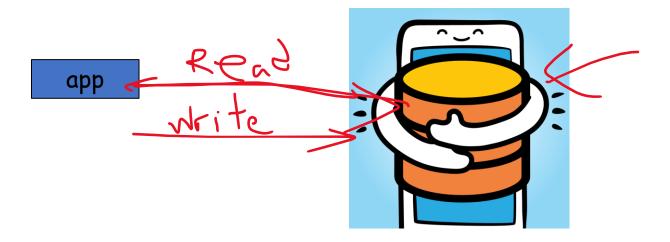
CMPS 312

Data Layer



Dr. Abdelkarim Erradi CSE@QU

Outline

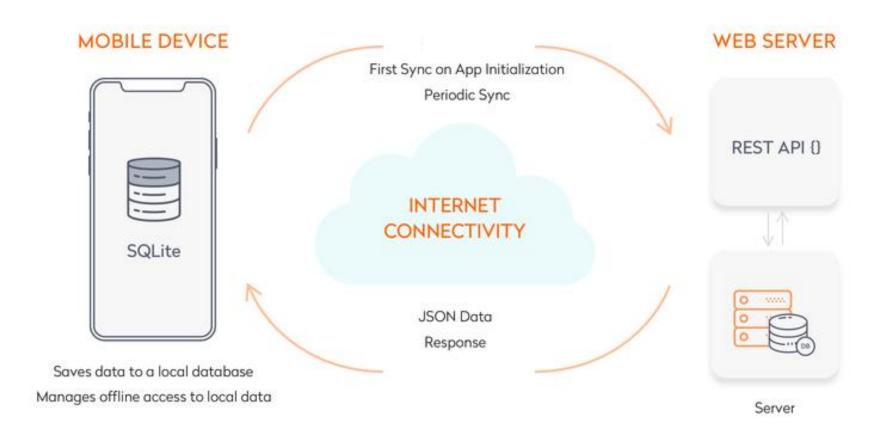
- Data persistence options for Mobile Apps
- 2. Floor package programming model
- 3. <u>Type Converters, Foreign keys & Database Views</u>
- 4. Observable Queries using Streams

Data persistence options for Mobile Apps





Offline app with Sync



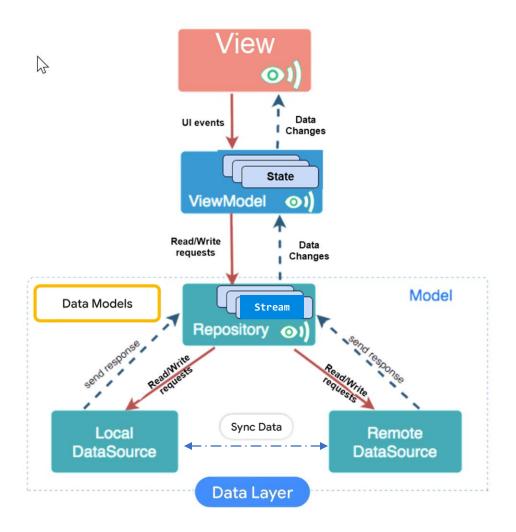
 Cache relevant pieces of data on the device. App continues to work offline when a network connection is not available.



 When the network connection is back, the app's repository syncs the data with the server.

MVVM Data Layer

- The Data Layer manages the app data:
 - implements read/write operations
 - notifies the ViewModel of data changes
- It includes data models, data sources and repositories:
 - Data models define the entities that holds the app data (i.e., in-memory representations of the data)
 - A data source is responsible for reading and writing data to a single source such as a database or a network service.
 - The repository exposes, updates, and synchronizes data using a local datasource and/or a remote datasource + implements datarelated logic



Data Storage Options on Android

Preferences DataStore

- Lightweight mechanism to store and retrieve key-value pairs
- Typically used to store application settings (e.g., app theme, language), store user details after login

Files

 Store unstructured data such as text, photos or videos, on the device or removable storage

