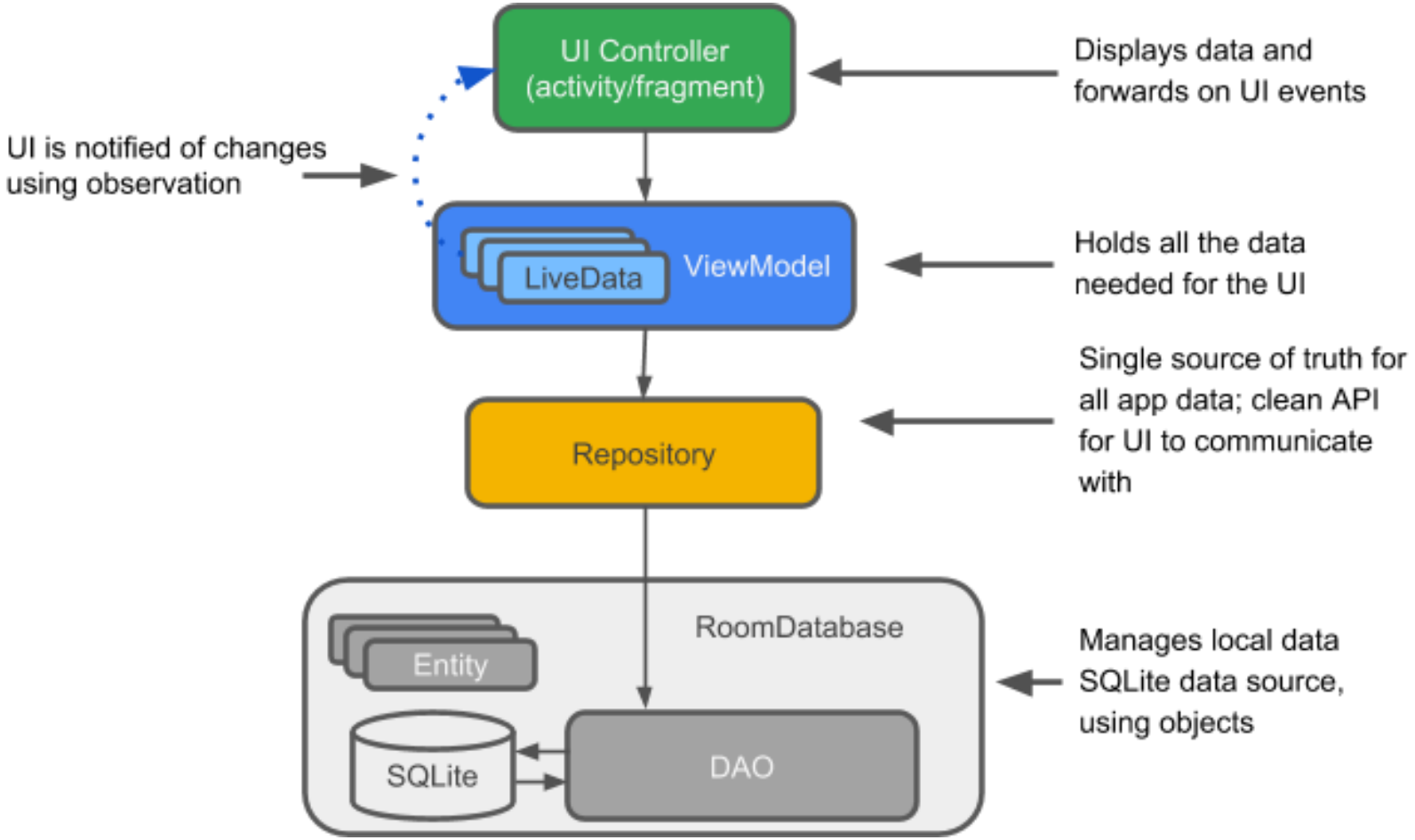

UI class
Fragment
LiveData
ViewModel
Room
Dao
Entity
SQLite db

ROOM



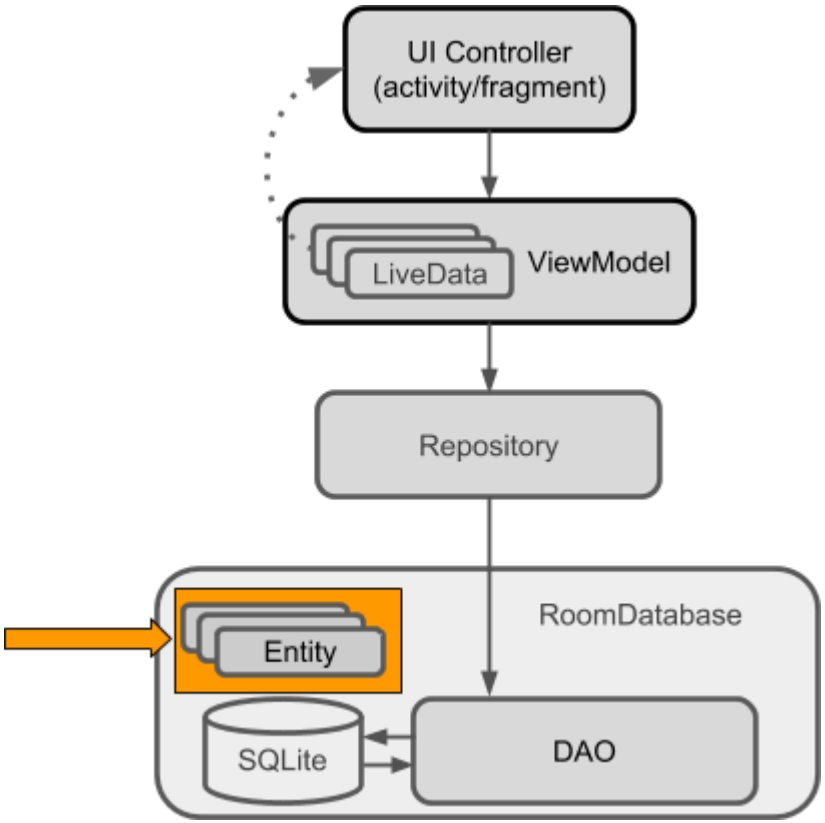
https://www.google.com/url?sa=i&url=https%3A%2F%2Fmedium.com%2F%40kuldeepmakwana3977%2Froom-database-using-android-kotlin-8cd3cd854b15

ROOM


UI Controller
(activity/fragment)
Displays data and
forwards on UI events
UI is notified of changes
using observation
LiveData
ViewModel
Holds all the data
needed for the UI
Repository
Single source of truth for
all app data; clean API
for UI to communicate
with
RoomDatabase
Entity
SQLite
DAO
Manages local data
SQLite data source,
using objects

Uniwersytet Wrocławski

Entity



https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-1-c-room-livedata-viewmodel/10-1-c-room-livedata-viewmodel.html

Uniwersytet Wrocławski

Entity

ID	First name	Last name	...
12345	Aleks	Becker	...
12346	Jhansi	Kumar	...

