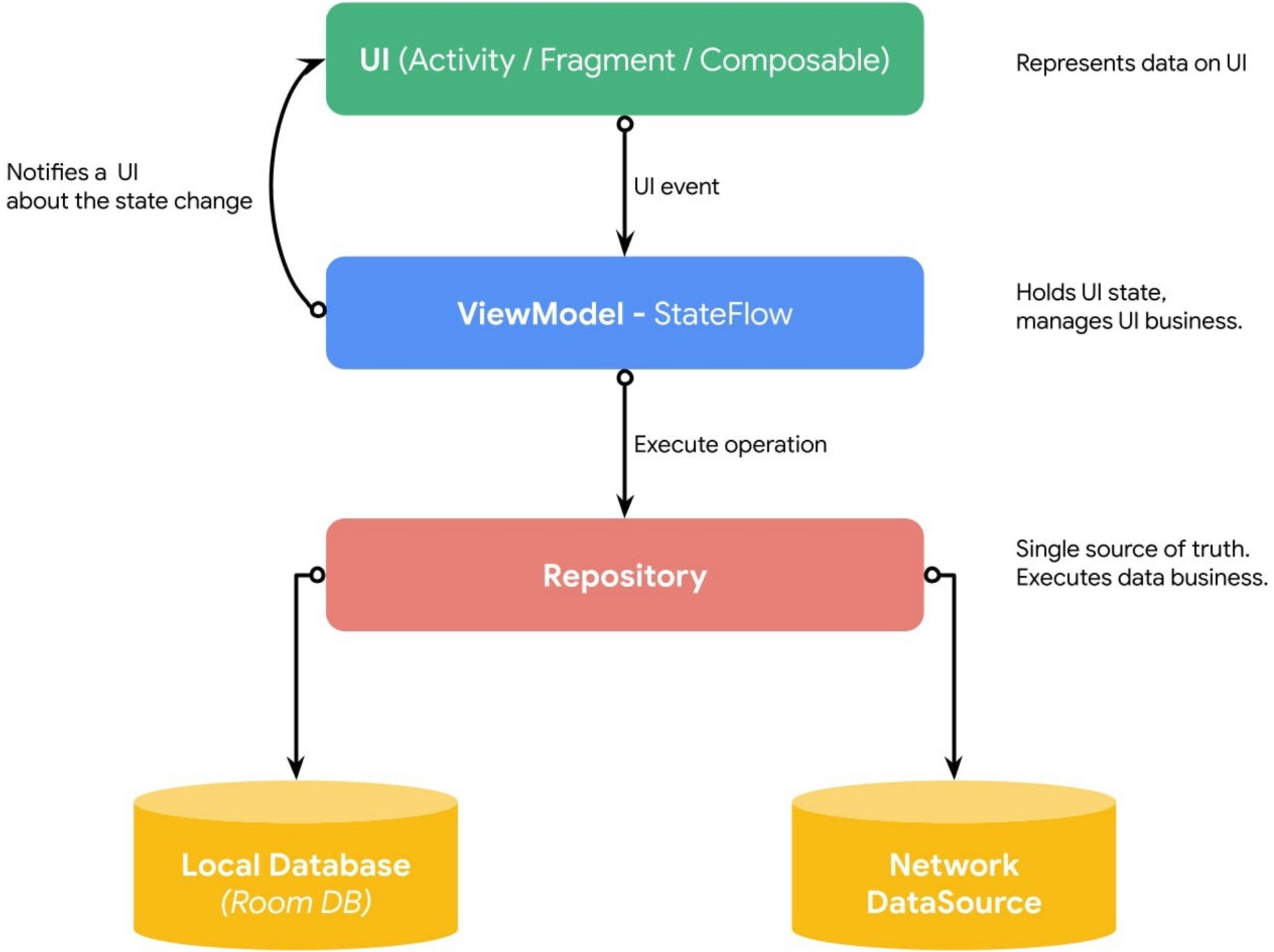
App data e files

Room

Lezione 9.3

App data e files: Overview

- Android utilizza un file system simile ai file system basati su disco di altre piattaforme
- Il sistema offre diverse opzioni per salvare i dati dell'app:
  1. App-specific storage
     - archivia i file destinati esclusivamente all'uso dell'app, in directory dedicate all'interno di un volume di archiviazione interno o in diverse directory dedicate all'interno dei un volume di archiviazione esterna. NB: Utilizzare le directory all'interno della memoria interna per salvare informazioni riservate a cui altre app non dovrebbero accedere
  2. Shared storage:
     - archivia i file che la tua app intende condividere con altre app, inclusi file multimediali, documenti e altri file
  3. Preferences:
     - archivia i dati primitivi, privati in coppie chiave-valore (key-value data)
  4. Database:
     - archivia i dati strutturati in un database privato utilizzando la libreria di persistenza Room


UI (Activity / Fragment / Composable)
Represents data on UI
Notifies a UI
about the state change
UI event
ViewModel - StateFlow
Holds UI state,
manages UI business.
Execute operation
Repository
Single source of truth.
Executes data business.
Local Database
(Room DB)
Network
DataSource

4. Database locale con Room

- *Room* offre uno strato di astrazione su SQLite per consentire un accesso fluido al database sfruttando la piena potenza di SQLite
- Le app che gestiscono quantità non banali di dati strutturati possono trarre grandi vantaggi dal memorizzare tali dati localmente. Il caso d'uso più comune è memorizzare parti di dati rilevanti (usando il DB anche come cache)
  - In questo modo, quando il dispositivo non è in grado di accedere alla rete, l'utente può comunque sfogliare quel contenuto mentre è offline. Eventuali modifiche al contenuto avviate dall'utente vengono quindi sincronizzate con il server dopo che il dispositivo è tornato online
- Poiché Room si occupa di queste gestioni «per te», è consigliato fortemente anziché usare direttamente SQLite

