Persistenza: Room library

Programmazione Mobile

A.A. 2021/22

M.O. Spata

Room

- Room è una libreria che permette di implementare un db SQL in un’app, fornendo uno livello di astrazione su SQLite.
- Una delle incombenze che si ha nell’usare db relazionali è quella di creare e mantenere il mapping tra le tabelle che permettono di memorizzare i dati e gli oggetti dell’app che permettono alla stessa di usarli (Object Relational Mapping).

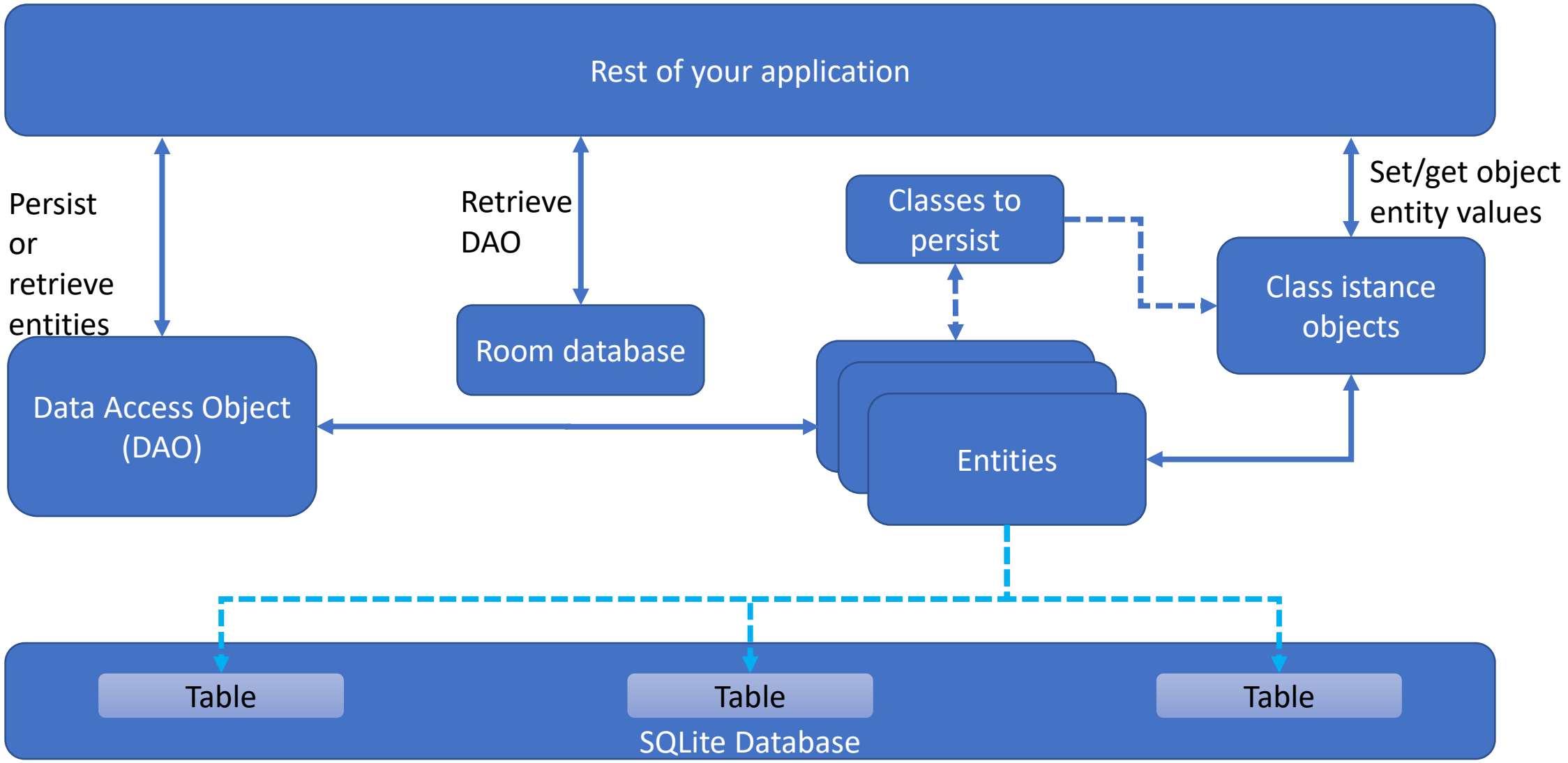
Architettura di Room

- La libreria Room, grazie ad un meccanismo di annotazione nella definizione delle classi, permette di semplificare l'ORM ovvero l'associazione tra gli attributi della classe e le colonne delle tabelle e tra i metodi della classe e le istruzioni SQL.
- Il modello di persistenza Room richiede la definizione di almeno tre componenti
  - Entity
  - Data Access Object
  - Room Database

I tre componenti di Room

- **Entity**: si tratta di un insieme di class con la annotazione ***@Entity*** che permettono di definire la struttura delle tabelle del database.
- **Data Access Object**: Una classe con l'annotazione ***@Dao*** che permette di definire i metodi utilizzati per modificare o interrogare il db
- **Database**: una classe astratta che estende RoomDatabase, annotata con ***@Database*** e che rappresenta l'accesso principale alla connessione SQLite; deve implementare un metodo che restituisce la classe Dao e una lista delle Entity che il db conterrà

Componenti principali di Room



Esempio: Entity

Saranno inclusi nella definizione della tabella tutti gli attributi public e quelli private dotati di get e set.

