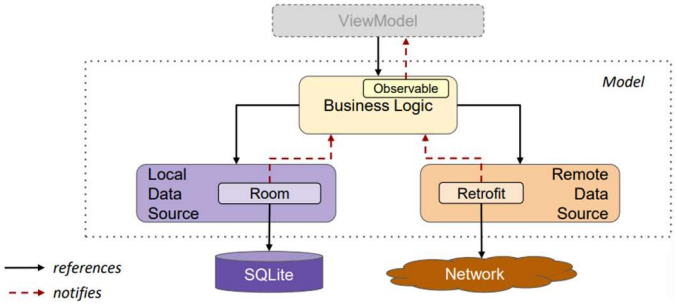
09-DataLayer

Model- data layer: parte responsabile per gestire i dati che la nostra applicazione usa, responsabile per la logica dell'applicazione e per mantenere inormazioni persistenti.

L'architettura proposta da google, si basa su vari sotto-componenti:

- **Business logic**: Domain model: mantiene i dettagli su applicazioni
  - Insieme di metodi per svolgere le operazioni che l'app fornisce agli utenti
    - *Creare team, vedere i task*
    - *Metodi garantiscono che si possano svolgere solo azioni valide.*
  - Mi permette di garantire il comportamento corretto dell'applicazione
  - Fornisce degli observable, internmente può avere dei LocalDataSource o RemoteDataSource o entrambe
- **Remote Data Source**: remote server dove la vera logica è implementata:
  - In alcuni casi, usato solo come luogo dove storare informazioni persistenti
- **Local Data Source**: contiene il codice necessario per mantenere e ottenere lo stato dell'applicazione da uno storage locale (che può essere la sola sorgente di informazione o una cache per un server remoto)



Storare o prendere informazioni ha una logica complessa:

- Business logic fa riferimento ad un LocalDataSource che usa Room per accedere al db.
- Mentre RemoteDataSource usa Retrofit che costruisce in automatico il codice necessario per contattare il server remoto

Models:
deve essere disegnato in modo **indipendente da views** e altri componenti android

- Managing separately the UI code from the application logic makes it easier to maintain the app and enables testing

Il modo in cui gestiamo il modello deve avere una **rappresentazione neutrale**, guidata solo dai requisiti di dominio

- Si deve rimappare ogni cosa che torna dal data-layer in modo che sia compatibile con l'applicazione
  - Data coming from the remote server is usually denoted as DTOs - Data Transfer Objects
  - Data coming from the local database is denoted as Entities

Aggiornamenti **asincroni**
Se modifico il local Database, lo so. Se i dati arrivano da remoto, può succedere che qualcuno modifica il valore dopo che ho preso dal server → ho bisogno di capire cosa è successo:

- Accettare comunque
- Non accettare

Query sono asincrone → da quando faccio la richiesta a quando ho la risposta, può passare del tempo:

- È possibile che l'informazione che ho richiesto, arriva quando il richiedente non esiste più.

Per ridurre i problemi di asincronicità, tipicamente esponiamo i dati come LiveData o **Flow** o RxJava observables.

In ogni caso, posso avere problemi → non posso fare query di quelle informazioni direttmente dal main thread: → bisogna switchare thread:

- Usando flow: riesco a fare più facilmente
  - Permette di eseguire un'operazione su un altro thread

In alcune situazioni conviene **modellare il model:**

- Insieme di Classi che forniscono metodi
  - Rappresentare i dati in modo mutuale
  - Aggiornamenti
  - Filtering

Persistenza: specificata nelle interfacce, l'implementazione di questa onterfaccia, viene supportata da tools che rendono tutto più semplice.