Dipendenze (build.gradle)

```
dependencies {
    val room_version = "2.6.1"

    implementation("androidx.room:room-runtime:$room_version")
    annotationProcessor("androidx.room:room-compiler:$room_version")

    // To use Kotlin annotation processing tool (kapt)
    kapt("androidx.room:room-compiler:$room_version")
    // To use Kotlin Symbol Processing (KSP)
    ksp("androidx.room:room-compiler:$room_version")

    // optional - Kotlin Extensions and Coroutines support for Room
    implementation("androidx.room:room-ktx:$room_version")

    // optional - RxJava2 support for Room
    implementation("androidx.room:room-rxjava2:$room_version")

    // optional - RxJava3 support for Room
    implementation("androidx.room:room-rxjava3:$room_version")

    // optional - Guava support for Room, including Optional and ListenableFuture
    implementation("androidx.room:room-guava:$room_version")

    // optional - Test helpers
    testImplementation("androidx.room:room-testing:$room_version")

    // optional - Paging 3 Integration
    implementation("androidx.room:room-paging:$room_version")
}
```

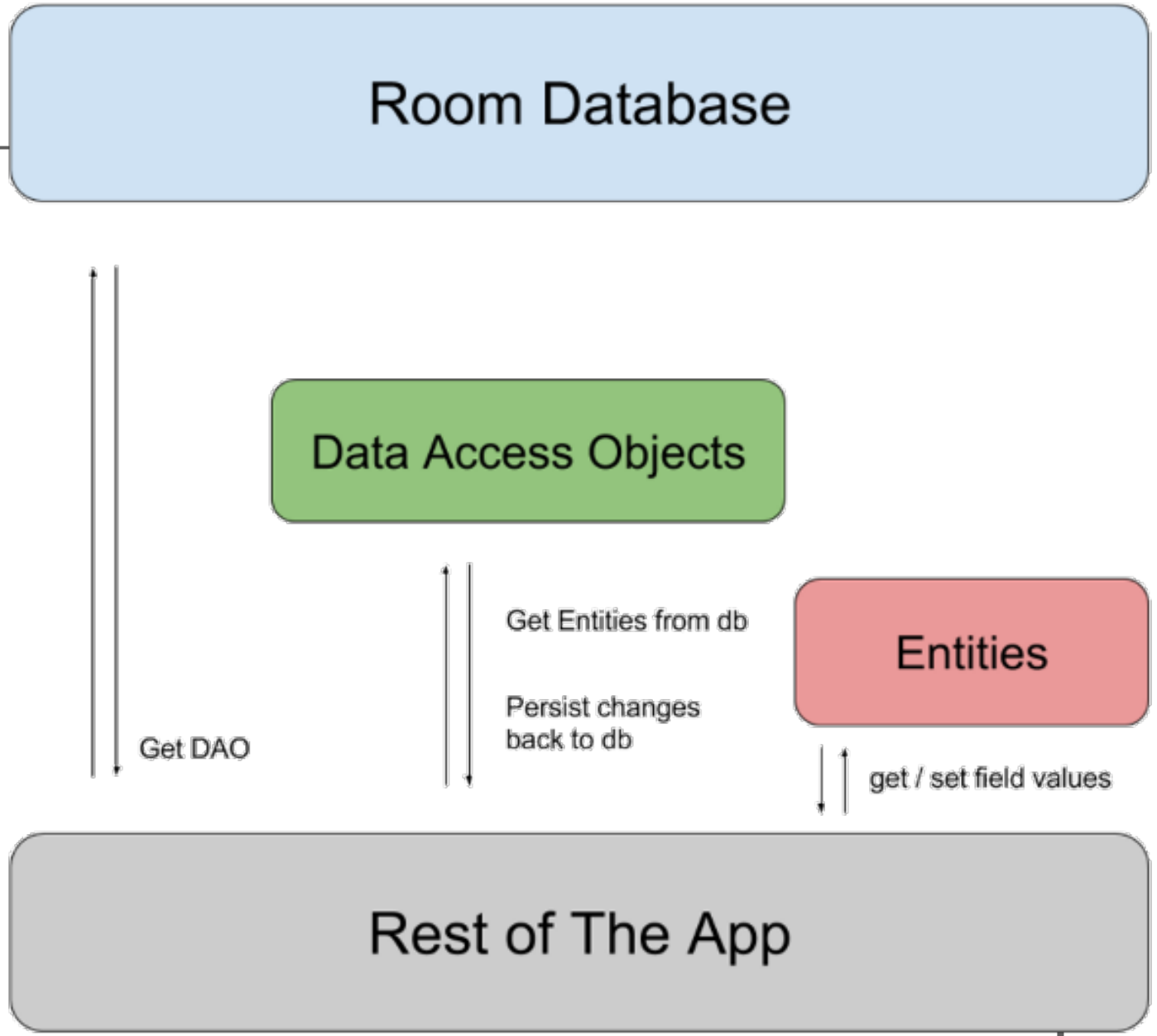
Componenti di Room

- **Database**
  - Contiene il database holder e funge da punto di accesso principale per la connessione sottostante ai dati relazionali persistenti dell'app
  - La classe annotata con *@Database* dovrebbe soddisfare le seguenti condizioni:
    - classe astratta che estende *RoomDatabase*
    - Includere l'elenco delle entità associate al database all'interno dell'annotazione
    - contenere un metodo astratto che ha 0 argomenti e restituisce la classe che è annotata con @Dao
  - In fase di esecuzione, è possibile acquisire un'istanza di *Database* chiamando *Room.databaseBuilder()* o *Room.inMemoryDatabaseBuilder()*
- **Data Entities**
  - Rappresentano le tabella nel database
- **DAO (*data access objects*)**
  - Contiene i metodi usati per accedere il database