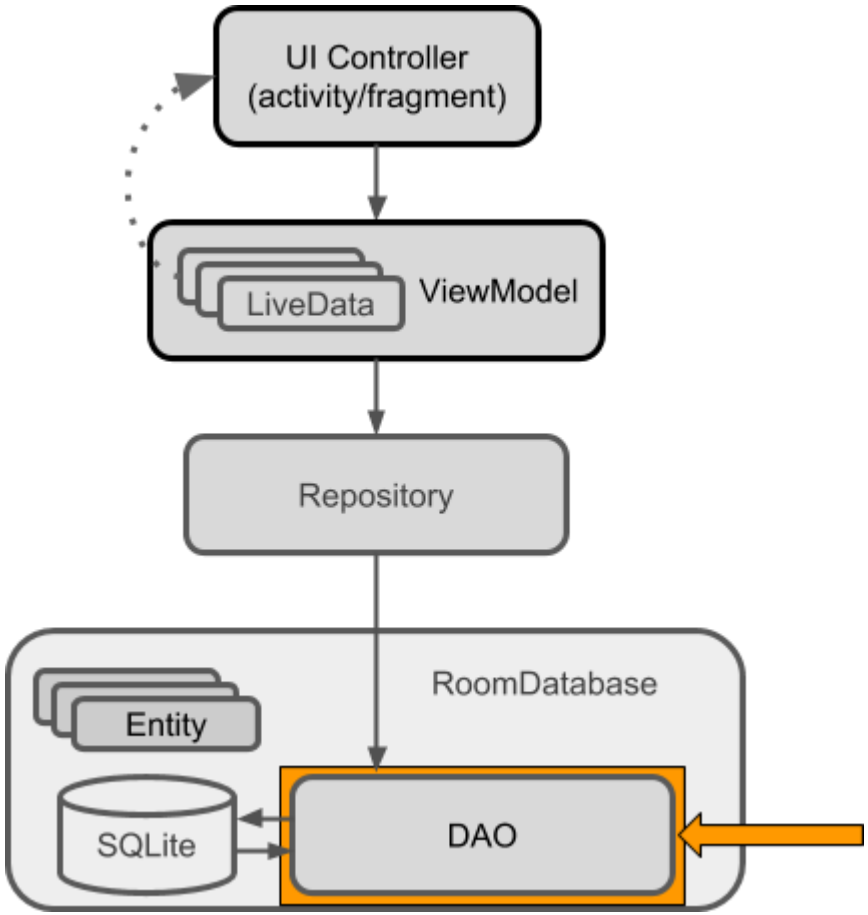
```
@Entity
public class Person {
    @PrimaryKey (autoGenerate=true)
    private int uid;

    @ColumnInfo(name = "first_name")
    private String firstName;

    @ColumnInfo(name = "last_name")
    private String lastName;
```

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-1-c-room-livedata-viewmodel/10-1-c-room-livedata-viewmodel.html

Dao



https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-1-c-room-livedata-viewmodel/10-1-c-room-livedata-viewmodel.html

Dao

```
@Dao
public interface WordDao {

   // The conflict strategy defines what happens,
   // if there is an existing entry.
   // The default action is ABORT.
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   void insert(Word word);

   // Update multiple entries with one call.
   @Update
   public void updateWords(Word... words);

   // Simple query that does not take parameters and returns nothing.
   @Query("DELETE FROM word_table")
   void deleteAll();

   // Simple query without parameters that returns values.
   @Query("SELECT * from word_table ORDER BY word ASC")
   List<Word> getAllWords();

   // Query with parameter that returns a specific word or words.
   @Query("SELECT * FROM word_table WHERE word LIKE :word ")
   public List<Word> findWord(String word);
}
```

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-1-c-room-livedata-viewmodel/10-1-c-room-livedata-viewmodel.html

Uniwersytet Wrocławski

SQLite

Główne cechy `SQLite` to:

- Działa jako biblioteka dostępna w postaci pliku w aplikacji, co oznacza, że nie wymaga oddzielnego procesu serwera bazy danych. Aplikacja może bezpośrednio komunikować się z bazą danych.
- Jest samowystarczalna, co oznacza, że cała baza danych jest przechowywana w jednym pliku. Nie ma potrzeby konfigurowania lub zarządzania wieloma plikami lub zasobami.
- Transakcje - Obsługuje transakcje, co umożliwia grupowanie operacji bazodanowych jako pojedyncze, atomowe działanie. Transakcje są ważne, gdy chodzi o utrzymanie integralności danych. Każda operacja na bazie jest wykonywana w całości lub w ogóle - jest to ważne przy modyfikowaniu wielu tabel, oraz asynchronicznym przetwarzaniu.
- Wsparcie dla standardowego `SQL` - obsługuje większość standardowego języka `SQL`, co ułatwia programowanie i wykonywanie zapytań.
- Jest zaprojektowany w taki sposób, aby działał wydajnie nawet na urządzeniach o ograniczonych zasobach sprzętowych.