• SQLite database

Store structured data (e.g., posts, events) in tables

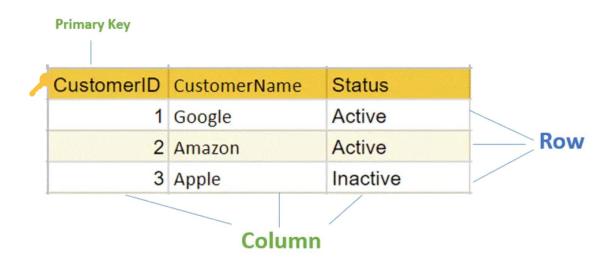
Cloud Data Stores

o e.g.,



Relational Database

- Database allows persisting structured data
- A relational database organizes data into tables
 - A table has rows and columns
 - Tables can have relationships between them
- Tables could be queries and altered using SQL



SQL Statements

- Structured Query Language (SQL)
 - Language used to define, query and alter database tables
 - SQL is a language for interacting with a relational database
- Creating data:

```
INSERT into User (firstName, lastName)
VALUES ("Ahmed", "Sayed")
```

Reading data:

```
SELECT * FROM User WHERE id = 2
```

Updating data:

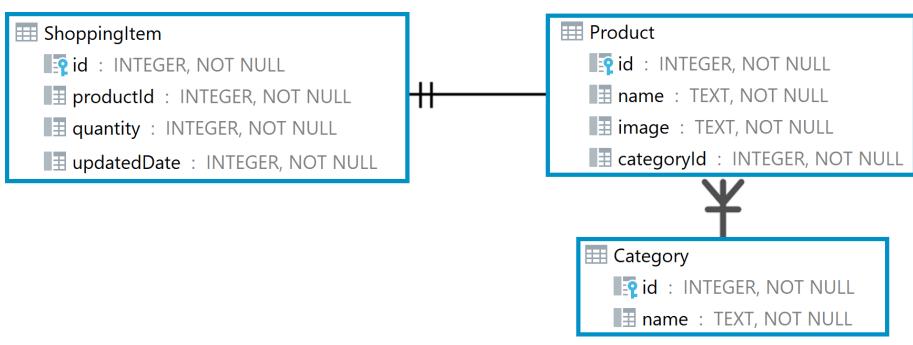
```
UPDATE User SET firstName = "Ali" where id = 2
```

Deleting data:

```
DELETE from User where id = 2
```

Database Schema of Shopping List App

- The Entity Relationship (<u>ER</u>) diagram of the Shopping List App database
 - A ShoppingItem has an associated Product
 - Product has a Category
 - Category has many products



Querying Multiple Tables with Joins

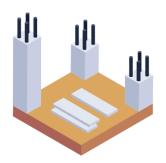
- Combine rows from multiple tables by matching common columns
 - For example, return rows that combine data from the Product and Category tables by matching the Product.categoryId foreign key to the Category.id primary key

select p.id, p.name, p.image, c.name category

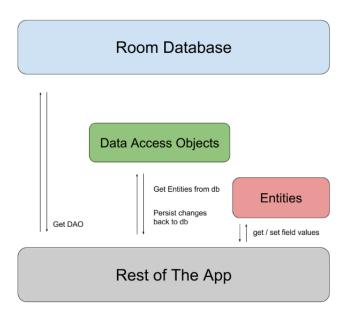
from Product p **join Category c on** p.categoryId = c.id

where p.categoryId = '1'

id	‡	name	\$ image	*	catego	ry (
1		Grapes	*		Fruits	4
2		Melon	•		Fruits	4
3		Watermelon	®		Fruits	-
4		Banana	2		Fruits	•
5		Pineapple	ě		Fruits	4
6		Mango			Fruits	•
7		Red Apple	\(\)		Fruits	1
8		Green Apple	ightharpoons		Fruits	
9		Pear	•		Fruits	1
10	,	Peach	 Ď	- man	Fruits	3