DAO e entity

- L'app utilizza il database Room per ottenere i ***data access objects*** **o DAO** associati a quel database
- L'app utilizza quindi ciascun DAO per recuperare entità dal database (usando query SQL) e salvare le modifiche di tali entità (entities) nel database. Infine, l'app utilizza un'entità (**entity**) per ottenere e impostare valori che corrispondono alle colonne della tabella all'interno del database

- Relazione tra le diverse componenti



Esempio: @Entity == tabella

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

Esempio: @Entity

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

Mettere @Entity permette di dire a Room che oggetti si vogliono memorizzare. Ogni Entity rappresenta una tabella! Ogni istanza della classe rappresenta una riga della tabella creata.

NOTA: l'annotazione **@ColumInfo** serve SOLO se si vuole salvare il nome dei campi della tabella in SQLite con un nome diverso dal nome della variabile. In questo caso il nome effettivo nella tabella in SQLite sarà first_name mentre, in Java resterà firstName.

Esempio: @Dao

```
@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)
}
```

E ora.. Il database

- La classe del database deve soddisfare le seguenti condizioni:
  - La classe deve essere annotata con un'annotazione @Database che include un array di entità che elenca tutte le entità di dati associate al database
  - La classe deve essere una classe astratta che estende RoomDatabase
  - Per ogni classe DAO associata al database, la classe database deve definire un metodo astratto con zero argomenti e restituire un'istanza della classe DAO

Esempio: AppDatabase

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

- Dopo aver creato i precedenti tre file (User - @entity, AppDatabase - @database, e UserDao - @Dao), si può ottenere un'instanza del database creato con

```
val db = Room.databaseBuilder(
            applicationContext,
            AppDatabase::class.java, "database-name"
        ).build()
```

Esempio: AppDatabase

- Per ottenere un'istanza di DAO

```
val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```

E Ora..

- Vediamo tutto nel dettaglio!

Definire i dati con Room entity

- Usando la libreria di persistenza Room per archiviare i dati della tua app, occorre definire le entità per rappresentare gli oggetti che occorre archiviare
- Ogni entità (entity) corrisponde a una tabella nel database Room associato e ogni istanza di un'entità rappresenta una riga di dati nella tabella corrispondente
  - Ciò significa che puoi utilizzare le entità Room per definire lo schema del tuo database senza scrivere alcun codice SQL

Definire un’entity

```
@Entity
data class User(
    @PrimaryKey val id: Int,

    val firstName: String?,
    val lastName: String?
)
```

Chiave privata

- Ogni entità deve definire almeno 1 campo come chiave primaria
  - NB: Anche quando esiste solo 1 campo, è comunque necessario annotare il field con l'annotazione @PrimaryKey

```
@PrimaryKey val id: Int
```

Chiave privata

- Ogni entità deve definire almeno 1 campo come chiave primaria
  - Se l'entità ha una chiave primaria composta, è possibile utilizzare la proprietà primaryKeys dell'annotazione @Entity

```
@Entity(primaryKeys = ["firstName", "lastName"])
data class User(
    val firstName: String?,
    val lastName: String?
)
```

Nome tabella

- Per impostazione predefinita, Room utilizza il nome della classe come nome della tabella del database.
- **SE** si desidera che la tabella abbia un nome diverso (per coerenza con le indicazioni SQL di nomenclatura), si può usare la proprietà tableName dell'annotazione @Entity per rinominare la tabella a livello di SQLite

```
@Entity(tableName = "users")
data class User (
    ….
)
```

Nome campi della tabella

- Nello stesso modo che per tableName, Room utilizza i nomi dei field come nomi di colonna nel database. Se desideri che una colonna abbia un nome diverso, aggiungi l'annotazione @ColumnInfo a un field

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

In questo modo, in Java le variabili seguono la nomenclatura di Java, mentre in SQLite, i campi seguono la nomeclatura di SQL

unique = true

- Se si vuole che alcuni fields siano univoci (ovvero NO più righe con stessi valori)

```
@Entity(indices = [Index(value = ["first_name",
"last_name"],
        unique = true)])
data class User(
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "first_name") val firstName:
String?,
    @ColumnInfo(name = "last_name") val lastName:
String?,
    @Ignore var picture: Bitmap?
)
```

Annotazione @Embedded

- Se si vuole esprimere un entity o un data object come un insieme coerente nella logica del database, anche se l'oggetto contiene diversi campi, è possibile utilizzare l'annotazione @Embedded per rappresentare un oggetto che si desidera scomporre nei suoi sottocampi all'interno di una tabella. È quindi possibile eseguire una query sui campi incorporati proprio come si farebbe per altre singole colonne

@Embedded