Główne składniki biblioteki `Room` to:

- `Entity` - Reprezentuje tabelę w bazie danych `SQLite`. Każda klasa oznaczona adnotacją `@Entity` może być mapowana do jednej tabeli w bazie danych, a pola klasy odpowiadają kolumnom tej tabeli.
- `DAO` (*Data Access Object*) - Definiuje metody, które umożliwiają dostęp do danych w bazie danych. Możemy zdefiniować interfejs `DAO` za pomocą adnotacji `@Dao`, a Room automatycznie dostarczy implementację tych metod.
- `Database` - Klasa bazowa, która reprezentuje bazę danych. To miejsce, gdzie definiujemy wszystkie *encje*, które mają zostać użyte w aplikacji, oraz wersję bazy danych. Room tworzy implementację bazy danych opartej na tej klasie.

Uniwersytet Wrocławski

ROOM + MVVM

```
@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int,
    val firstName: String,
    val lastName: String
)
```

Uniwersytet Wrocławski

ROOM + MVVM

```kotlin
@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int,
    val firstName: String,
    val lastName: String
)
```

```kotlin
@Dao
interface UserDao {
    @Query("SELECT * FROM user_table ORDER BY lastName ASC, firstName ASC")
    fun getUsers(): Flow<List<User>>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(user: User)

    @Query("DELETE FROM user_table")
    suspend fun deleteAll()
}
```

Uniwersytet Wrocławski

ROOM + MVVM

```
@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int,
    val firstName: String,
    val lastName: String
)
```

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user_table ORDER BY lastName ASC, firstName ASC")
    fun getUsers(): Flow<List<User>>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(user: User)

    @Query("DELETE FROM user_table")
    suspend fun deleteAll()
}
```

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

    companion object {
        @Volatile
        private var Instance: UserDatabase? = null

        fun getDatabase(context: Context): UserDatabase {
            return Instance ?: synchronized(this) {
                Room.databaseBuilder(context, UserDatabase::class.java, "user_database")
                    .build()
                    .also { Instance = it }
            }
        }
    }
}
```

ROOM + MVVM

```
1   package com.example.roombasicskotlin.repository
2
3   import com.example.roombasicskotlin.data.User
4   import com.example.roombasicskotlin.data.UserDao
5
6   class UserRepository(private val userDao: UserDao) {
7       fun getUsers() = userDao.getUsers()
8       suspend fun clear() = userDao.deleteAll()
9       suspend fun add(user: User) = userDao.insert(user)
10  }
```

Uniwersytet Wrocławski

ROOM + MVVM

```
package com.example.roombasicskotlin.repository

import com.example.roombasicskotlin.data.User
import com.example.roombasicskotlin.data.UserDao

class UserRepository(private val userDao: UserDao) {
    fun getUsers() = userDao.getUsers()
    suspend fun clear() = userDao.deleteAll()
    suspend fun add(user: User) = userDao.insert(use
}
```

```
class UserViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: UserRepository
    private val _usersState = MutableStateFlow<List<User>>(emptyList())
    val usersState: StateFlow<List<User>>
        get() = _usersState

    init {
        val db = UserDatabase.getDatabase(application)
        val dao = db.userDao()
        repository = UserRepository(dao)

        fetchUsers()
    }

    private fun fetchUsers() {
        viewModelScope.launch {
            repository.getUsers().collect { users ->
                _usersState.value = users
            }
        }
    }

    fun clearUsers() {
        viewModelScope.launch {
            repository.clear()
        }
    }

    fun addUser(user: User) {
        viewModelScope.launch {
            repository.add(user)
        }
    }
}
```

ROOM + MVVM

```
class UserViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: UserRepository
    private val _usersState = MutableStateFlow<List<User>>(emptyList())
    val usersState: StateFlow<List<User>>
        get() = _usersState

    init {
        val db = UserDatabase.getDatabase(application)
        val dao = db.userDao()
        repository = UserRepository(dao)

        fetchUsers()
    }

    private fun fetchUsers() {
        viewModelScope.launch {
            repository.getUsers().collect { users ->
                _usersState.value = users
            }
        }
    }

    fun clearUsers() {
        viewModelScope.launch {
            repository.clear()
        }
    }

    fun addUser(user: User) {
        viewModelScope.launch {
            repository.add(user)
        }
    }
}
```