Come mantenere dati persistenti in db locale

SQLite: libreria che espone un insieme di metodi per avere sql, tutto fornito dal sql language ma implementato ed eseguito come una libreria on demand, in ogni momento che io performo. Noi possiamo effettuare query su queste librerie.

```
dependencies {
    val room_version = "2.6.1"

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

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

Plugins {id("kotlin-kapt")}

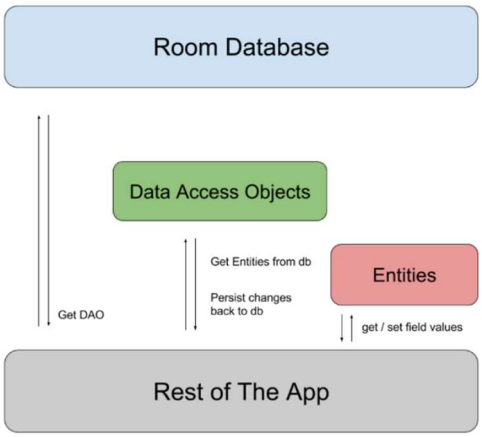
room-library: fornisce uno strato per far interagire un'applicazione logica su SQLite

Tipicamente vogliamo descrizione delle tabelle → per ogni record → classe.

***Nota**: Room funziona a compile time*

Room:

- **Db object**: responsabile per gestire la comunicazione tra la nostra applicazione e i dati storati
- File hanno una versione in quanto db possono evolvere



1 **room db**: con versione e seconda classe

- Il db espone un insieme funzioni che lavorano con Dao object
  - Se uso 3 tabelle, probabilmente 3 dao
  - Interagire con db
  - Possono rititrare booleani

Molti dati, per ogni dato, più entitiies

Componenti di una Room:

- **Database**: entry point per accedere tutti i dati
- **Dao**: uso i metodi in dao per Storare/eliminare/prendere entity
- **Entity**: classe/rappresentazione di tabella in db/ come oggetto: singola linea in db
  - Con chiave ID

```
@Entity(tableName = "users")
data class User (
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,

    @ColumnInfo(name = "first_name")
    val firstName: String,

    @ColumnInfo(name = "last_name")
    val lastName: String
```

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getAll() : LiveData<List<User>>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: List<Int>): LiveData<List<User>>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND "
           + "last_name LIKE :last LIMIT 1")
    fun findByName(first: User, last: String): User

    @Insert(onConflict = REPLACE)
    fun save(user: User)

    //@Update  //NOT NEEDED because of the REPLACE on the above method
    //fun update(user: User)

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

DATABASE CLASS

- Abstract class che **estende roomDatabase**, con label @Database
    - Deve avere un insieme di dati astratti
- Database deve essere istanziato a runTime: deve esserci massimo una istance per db:
    - Altrimenti si distrugge tutto
    - Ci sono molte tecniche per garantirlo
- mantiene versione

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

    companion object {
        @Volatile
        private var INSTANCE: UserDatabase? = null
        fun getDatabase(context: Context): UserDatabase =
            (INSTANCE ?: synchronized(this) {
                val i = INSTANCE ?: Room.databaseBuilder(context.applicationContext,
                                                         UserDatabase::class.java,
                                                         "user_database").build()

                INSTANCE = i
                INSTANCE
            })!!
        }
    }
}

val db =  UserDatabase.getDatabase(application)
```

Due tecniche:

- espondo un **metodo statico per prendere il db fornendo il Context**:
    - Si basa sulla *variabile statica ISTANCE*
        - Se ho già istance, ritorno il valore
        - Se non lo ho, la devo creare
    - Se istance null, devo buildare il db e ritornare istance
        - Questo garantisce che getDb può essere invocato da vari thread contemporaneamente
            - Solo uno lo builda grazie al synchronize
- sfrutto **application class** dato che android mi garantisce che questa sia unica
    - Chi vuole accedere al db, deve avere una istance valida dell'application class
    - Si può usare una lateinit var in cui salvare la istance (nel companion)
        - Chi vuole accedere al db, non deve accedere all'application istance ma gli basta fare customAppllication.db

come modellare → data class

```
@Entity(tableName = "users")
data class User (
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,

    @ColumnInfo(name = "first_name")
    val firstName: String,

    @ColumnInfo(name = "last_name")
    val lastName: String

)
```

i**d numerico** obbligatorio per sqlite, in questo caso auto_generato partendo da 0 *(conviene metterlo come ultimo nella classe in modo tale da poter evitare di passarla essendo di default)*
nota: nei db solitamente si usa **snake_rappresentation**

per ogni entity, serve un DAO: fornisce insieme di metodi:

- **Queery**
    - Facendo on**Conflict=replace**, posso evitare di fare update
    - Attenzione alle foreign key

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getAll() : LiveData<List<User>>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: List<Int>): LiveData<List<User>>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND "
           + "last_name LIKE :last LIMIT 1")
    fun findByName(first: User, last: String): User

    @Insert(onConflict = REPLACE)
    fun save(user: User)

    //@Update  //NOT NEEDED because of the REPLACE on the above method
    //fun update(user: User)

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

Per ogni parametor della nostra entità, viene creata una colonna:

- Per evitare che questo accada, posso mettere **@Ignore** alle colonne che non voglio creare

- Uniqueness can be specified with the property ***unique*** of a `@Index` annotation set to ***true***
- Relationships are expressed through the `@ForeignKey` annotation passed as a property to the `@Entity` one

```
@Entity(foreignKeys = @ForeignKey(entity = User.class,
                                  parentColumns = "id",
                                  childColumns = "user_id"))
```

  - SQLite handles `@Insert(onConflict = REPLACE)` as a set of `REMOVE` and `REPLACE` operations instead of single update one. This method could affect the foreign keys constraints
  - By setting `onDelete = CASCADE` in the `@ForeignKey` annotation you can tell SQLite to delete all related items, if the source record is removed

Nella mia tabella ho user_id, nlla user.class avrò id

Query possono:

- Tornare entità singole
- Option dellle entity
- Collezioni

➔ Possono essere incapsulati in Flows, LiveData → query diventa Observable

Se dichiaro i **metodi come SuspendFan**, questi possono essere attivati nel contesto delle coroutine che switchano al IO thread senza che mi debba occupare dello switch

Ci sono situzioni in cui la mia tabella piò cointenere tante cose → utile **paginare**: può esser fatto si in modo esplicito

- Limitando spazio
- Usando un oggetto cursore: invece che fetchare k'intero dataset, mi torna un pointer, iterator responsabile di qfare query su prossimo posto.
    - Ho la possibilità di chiedere per altro
    - Problema: non sopravvive al destroy di un'attività

MIGRATION:

- Quando definisco il db, devo definire le varie cose
- Prima di invocare il build, devo invocare migration
    - Codice responsibile per farmi spostare per le versione

› Room provides the migrations mechanism for this particular need

```
Room.databaseBuilder(applicationContext, MyDb::class.java, "database-name")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build()

val MIGRATION_1_2 = object : Migration(1, 2) {
       override fun migrate( database : SupportSQLiteDatabase) {
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, PRIMARY KEY(`id`))")
    }
};

val MIGRATION_2_3 = Migration(2, 3) {
    override fun migrate(database : SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE Book ADD COLUMN pub_year INTEGER")
    }
};
```

23- Accessing the network

Come connettersi ad un sevrer remoto. Si può usare un qulalsiasi remote cliente per connettersi ad un server remoto ma non è conveniente per questioni di complessità.

È meglio se ci si basa su librerie → retrofit. Bisogna aggiungere nel manifest la internet permission, che viene accettata di default `<uses-permission android:name="android.permission.INTERNET" />`

Libreria Retrofit:

```
dependencies {
    implementation 'com.google.code.gson:gson:2.10'
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
}
```

Prima di muovere le cose, biosgna definire quali tipi di dato bisogna muovere attraverso la nostra connessione http

- Si usa una **data class** contenente tanti campi quanti sono quelli che verranno scambiati
    - Si può usare **@SerualizedName** per passare dai nomi usati nel json a quelli che usiamo internamente

```
data class ToDo (
  val id: Int,

  @SerializedName("task-description")
  val taskDescription: String,

  val before: LocalDate
)
```

Dopo che si modellano i vari dati, si modella il **Service**: interfaccia contenente tanti metodi quanti quelli presenti nella remote url su cui si vogliono fare le query.

