# ANDROID

## SQLITE SESSION

# Agenda

- Local Storage
- Room
- Step-by-Step Implementation
- Repository
- View model
- View Model Factory

# LOCAL STORAGE

# Local Storage

## Why Do We Need Local Storage?

In any app, we often need to store data — but not all data needs to come from or go to a server.

Sometimes, we just want to store data that is:

- For the user only
- Doesn't need to be shared across devices
- Should still be available even without internet

# Local Storage

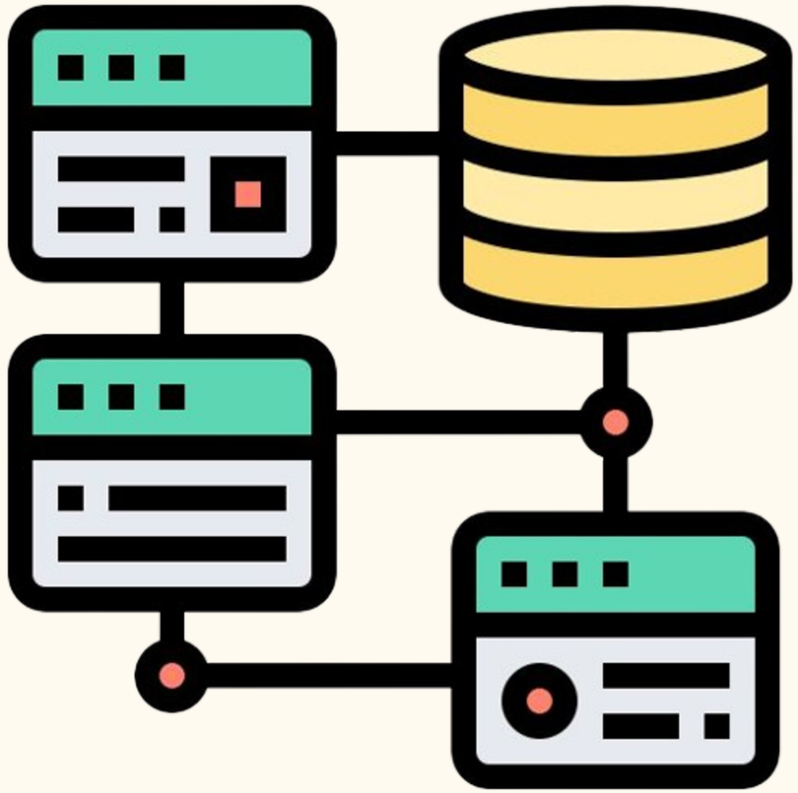
## Example

In a meal or recipe app, if the user adds a meal to their Favorites, this is:

- Not something we need to save on a server
- It's personal and should be available offline

# Local Database vs Remote Database

| Feature | Local Database (Room) | Remote Database (e.g., Firebase, API) |
|---|---|---|
| Stored on | The user's device | A server on the internet |
| Requires internet? | No | Yes |
| Best for | Favorites , offline content | User accounts, shared posts |
| Example | Favorite meals | Login info, chat messages, global recipes |



# ROOM

# Room

## What is Room?

Room is a library provided by Google that makes it easier to use a local SQLite database in Android apps.A smarter and cleaner way to talk to a local database using simple Kotlin/Java code — without writing raw SQL.

# Room Architecture Overview

Room is built on three core components that work together to store and retrieve data locally:

1. Entity :  Defines the structure of your table
2. DAO  : Provides methods to interact with data
3. Database : Ties everything together

# STEP-BY-STEP IMPLEMENTATION

# Step-by-Step Room Implementation Using MVVM

**We’ll build a simple User App with:**

- Entity : User: name, email
- Room Database
- DAO
- Repository
- ViewModel
- Activity/Fragment to display or insert data

# Step-by-Step Room Implementation Using MVVM

**Add Room dependencies**

```
// Room components
implementation ("androidx.room:room-runtime:2.6.1")
kapt ("androidx.room:room-compiler:2.6.1")

// Kotlin Extensions and Coroutines support for Room
implementation ("androidx.room:room-ktx:2.6.1")
```

# Entity

## What is an Entity in Room?

An Entity in Room represents a table in the SQLite database.

- Each Entity class = One Table
- Each property = One Column
- It’s just a Kotlin data class with the @Entity annotation

# Entity

## Example: User Entity

```
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "users") // This will create a table named "users"
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int = 0, // Primary key column
    val name: String,  // Becomes "name" column
    val email: String  // Becomes "email" column
)
```

# Entity

## Example: User Entity

| Field | Meaning |
| --- | --- |
| @Entity | Marks this class as a Room table |
| tableName | Optional: specify table name (default = class name) |
| @PrimaryKey | Required: tells Room this is the unique ID |
| autoGenerate | Automatically increments the ID |
| Other fields | Regular Kotlin properties = DB columns |

# DAO

## What is DAO (Data Access Object)?

DAO stands for Data Access Object. It is an **interface** (or abstract class) that tells Room how to interact with your database.

in other way,In Room, each table (Entity) should usually have its own DAO.