```
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?
)
```

Address rappresenta una composizione di campi denominati street, city, state, e postCode. Per memorizzare le colonne composte separatamente nella tabella, includere un campo address nella classe User che è annotato con @Embedded

La tabella che rappresenta un oggetto User contiene quindi colonne con i seguenti nomi: id, firstName, street, city, state, e postCode

Accedere ai dati con Room DAO

- Per accedere ai dati della tua app utilizzando la libreria di persistenza Room, si devono usare gli oggetti di accesso ai dati (data access objects) o DAO
  - Questo insieme di oggetti DAO costituisce il componente principale di Room, poiché ogni DAO include metodi che offrono un accesso astratto al database dell'app
- Un DAO valida l'SQL in fase di compilazione e lo associa ad un metodo
- Accedendo a un database utilizzando una classe DAO anziché generatori di query o query dirette, è possibile separare diversi componenti dell'architettura del database

DAO

- Un **DAO** può essere **un'interfaccia o una classe astratta**
  - Se è una classe astratta, può avere opzionalmente un costruttore che accetta RoomDatabase come unico parametro
- Room crea ogni implementazione DAO al momento della compilazione

Insert

```
@Dao
interface UserDao {
    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)

    @Query("SELECT * FROM user")
    fun getAll(): List<User>
}
```

NOTA:
Usando @Insert non occorre definire la query SQL! Room in automatico esegue la query sql di INSERT

INSERT e onConflict

- Usando @Insert, si può definire cosa occorre fare se l'elemento da inserire esiste già nel database, usando la proprietà @OnConflit
  - ABORT: se c'è un conflitto, l'operazione viene annullata (*rolled back)*
  - IGNORE: ignorare l'inserimento se c'è un conflitto (restituendo -1), ma non annulla operazione
  - REPLACE: sostituire la vecchia istanza con la nuova

INSERT e onConflict: esempio

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

    @Insert
    fun insertBothUsers(user1: User, user2: User)

    @Insert
    fun insertUsersAndFriends(user: User, friends: List<User>)
}
```

Update

- Il metodo update modifica un insieme di entità, fornite come parametri, nel database. Utilizza una query che trova la corrispondenza con la chiave primaria di ciascuna entità

```
@Dao
interface UserDao {
    @Update
    fun updateUsers(vararg users: User)
}
```

Delete

- Il metodo Delete rimuove una serie di entità, fornite come parametri, dal database. Utilizza le chiavi primarie per trovare le entità da eliminare

```
@Dao
interface UserDao {
    @Delete
    fun deleteUsers(vararg users: User)
}
```

Query

- @Query è l'annotazione principale utilizzata nelle classi DAO. Consente di eseguire operazioni di lettura/scrittura su un database
- NB: Ogni metodo @Query viene verificato al momento della compilazione, quindi se si verifica un problema con la query, si verifica un errore di compilazione anziché un errore di runtime
- Room verifica inoltre il valore restituito della query in modo tale che se il nome del campo nell'oggetto restituito non corrisponde ai nomi di colonna corrispondenti nella risposta della query, Room avvisa l'utente in uno dei due modi seguenti:
  - Fornisce un avviso se solo alcuni nomi di campo corrispondono
  - Dà un errore se nessun nome di campo corrisponde

@Query

```
@Query("SELECT * FROM user")
fun loadAllUsers(): Array<User>
```

Questa è una query molto semplice che carica tutti gli utenti
Al momento della compilazione, Room sa che sta eseguendo una query su tutte le colonne nella tabella User
Se la query contiene un errore di sintassi o se la tabella utente non esiste nel database, Room visualizza un errore con il messaggio appropriato durante la compilazione dell'app

Restituire sottoinsieme di colonne

```
data class NameTuple(
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

```
@Query("SELECT first_name, last_name
FROM user")
fun loadFullName(): List<NameTuple>
```

Il più delle volte, è necessario ottenere solo pochi campi di un'entità. Ad esempio, l'interfaccia utente potrebbe visualizzare solo il nome e il cognome di un utente, anziché tutti i dettagli relativi all'utente. Recuperando solo le colonne visualizzate nell'interfaccia utente dell'app, si risparmiano risorse preziose e la query viene completata più rapidamente.

Passare parametri alla query

- Il più delle volte, è necessario passare dei parametri nelle query per eseguire operazioni di filtro, ad esempio per visualizzare solo gli utenti di età superiore a una determinata età

```
@Query("SELECT * FROM user WHERE age > :minAge")
fun loadAllUsersOlderThan(minAge: Int): Array<User>
```

Quando questa query viene elaborata al momento della compilazione, Room fa il match tra il parametro :minAge con il parametro del metodo minAge. Room esegue la corrispondenza utilizzando i nomi dei parametri. In caso di mancata corrispondenza, si verifica un errore durante la compilazione dell'app

Passare parametri alla query

- Puoi anche passare più parametri o fare riferimento a loro più volte in una query

```
@Query("SELECT * FROM user WHERE age BETWEEN :minAge AND
:maxAge")
fun loadAllUsersBetweenAges(minAge: Int, maxAge: Int):
Array<User>

@Query("SELECT * FROM user WHERE first_name LIKE :search
" +
       "OR last_name LIKE :search")
fun findUserWithName(search: String): List<User>
```

Passare una collezione di argomenti

- Alcuni dei metodi DAO potrebbero richiedere di passare un numero variabile di parametri che non è noto fino al runtime.
- Room comprende quando un parametro rappresenta una raccolta e la espande automaticamente in fase di esecuzione in base al numero di parametri forniti
- Ad esempio, il codice seguente definisce un metodo che restituisce informazioni su tutti gli utenti da un sottoinsieme di regioni:

```
@Query("SELECT * FROM user WHERE region IN
(:regions)")
fun loadUsersFromRegions(regions: List<String>):
List<User>
```

Querying di varie tabelle

- Alcune delle tue query potrebbero richiedere l'accesso a più tabelle per calcolare il risultato.
- È possibile utilizzare le clausole JOIN nelle query SQL per fare riferimento a più di una tabella
- Il codice seguente definisce un metodo che unisce tre tabelle insieme per restituire i libri che sono attualmente in prestito a un utente specifico:

```
@Query(
    "SELECT * FROM book " +
    "INNER JOIN loan ON loan.book_id = book.id " +
    "INNER JOIN user ON user.id = loan.user_id " +
    "WHERE user.name LIKE :userName"
)
fun findBooksBorrowedByNameSync(userName: String): List<Book>
```

Querying di varie tabelle

- È inoltre possibile definire semplici objects per restituire un sottoinsieme di colonne da più tabelle unite
- Il codice seguente definisce un DAO con un metodo che restituisce i nomi degli utenti e i nomi dei libri che hanno preso in prestito:

- ```
  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>>

      // You can also define this class in a separate file.
      data class UserBook(val userName: String?, val bookName: String?)
  }
  ```
  ```

Creare relazioni in Room

- Ci sono due approcci per creare relazioni in room
  - **Intermediate data class** -> in questo caso si definisci una classe di dati che modella la relazione tra le entità Room
    - Questa classe di dati contiene gli accoppiamenti tra le istanze di un'entità e le istanze di un'altra entità come oggetti **embedded**. I tuoi metodi di query possono quindi restituire istanze di questa classe di dati da utilizzare nella tua app
  - **Multimap return type** -> in questo caso non è necessario definire classi di dati aggiuntive. Invece, si definisce un tipo restituito multimap per il tuo metodo in base alla struttura della mappa che desideri e definisci la relazione tra le tue entità direttamente nella tua query SQL

Creare relazioni in Room

- Che approccio usare?
  - Dipende!
- Room li supporta entrambi ovviamente!
  - L'approccio intermediate data class evita di scrivere query sql complesse ma richiede data class aggiuntive
  - Il multimap return type richiede alle query SQL di fare la maggior parte del lavoro
- Android dice: se non hai un motivo particolare per usare «intermediate data classes», allora conviene usare multimap return type

Restituire un **multimap**

- Da Room 2.4, puoi eseguire query su colonne da più tabelle senza definire una classe di dati aggiuntiva scrivendo metodi di query che restituiscono una **multimap** (come tipo di ritorno)

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

Restituire un **multimap**

- Poi quindi usare query che utilizzano «group by»

```
@Query(
    "SELECT * FROM user" +
    "JOIN book ON user.id = book.user_id" +
    "GROUP BY user.name WHERE COUNT(book.id) >= 3"
)
fun loadUserAndBookNames(): Map<User, List<Book>>
```

Restituire un **multimap**

- Se non hai bisogno di mappare interi oggetti, puoi anche restituire mappature tra colonne specifiche nella tua query impostando gli attributi keyColumn e valueColumn in un'annotazione @MapInfo sul tuo metodo di query:

```
@MapInfo(keyColumn = "userName", valueColumn = "bookName")
@Query(
    "SELECT user.name AS username, book.name AS bookname FROM user" +
    "JOIN book ON user.id = book.user_id"
)
fun loadUserAndBookNames(): Map<String, List<String>>
```

Intermediate data class

- Ora vediamo l’approccio con intermediate data class

Definire una relazione one-to-one

- Una relazione uno a uno tra due entità è una relazione in cui ogni istanza dell'entità padre corrisponde esattamente a un'istanza dell'entità figlio e viceversa
  - Ad esempio, considera un'app di streaming musicale in cui l'utente ha una libreria di brani di sua proprietà. Ogni utente ha una sola libreria e ogni libreria corrisponde esattamente a un utente. Pertanto, dovrebbe esistere una relazione uno a uno tra l'entità Utente e l'entità Libreria
- Innanzitutto, bisogna creare una classe per ciascuna delle tue due entità. Una delle entità deve includere una variabile che è un riferimento alla chiave primaria dell'altra entità

Definire una relazione one-to-one

```
@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
)
```

Le due classi corrispondenti alle due entity

Definire una relazione one-to-one

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

Definire una relazione one-to-one

- Nella classe DAO

@Transaction permette di garantire che ‘l’intera operazione venga eseguita in modo atomico da Room

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

Relazione one-to-many

- Una relazione uno-a-molti tra due entità è una relazione in cui ogni istanza dell'entità padre corrisponde a zero o più istanze dell'entità figlio, ma ogni istanza dell'entità figlio può corrispondere esattamente a un'istanza dell'entità padre
  - Nell'esempio dell'app di streaming musicale, supponiamo che l'utente abbia la possibilità di organizzare i propri brani in playlist. Ogni utente può creare tutte le playlist che desidera, ma ogni playlist è creata esattamente da un utente. Pertanto, dovrebbe esserci una relazione uno-a-molti tra l'entità Utente e l'entità Playlist

Relazione one-to-many

```
@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
)
```

Innanzitutto, occorre creare una classe per ciascuna delle tue due entità. Come nell'esempio precedente, l'entità figlio deve includere una variabile che sia un riferimento alla chiave primaria dell'entità padre

Relazione one-to-many

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

Relazione one-to-many

- Nella classe DAO

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

Relazione many-to-many

- Una relazione molti-a-molti tra due entità è una relazione in cui ogni istanza dell'entità padre corrisponde a zero o più istanze dell'entità figlio e viceversa
  - Nell'esempio dell'app per lo streaming musicale, considera di nuovo le playlist definite dall'utente. Ogni playlist può includere molti brani e ogni brano può far parte di diverse playlist. Pertanto, ci dovrebbe essere una relazione molti-a-molti tra l'entità Playlist e l'entità Song
- Innanzitutto, si deve creare una classe per ciascuna delle tue due entità. Poi, **crea una terza classe per rappresentare un'entità associativa** (o tabella di riferimento incrociato) tra le due entità. **La tabella dei riferimenti incrociati deve contenere colonne per la chiave primaria di ciascuna entità nella relazione molti-a-molti rappresentata nella tabella**
  - Nell’esempio, ogni riga nella tabella dei riferimenti incrociati corrisponde a un'associazione di un'istanza della playlist e un'istanza della song in cui la canzone a cui viene fatto riferimento è inclusa nella playlist a cui si fa riferimento

