CMPS 312

Data Layer

Dr. Abdelkarim Erradi

CSE@QU

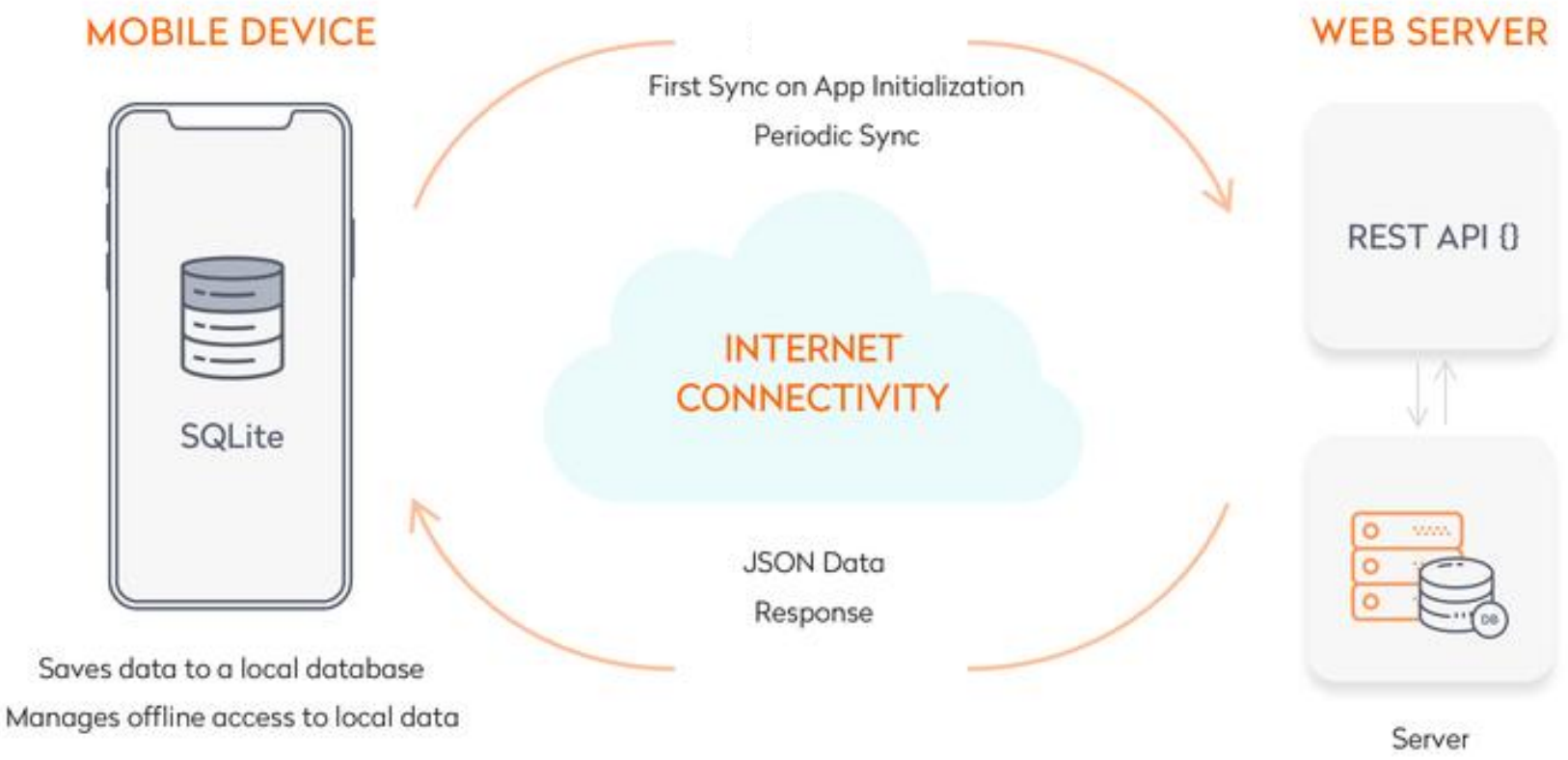
Outline

1. Data persistence options
2. Floor package programming model
3. One-to-many relationships, Views and Type Converters
4. Observable Queries using Streams
5. AsyncNotifierProvider