DAO = a list of functions you use to insert, read, update, or delete data.

# DAO

## What Must You Write in a DAO?

You write abstract functions and mark them with annotations like:

- @Insert ➜ to add data
- @Query ➜ to get data
- @Update ➜ to update data
- @Delete ➜ to remove data

Room automatically generates the implementation.

# DAO

## Example: User DAO

```
@Dao
interface UserDao {

    @Insert
    suspend fun insertUser(user: User)

    @Query("SELECT * FROM users")
    suspend fun getAllUsers(): List<User>
```

# DAO

## Example: User DAO

```kotlin
@Query("SELECT * FROM users WHERE id = :id")
suspend fun getUserById(id: Int): User?

@Delete
suspend fun deleteUser(user: User)

@Update
suspend fun updateUser(user: User)
```

**very important note**

**What does :id mean?**
**It means:**
**"Use the value from the Kotlin parameter named id here in the SQL query."**

# DAO

## Example: User DAO

| Annotation | Purpose |
| --- | --- |
| @Dao | Add it above the interface |
| @Insert | Inserts one or more rows |
| @Query | Executes SQL query |
| @Delete | Deletes a row |
| @Update | Updates an existing row |

# Room Database

## What is a Room Database?

It’s the main class that connects:

- Entities (your tables)
- DAOs (your queries)

It tells Room which tables and DAOs exist, and it gives you access to them.

# Database

## Example: Database

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

    abstract fun userDao(): UserDao

    companion object {
        @Volatile
        private var instance: AppDatabase? = null
```

# Database

## Example: Database

```
fun getDatabase(context: Context): AppDatabase =
    instance ?: synchronized(this) {
        instance ?: Room.databaseBuilder(
            context.applicationContext,
            AppDatabase::class.java,
            "user_database"
        ).fallbackToDestructiveMigration()
         .build().also { instance = it }
```

# Database

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

This annotation tells Room:
entities = [User::class]: This database includes the User table.

# Database

## abstract class AppDatabase : RoomDatabase()

- This class extends RoomDatabase and represents the actual SQLite database.
- Being abstract means you don’t create objects of it directly — Room generates the necessary implementation.

## abstract fun userDao(): UserDao

This function tells Room: “Please provide an implementation of the UserDao interface.”

# Database

## companion object

- This is a Kotlin way to create static-like functions and variables.
- In Room, we often use it to implement the singleton pattern — making sure we only create one instance of the database.

## @Volatile private var instance: AppDatabase? = null

- @Volatile makes sure all threads see the latest value of instance.
- instance stores the single shared database object.
- It's nullable at first because we haven’t created it yet.

# Database

## fun getDatabase(context: Context): AppDatabase

- This function returns the singleton database instance.
- If the database has already been created (instance != null), it returns it.
- If not, it enters a synchronized block to make sure only one thread can build it.

## synchronized(this) { ... }

- Prevents multiple threads from creating the database at the same time.
- Ensures thread safety — only one instance will ever be created, even in multi-threaded apps.

# Database

## Room.databaseBuilder(...)

- This line builds the Room database.
- context.applicationContext: Ensures no memory leaks from passing activities.
- AppDatabase::class.java: Tells Room which class represents the DB.
- "user_database": The database file name stored on the device.

## .fallbackToDestructiveMigration()

- Warning: This deletes all old data and rebuilds the database if a migration (e.g. version change) is needed.
- Useful during development but not recommended in production unless you handle backup/migrations.

# Database

## .build().also { instance = it }

- build() finalizes the database setup. Builder design pattern
- also { instance = it } saves the result into our instance variable. read about also function

# Database

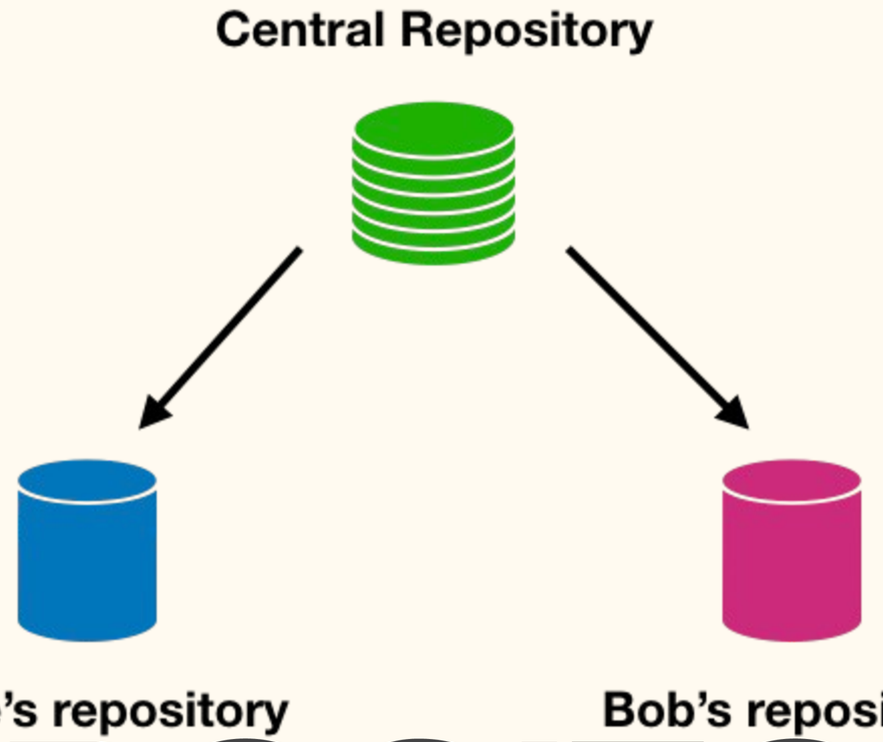
## How to use your database