Relazione many-to-many

```
@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
)
```

Relazione many-to-many

- Il passaggio successivo dipende da come si desidera interrogare queste entità correlate
  - Se si desidera interrogare playlist e un elenco delle canzoni corrispondenti per ciascuna playlist, occorre creare una nuova classe di dati che contiene un singolo oggetto Playlist e un elenco di tutti gli oggetti Song inclusi nella playlist
  - Se si desidera eseguire la query di brani e un elenco delle playlist corrispondenti per ciascuno, creare una nuova classe di dati che contiene un singolo oggetto Song e un elenco di tutti gli oggetti della playlist in cui è inclusa la canzone
- In entrambi i casi, occorre modellare la relazione tra le entità utilizzando la proprietà *associateBy* nell'annotazione *@Relation* in ciascuna di queste classi per identificare l'entità di riferimento incrociato che fornisce la relazione tra l'entità Playlist e l'entità Song

Relazione many-to-many

```
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>
)
```

Le Due alternative!

Relazione many-to-many

- Poi occorre aggiungere un metodo nella casse DAO per esporre le funzionalità query di cui l'app ha bisogno
  - getPlaylistsWithSongs: questo metodo esegue una query sul database e restituisce tutti gli oggetti PlaylistWithSongs risultanti
  - getSongsWithPlaylists: questo metodo esegue una query sul database e restituisce tutti gli oggetti SongWithPlaylists risultanti
- Entrambi i metodi necessitano di Room per eseguire le due query, quindi va aggiunta l'annotazione @Transaction

Relazione many-to-many

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

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

Scrivere query DAO asincrone

- Per evitare che le query blocchino l'interfaccia utente, Room non consente l'accesso al database sul thread principale
- Questa restrizione significa che devi rendere asincrone le tue query DAO
- La libreria Room include integrazioni con diversi framework per fornire l'esecuzione di query asincrone
- Le query DAO rientrano in tre categorie:
  - *One-shot write* che inseriscono, aggiornano o eliminano i dati nel database.
  - *One-shot read* che leggono i dati dal database solo una volta e restituiscono un risultato con lo snapshot del database in quel momento
  - *Observable read* queries che leggono i dati dal database ogni volta che le tabelle del database sottostanti cambiano ed emettono nuovi valori per riflettere tali modifiche

In Kotlin..

Query type	Kotlin language features
One-shot write	Coroutines (**suspend**)
One-shot read	Coroutines (**suspend**)
Observable read	`Flow<T>`

Kotlin with Flow and couroutines

- Kotlin fornisce language features che consentono di scrivere query asincrone senza framework di terze parti:
  - In Room 2.2 e versioni successive, puoi utilizzare la funzionalità Flow di Kotlin per scrivere query osservabili
  - In Room 2.1 e versioni successive, puoi utilizzare la parola chiave suspend per rendere asincrone le tue query DAO utilizzando le coroutine Kotlin

Asynchronous one-shot queries

```
@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>
}
```

Observable queries

```
@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>>
}
```

Creare Viste in un Database

- La versione 2.1.0 e successive della libreria di persistenza Room fornisce supporto per le viste del database SQLite, consentendo di incapsulare una query in una classe
- Room si riferisce a queste query-backed classes come View e si comportano allo stesso modo dei semplici oggetti dati quando utilizzati in un DAO

Creare una Vista

- Per creare una vista, aggiungi l'annotazione @DatabaseView a una classe, imposta il valore dell'annotazione sulla query che la classe dovrebbe rappresentare

```
@DatabaseView("SELECT user.id, user.name, user.departmentId," +
        "department.name AS departmentName FROM user " +
        "INNER JOIN department ON user.departmentId = department.id")
data class UserDetail(
    val id: Long,
    val name: String?,
    val departmentId: Long,
    val departmentName: String?
)
```

Associare la vista con il database

- Per includere questa vista come parte del database dell'app, includere la proprietà views nell'annotazione @Database dell'app:

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

DataBase e Singleton

- Il database dovrà anche definire un **Singleton** in modo tale da avere sempre un'unica istanza del db
- Per restituire questo Singleton è necessario un getter che deve creare il database - se non ancora creato - tramite databaseBuilder (di Room)