Data persistence options for Mobile Apps

Back

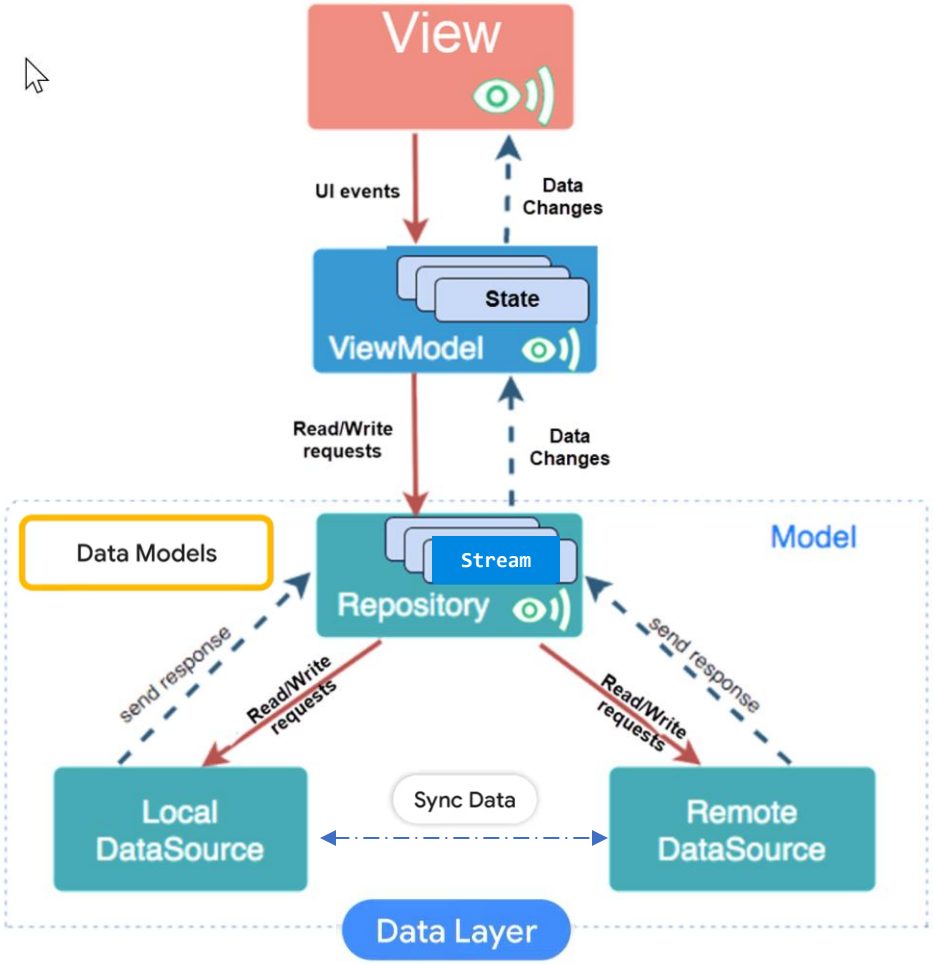
Offline Functionality with Data Sync



- **Local Caching**: Store essential data on the device for App to remain fully functional **offline** without network connectivity + **Improved performance**
- **Automatic Sync**: When the connection is restored, the app's repository **synchronizes** local data changes with the server.

MVVM Data Layer

- **Role of Data Layer**:
  - Handles read/write operations
  - Notifies ViewModel of data changes
- **Core Components**:
  - Data Models: Represent app entities in memory
  - Repository:
    - Exposes and updates data
    - Synchronizes local and remote data sources
    - Implements data-related logic
  - Data Sources: Local DB, cloud DB, or remote API