Il costruttore della classe dovrebbe contenere tutte gli attributi che definiscono la tabella

Se un attributo non deve essere incluso come colonna della tabella si deve annotare con ***@Ignore***

```
@Entity
public class Hoard {
        @NotNull
        @PrimaryKey
        public String HoardName;
        public int Gold Hoarded;
        public boolean HoardAccessible;
}
```

Esempio

```
@Entity(tableName = "customers")
class Customer {
    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "customerId")
    private int id;
    @ColumnInfo(name = "customerName")
    private String name;
    private String address;
    public Customer(String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }
    public int getId() {
        return this.id;
    } public String getName() {
        return this.name;
    }
    public String getAddress() {
        return this.address;
    }
    public void setId(@NonNull int id) {
        this.id = id;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(int quantity) {
        this.address = address;
    }
}
```

Esempio

```
@Entity(foreignKeys = @ForeignKey(entity = Customer.class,
        parentColumns = "id", childColumns = "purchaseId"))
 class Purchase {
    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "purchaseId")
    private int id;
    @ColumnInfo(name = "productName")
    private String name;


}
```

RoomDatabase

- Quando sono state definite tutte le class **Entity** si dovrà creare la nuova classe astratta **RoomDatabase** con l'annotazione **@Database** che deve includere una lista delle entità del db e la versione corrente

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

- Adesso sarà necessario creare la classe DAO che dovrà essere restituita dal RoomDatabase.

Creazione di più classi Dao

- Se il db contiene più tabelle è una good practice avere più classi Dao.
- In realtà si tratta di una interface perché presenterà solo metodi astratti annotati opportunamente che di fatto permetteranno l'interazione con il DB (insert, update, delete e query)

```
@Dao
public interface CustomerDao {
}
```

Esempio: @Insert

```
@Insert
void addCustomer(Customer customer);

@Insert
public void insertCustomers(Customer... customers);
```

```
@Dao
public interface HoardDAO {
      // Insert a list of hoards, replacing stored
      // hoards using the same name
      @Insert (onConflict = OnConflictStrategy.REPLACE)
      public void insertHoards (List<Hoard> hoards);
}
```

Alternative al parametro REPLACE

- Lista di parametri alternativi a REPLACE:
  - **ABORT**: cancella la transazione in corso
  - **FAIL**: Causa il fallimento della transazione corrente
  - **IGNORE**: Ignorare i conflitti dei nuovi dati e continuare la transazione
  - **REPLACE**: Sovrascrivere il valore esistente con il nuovo valore fornito e continuare la transazione
  - **ROLLBACK**: Torna indietro alla transazione corrente, annullando tutti i cambiamenti fatti

@Update, @Delete

```
@Update
public void updateCustomers(Customer... customers);

@Delete
public void deleteCustomers(Customer... customers);

@Delete
public int deleteCustomers(Customer... customers);

@Query("DELETE FROM customers WHERE name = :name")
void deleteCustomer(String name);
```

@Query

```
@Query("SELECT * FROM customers WHERE name = :customerName")
List<Customer> findCustomer(String customerName);

@Query("SELECT * FROM customers WHERE name IN (:customerNames)")
List<Customer> findCustomer(String[] customerNames);
```

- Si possono usare le funzioni di aggregazione
- Si possono fare le proiezioni (definendo una classe con un sottoinsieme di attributi public che corrispondono ai campi da restituire)

RoomDatabase

- La classe **RoomDatabase** deve contenere metodi astratti che restituiscono i Dao:

```
@Database (entities = {Hoard.class}, version = 1)
public abstract class HoardDatabase extends RoomDatabase{
      public abstract HoardDAO hoardDAO();
}
```

Creare un db…

```
public class HoardDatabaseAccessor {
        private static HoardDatabase HoardDatabaseInstance;
        private static final String HOARD_DB_NAME = "hoard_db";
        private HoardDatabaseAccessor() {}

        public static HoardDatabase getInstance(Context context) {
                if(HoardDatabaseInstance == null) {
                        //Create or open a new SQLite database, and return it as
                        //a Room Database instance
                        HoardDatabaseInstance = Room.databaseBuilder(context,
                                                HoardDatabase.class, HOARD_DB_NAME).build();
                }
                return HoardDatabaseInstance;
        }
}
```

Ed interagire con il db creato precedentemente…

```
//Access the Hoard Database instance
HoardDatabase hoardDB = HoardDatabaseAccessor.getInstance(getApplicationContext());
//Add a new hoards to the database
hoardDB.hoardDAO().insertHoard(new Hoard("Smego1"), 1, true));
hoardDB.hoardDAO().insertHoard(new Hoard("Smaug"), 200000, false));
//Query the database
int totalGold = hoardDB.hoardDAO().totalGoldHoarded();
List<Hoard> allHoards = hoardDB.hoardDAO().loadAllHoards();
```