- Metodi devono essere annotati con i corrispondenti:
    - Method url
    - Relative url
- Metodi possono avere 0+ parametri in base a cosa si sta usando, i vari parametri possono avere label:
    - **@Body**: quando si vuole fornire un oggetto come body di una richiesta
        - Post/put/patch
    - **@Url:** quando si vuole usare una dynamic url
    - **@Query:** cosa c'è dopo il ? nel url
    - **@Field:** nel caso iun cui serva mandare il contenuto dei nostri parametri usando FormUrlEncoded
        - Server Processerà il body decodifiicando partendo dal contenuto

Valore di ritorno:

- Alcuni metodi non hanno un return value → **void**.
- Altri casi**, retrofit.Call<T>** → questo rappresente una interfaccia calback; quando la risposta sarà ricevuta, la callback interface, che accetta parametri di tipo T, sarà invocata.
- Se definiamo la nostra **funzione come suspend fun**, questa può ritornare direttamente i valori attesi e la funzione viene implementata attraverso coroutine.
  Se dichiariamo una funzione come return LiveData → funzione ritorna immediatamente fornendo un liveData che non contiene nulla e pian piano il LiveData sarà popolato in base alle risposte del server

```
interface APIInterface {

    @POST("/api/users")
    suspend fun createUser(@Body user: User): User

    @GET
    suspend fun getUsers(@Url url: String): List<User>

    @GET("/api/users?")
    suspend fun doGetUserList(@Query("page") page: String): List<User>

    @FormUrlEncoded
    @POST("/api/users?")
    suspend fun doCreateUserWithField( @Field("name")name:String,
                                       @Field("job") job: String
    ): User
}
```

esempio con suspend function:

- Metodi con cui interagisco con il server remoto
- Invocsti nel contesto delle coroutine che non devono essere sul main thread

Creare un servizio:

Implementazione generata automaticamente creando una istance dell'oggetto retrofit.
GsonBuilder → traduce dati

Build: accetta come input tutte le proprietà e mi restituisce un oggetto retrofit

L'oggetto retrofit mi fornisce il metodo create che accetta come input un nome dell'interfaccia e ritorna l'implementazione corrispondente.

- In questo caso accetta il nome dell'interfaccia
- Lo linko ad un indirizzo

```
object ApiInstance {

    private val gson = GsonBuilder().create()

    private val retrofitBuilder: Retrofit.Builder =
        Retrofit.Builder()
            .baseUrl("http://10.0.2.2:8080/")
            .addConverterFactory(GsonConverterFactory.create(gson)

    private val retrofit: Retrofit = retrofitBuilder.build()

    fun <T> createService(serviceClass: Class<T>): T {
        return retrofit.create(serviceClass)
    }

}
```

***Nota**: Se necessito di contattare 2+ server, devo creare 2+ retrofit object*

Supporto Flow<T>

Aggiungendo una libreria per supportare Flow, retrofit è in grado di ritornare un Flow di valori. Questo può esere conveniente perché mida la possibilità di ottenere una **versione streamer delle cose**:

- **Server side events** → ho un'applicazione dove ricevo una single query ma ritorna molti valori durante il tempo
    - Quando c'è una change nel server side → aggiornamento valori

The library `tech.thdev:flow-call-adapter-factory:1.0.0` defines class `FlowCallAdapterFactory` that is used to customize the `Retrofit.Builder` object

```
val retrofitBuilder = Retrofit.Builder()
    .baseUrl("http://10.0.2.2:8080/")
    .addConverterFactory(
        GsonConverterFactory().create(gson)
    )
    .addCallAdapterFactory(
        FlowCallAdapterFactory()
    )
```

```
interface MyDataService {

    @GET("/")
    fun listItems(): Flow< List< Items > >

    @GET("/data")
    fun getData(): Flow< MyData >
}
```

***Nota**: aggiornamenti avvengono in questo modo solo unidirezionalmente (da server ad app)*

Invocare un servizio:

Whoever is in need to contact the remote server may create an instance of the service interface via the singleton ApiInstance

- Or it can get directly an instance of it via dependency injection

Care has to be taken not to make invoke the service in the main thread

- As well as to properly handle possible exceptions and cancellation