```
val db = AppDatabase.getDatabase(context)
val userDao = db.userDao()

lifecycleScope.launch {

    userDao.insert(User(name = "Abdelrahman", email = "abdo@example.com"))

    val allUsers = userDao.getAllUsers()
}
```




# REPOSITORY

# Repository

## What is a Repository in MVVM?

In Android’s MVVM architecture, a Repository is a class that abstracts the data layer. It acts as a single source of truth for your ViewModel — whether your data comes from:

- A Room database
- A Remote API
- Or even SharedPreferences

# Repository

## Why Do We Use It?

### Without a repository:

Your ViewModel would directly call DAOs, APIs, etc. — tightly coupling them to data sources.

### With a repository:

- You get clean separation of concerns.
- Your ViewModel is only responsible for UI logic.
- Easier to test and scale your app.

# Repository

```
class UserRepository(private val userDao: UserDao) {

    val allUsers: LiveData<List<User>> = userDao.getAllUsers()

    suspend fun insert(user: User) {
        userDao.insertUser(user)
    }
}
```

AGAIN

# VIEW MODEL

# Quick Recap of ViewModel

1. “Last time, we learned that ViewModel is responsible for holding UI-related data and surviving configuration changes.”
2. "But ViewModel shouldn't directly talk to the database or API — that breaks separation of concerns."
3. Repository is the middle layer between ViewModel and data sources like Room or network.”

# NEW ViewModel

```
class UserViewModel(private val repository: UserRepository) : ViewModel() {

    val allUsers: LiveData<List<User>> = repository.getAllUsers()

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

How to send paramter to our view model ?? we will need ViewModelFactory

# VIEW MODEL FACTORY

# View Model Factory

In Android, we cannot directly pass parameters to a ViewModel constructor when using ViewModelProvider because of how the system instantiates ViewModels behind the scenes.

## Why do we use ViewModelFactory?

A ViewModelFactory is a class you create to tell the system how to construct your ViewModel with the needed parameters. it is based on the Factory Design Pattern

# View Model Factory

## what is Factory Design Pattern

The Factory Design Pattern is used to create objects without showing the creation logic to the user. Instead of creating the object directly, we use a factory class with a create() method that returns the right object based on some input. This makes the code easier to manage, more flexible, and allows us to pass any needed data or dependencies when creating the object.

# View Model Factory

```
class FavoriteViewModelFactory(
    private val repository: Repository
) : ViewModelProvider.Factory {


    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return if (modelClass.isAssignableFrom(FavoriteViewModel::class.java)) {
            FavoriteViewModel(repository) as T
        } else {
            throw IllegalArgumentException("Unknown ViewModel class")
        }
```

# View Model Factory

## Simple Explanation

It is a class that creates ViewModel instances.
I extend ViewModelProvider.Factory,
then I override the create() method,
and inside it, I call the ViewModel constructor manually, especially if it needs parameters like a Repository.

# View Model Factory

**MyViewModelFactory(private val repo: Repository) : ViewModelProvider.Factory {**

- You're creating a class called MyViewModelFactory.
- It takes a Repository object as a constructor parameter (repo).
- It inherits from ViewModelProvider.Factory, which is required for creating ViewModels manually.

# View Model Factory

```
override fun <T : ViewModel>
create(modelClass: Class<T>): T {
```

- You are overriding the create function from the ViewModelProvider.Factory interface.
- It returns an object of type T.
- T is must be inherit from ViewModel

# View Model Factory

```
return
 if (modelClass.isAssignableFrom(
MyViewModel::class.java))
```

You're checking if the requested modelClass is the same as, or a superclass of, MyViewModel.

# View Model Factory

**MyViewModel(repo) as T**

- You create a new instance of MyViewModel, passing in the repo dependency.
- You cast it to type T, which is safe because you already checked the class.

# View Model Factory

## How to create viewmodel with paramter

```
val viewModel = ViewModelProvider(this,
MyViewModelFactory(repo)).get(MyViewModel::class.java)
```

you will pass the factory object with repository paramter

# View Model Factory

## Summary Flow

1. You pass the factory to ViewModelProvider.
2. ViewModelProvider calls create() in your factory.
3. Factory creates the ViewModel using your custom constructor.
4. ViewModelProvider gives you the instance.
5. It caches that instance until the lifecycle ends.

# Thank You