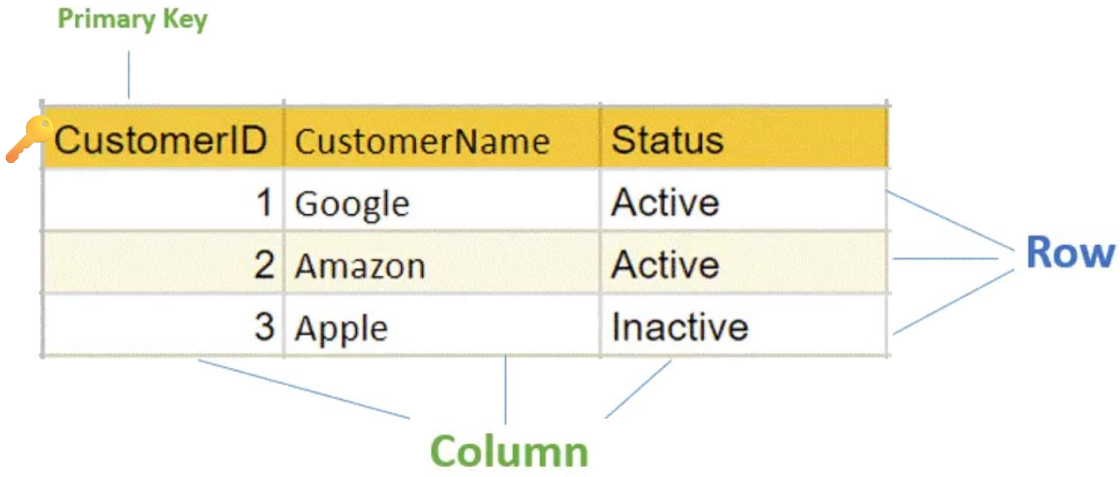
Data Storage Options for Mobile Apps

- **Key-Value Store**
  - 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
- 

database
  - Store structured data (e.g., posts, events) in tables
- **Cloud Data Stores**
  - 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
- **C**reating data:

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

- **R**eading data:

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

- **U**pdating data:

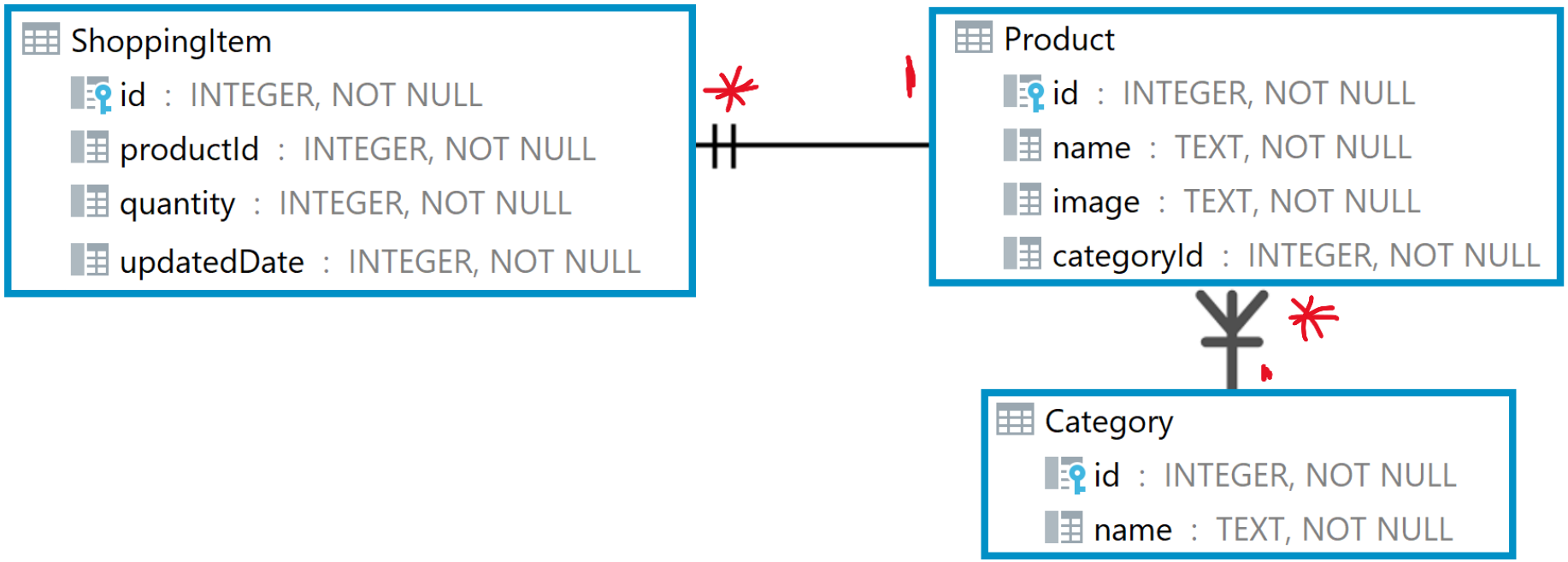
```
UPDATE User SET firstName = "Ali" where id = 2
```