Floor package programming model







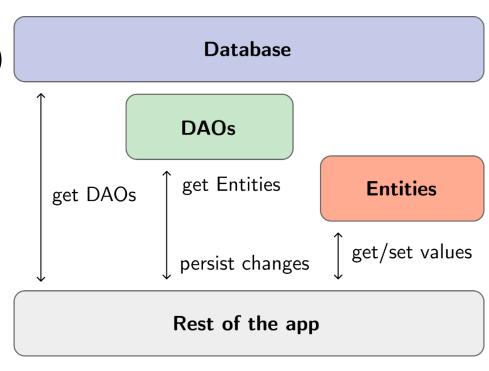
Floor Library

- The Floor persistence library provides an abstraction layer over SQLite to ease data management
 - Define the database, its tables and data operations using annotations
 - Floor automatically translates these annotations into SQLite instructions/queries to be executed by the DB engine
 - Enables automatic mapping between in-memory objects and database rows while still offering full control of the database with the use of SQL
- Dependencies and documentation:
 - https://pinchbv.github.io/floor/

Floor architecture diagram

Working with Floor

- 3 major components in Floor
- Model DB Tables as regular Entity
 Classes
- Create Data Access Objects (DAOs)
 - DAO abstract class methods are annotated with queries for Select, Insert, Update and Delete
 - DOA implementation is autogenerated by the complier
 - DOA is used to interact with the database
- FloorDatabase → holds a connection to the SQLite DB and all the operations are executed through it



Floor main components

- Entity → each entity class is mapped to a DB table
 - Dart class annotated with @entity to map it to a DB table
 - Must specify one of the entity properties as a primary key
 - Table name = Entity name & Column names = Property names (but can be changed by annotations)
- Data Access Object (DAO) → has methods to read/write entities
 - Contains CRUD methods defining operations to be done on data
 - Interface or abstract class marked as @dao
 - One or many DAOs per database
- Database → where data is persisted
 - Abstract class that extends FloorDatabase and annotated with @Database

Entity

- Entity represents a database table, and each entity instance corresponds to a row in that table
 - Class properties are mapped to table columns
 - Each entity object has a Primary Key that uniquely identifies the entity object in memory and in the DB
 - The primary key values can be assigned by the database by specifying autoGenerate = true

Customizing Entity Annotations



- By default, the name of the entity class is the same as the associated table and the name of table columns are the same as the class properties
 - In most cases, the defaults are sufficient but can be customized
 - Use @entity (tableName = "...") to set the name of the table
 - The columns can be customized using @ColumnInfo(name = "column_name") annotation
- If an entity has properties that you don't want to persist, you can annotate them using @lgnore

DAO @Query

- @Query used to annotate query methods
- Floor ensures compile time verification of SQL queries

```
@dao
abstract class UserDao {
    @Query("select * from User limit 1")
    Future<User> getFirstUser();
    @Query("select * from User")
    Future<List<User>> getAll();
    @Query("select firstName from User")
    Future<List<String>> getFirstNames()
    @Query("select * from User where firstName = :fn")
    Future<List<User>> getUsers(fn: String);
    @Query("delete from User where lastName = :ln")
    int deleteUsers(ln: String)
```

DAO @insert, @update, @delete

- Used to annotate insert, update and delete methods
- DAO methods are async functions to ensure that DB operations are not done on the main UI thread

```
@dao
abstract class UserDao {
    @insert
    Future<Long> insert(user: User);
    @insert
    Future< List<Long>> insertList(users: List<User>);
    @delete
    Future<void> delete(user: User);
    @delete
    Future<void> deleteList(users: List<User>);
    @update
    Future<void> update(user: User);
    @update
    Future<void> updateList(users: List<User>);
```

Floor database class

- Abstract class that extends FloorDatabase and Annotated with @Database
- Serves as the main access point to get DAOs to interact with DB

```
@Database(version: 1, entities: [Item, User])
abstract class ShoppingDB extends FloorDatabase()
UserDao get userDao;
}
```