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

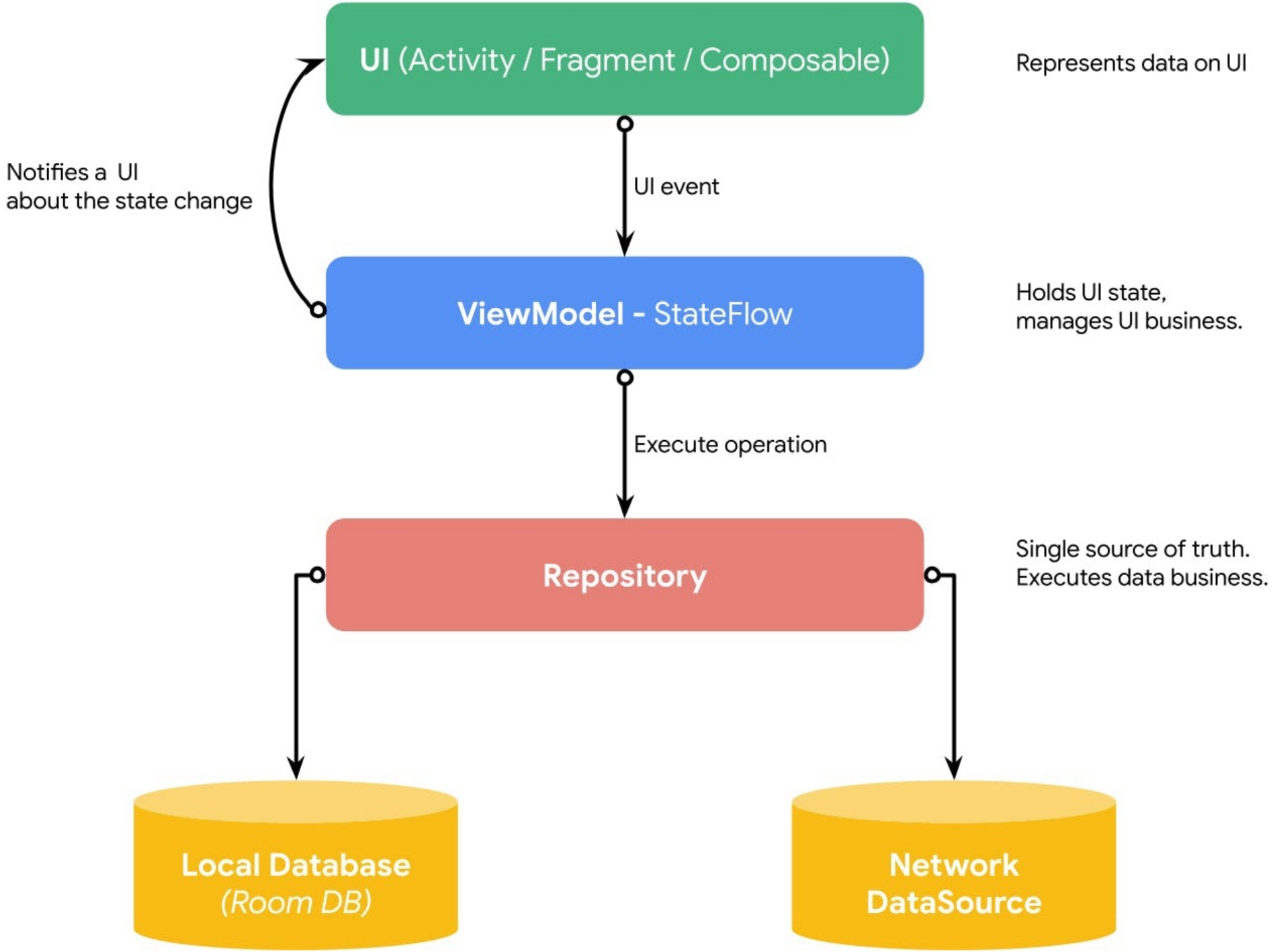
    abstract fun itemDAO(): ItemDAO

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

        fun getDatabase(context: Context): ItemDatabase {
            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    ItemDatabase::class.java,
                     name: "items_database"
                ).build()
                INSTANCE = instance

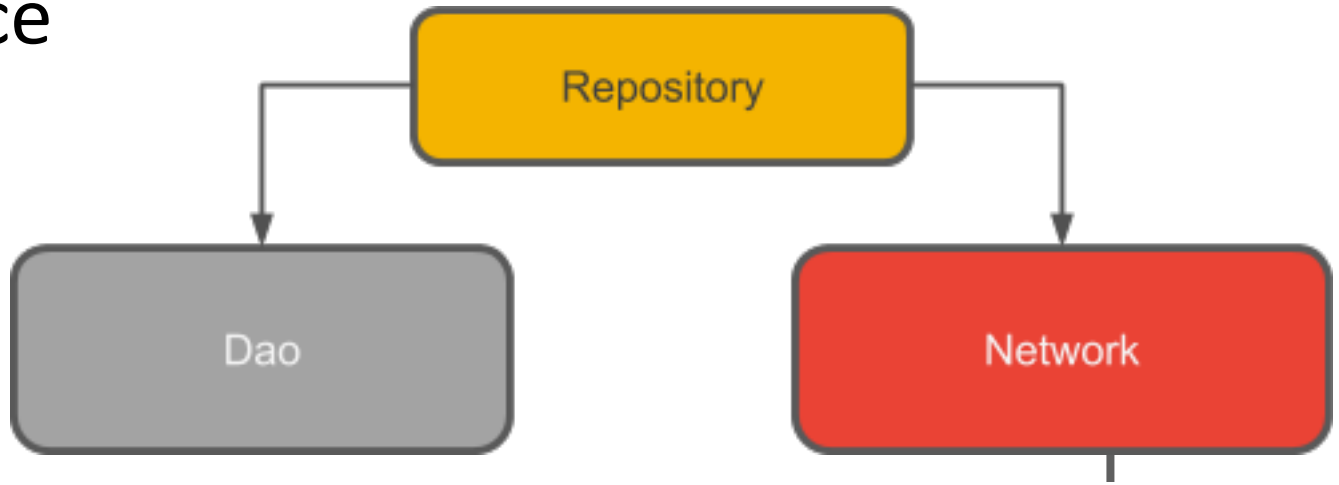
                instance ^synchronized
            }
        }
    }