Type Converters, Embedded object, Foreign keys & Map Queries





TypeConverter

- SQLite only support basic data types, no support for data types such as Date, enum, BigDecimal etc. Need to add a TypeConverter for such data types
- Convert an entity property datatype to a type that can be written to the associated table column and vice versa

```
class DateTimeConverter extends TypeConverter<DateTime, int> {
    @override
    DateTime decode(int databaseValue) {
       return DateTime.fromMillisecondsSinceEpoch(databaseValue);
    }
    @override
    int encode(DateTime value) {
       return value.millisecondsSinceEpoch;
    }
}

@TypeConverters([DateTimeConverter])
abstract class ShoppingDB extends FloorDatabase() { ... }
```

Nested @Embedded object

- Nested object annotated with @Embedded -> Floor flattens out the embedded object and maps its properties to columns in the DB table
 - User table will have a buildingNo, street and city columns

```
data class Address(val buildingNo: String,
                     val street: String,
                     val city: String)
@entity

Ⅲ User
data class User (
                                               id: Integer, not null
    @PrimaryKey(autoGenerate = true)
                                               firstName: TEXT, NOT NULL
    var id: Long,
                                               ■ lastName : TEXT, NOT NULL
    val firstName: String,
                                               buildingNo: TEXT, NOT NULL
    val lastName: String,
                                               street: TEXT, NOT NULL
    @Embedded val address: Address
                                               tity: TEXT, NOT NULL
```

Enforce integrity checks with foreign keys

- Foreign key allows integrity checks (e.g., can insert pet only for a valid owner) & cascading deletes
 - onDelete = ForeignKey.CASCADE when owner is deleted then auto-delete associated pets

```
@entity(foreignKeys = [
        ForeignKey(entity = Owner::class,
                parentColumns = ["id"],
                childColumns = ["ownerId"],
                onDelete = ForeignKey.CASCADE)
        ],
    // Create an index on the ownerId column to speed-up query execution
    indices = [Index(value = ["ownerId"])])
data class Pet(@PrimaryKey val id: Long,
               val name: String, val ownerId: Long)
@entity
data class Owner(@PrimaryKey val id: Long, val name: String)
```

Query Database Views

For each category return a list of associated products

```
@Query("""
    Select c.*, p.*
    From Category c join Product p on c.id = p.categoryId
    """)
suspend getCategoriesAndProducts(): Map<Category, List<Product>>
```

- For each category return the count of associated products
- @MapColumn is required when the key or value column
 of the map are from a single column

```
@Query("""
    Select c.name categoryName, count(p.id) as productCount
    From Category c join Product p on c.id = p.categoryId
    Group by c.id
    """)
suspend getCategoriesAndProductCounts(): Map<@MapColumn(columnName =
    "categoryName") String,
    @MapColumn(columnName = "productCount") Long>
```

Observable Queries using Stream







Observable Queries

- Observable queries allow automatic notifications when data changes
 - Notifies the app with of any data updates
- We can accomplish this by simply wrapping the return type of our DAO methods with Stream
 - active observers (i.e., UI) get notified when data change

```
// App will be notified of any changes of the Item table data
// Whenever Floor detects Item table data change, the new list of
Items will be provided to the app
@Query("Select * from Item")
Stream<List<Item>> observeItems();
```

Summary

Major Components

- @entity Defines table structure
- @dao An abstract class with functions to read/write from the database
- @Database Serves as the main access point to get DAOs to interact with DB



Resources

- Save data in a local database using Floor
 - https://developer.android.com/training/data-storage/Floor
- Floor codelabs
 - https://developer.android.com/codelabs/building-a-datalayer
 - https://developer.android.com/codelabs/Dart-androidtraining-Floor-database
- Working with Preferences DataStore
 - https://codelabs.developers.google.com/codelabs/androidpreferences-datastore