}
```


UI (Activity / Fragment / Composable)
Represents data on UI
Notifies a UI
about the state change
UI event
ViewModel - StateFlow
Holds UI state,
manages UI business.
Execute operation
Repository
Single source of truth.
Executes data business.
Local Database
(Room DB)
Network
DataSource

Repository

- Una classe Repository astrae l'accesso a più origini dati (lo avevamo visto quando abbiamo parlato del ViewModel)
- Il repository non fa parte delle librerie Architecture Components, ma è una procedura consigliata per la separazione del codice e l'architettura
- Una classe Repository fornisce un'API «pulita» per l'accesso ai dati al resto dell'applicazione



Perché usarlo?

- Un repository gestisce le query e ti consente di utilizzare più backend
- Nell'esempio più comune, il Repository implementa la logica per decidere se recuperare i dati da una rete o utilizzare i risultati memorizzati nella cache in un database locale

Repository

- Ogni repository deve avere al suo interno il riferimento al DAO, in modo da poter richiamare i suoi metodi
- Per ottenere il DAO, è necessario recuperarlo tramite il database

Pre-Popolare un database

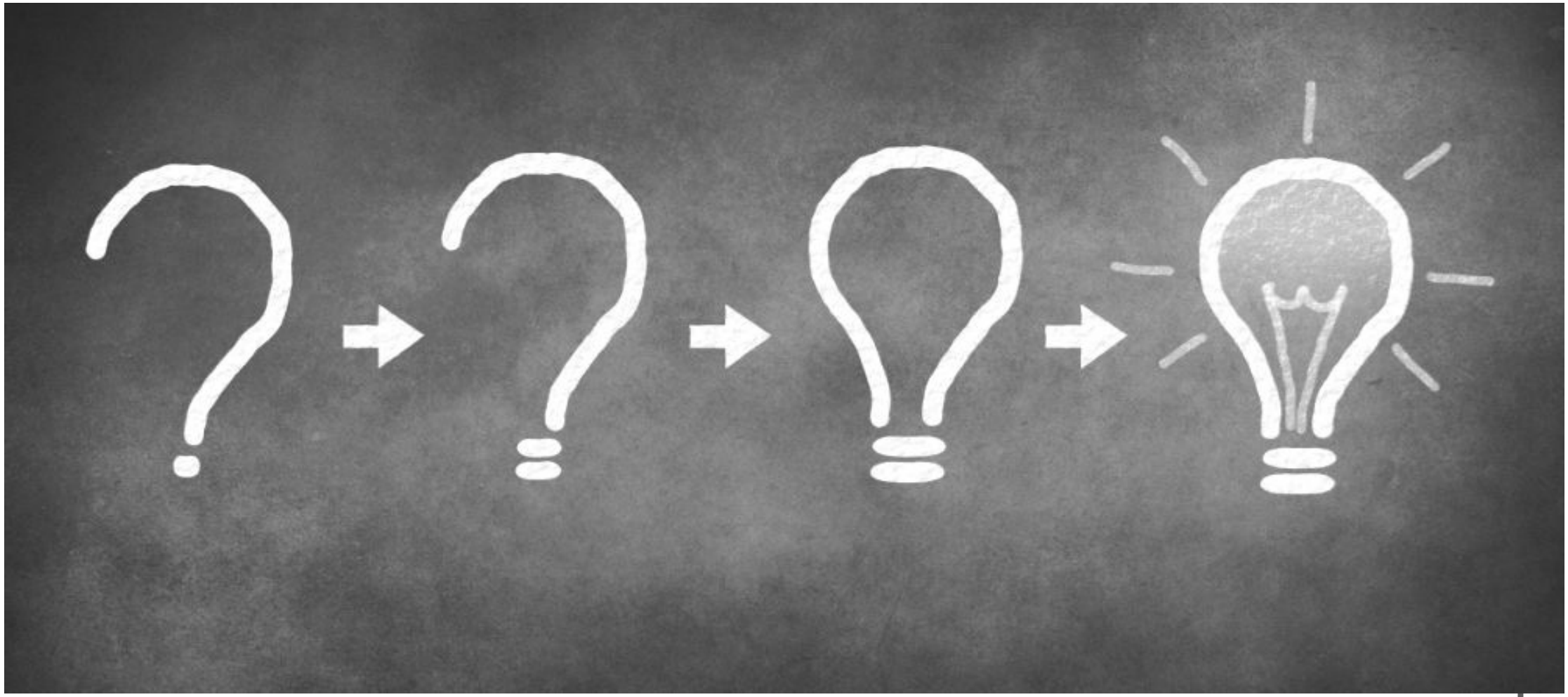
- Se c'è il bisogno di pre-popolare il database con uno specifico set di dati, si può fare utilizzando metodi API con Room 2.2.0 e versioni successive al momento dell'inizializzazione con i contenuti di un file di database preconfezionato nel file system del dispositivo

```
//se avete il file salvato nel device
Room.databaseBuilder(appContext, AppDatabase::class.java, "Sample.db")
    .createFromFile(File("mypath"))
    .build()

//se avete il file salvato in app\src\main\assets
Room.databaseBuilder(appContext, AppDatabase::class.java, "Sample.db")
    .createFromAsset("database/myapp.db")
    .build()
```

Per maggiori dettagli: https://developer.android.com/training/data-storage/room/prepopulate

Domande???



References e Approfondimenti

- https://developer.android.com/training/data-storage/room
- https://developer.android.com/reference/androidx/room/FtsOptions
- https://developer.android.com/training/data-storage/room/referencing-data#understand-no-object-references
- https://developer.android.com/training/data-storage/room/accessing-data?skip_cache=true#query-multiple-tables
- https://developer.android.com/training/data-storage/room/accessing-data#additional-resources
- https://developer.android.com/training/data-storage/room/accessing-data#multimap