- **D**eleting data:

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

Database Schema of Shopping List App

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

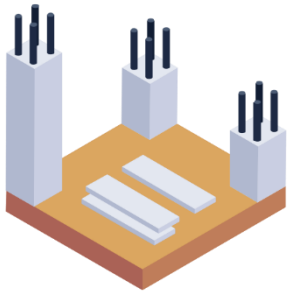


Querying Multiple Tables with Joins

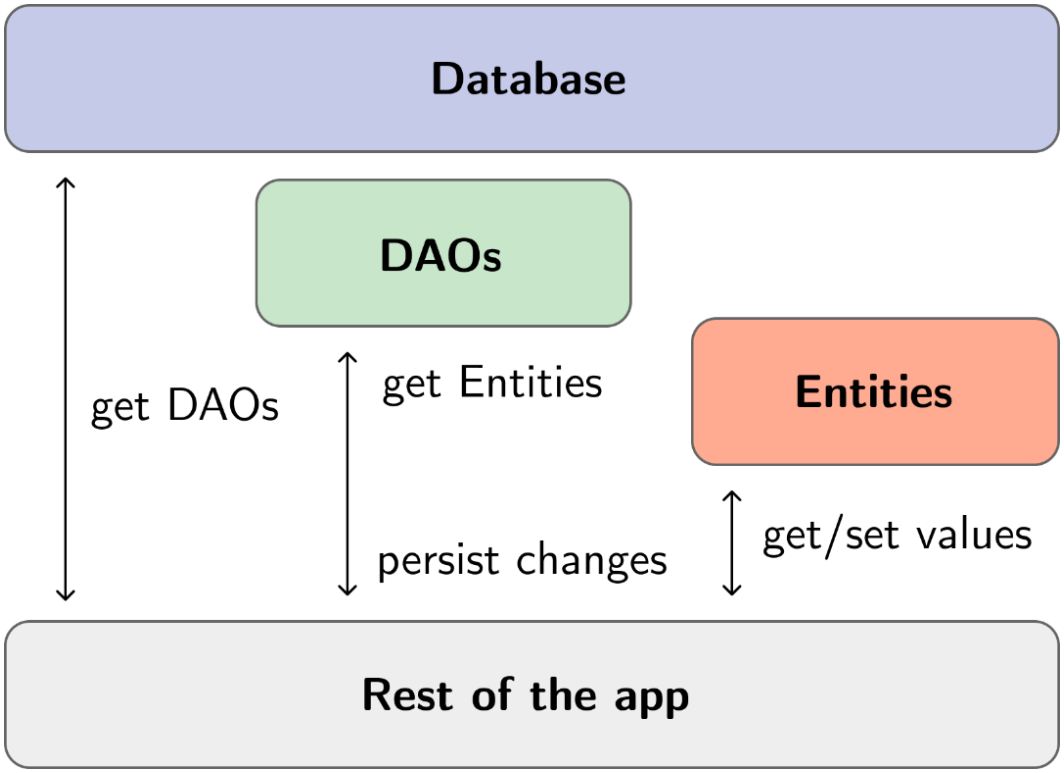
- Combine rows from multiple tables by matching common columns
  - For example, return rows that combine data from the **Product** and **Category** tables by matching the **Product.categoryId** foreign key to the **Category.id** primary key

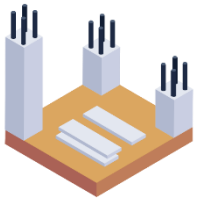
```
select p.id, p.name, p.image, c.name category
    from Product p join Category c on p.categoryId = c.id
    where p.categoryId = '1'
```

id	name	image	category
1	Grapes		Fruits
2	Melon		Fruits
3	Watermelon		Fruits
4	Banana		Fruits
5	Pineapple		Fruits
6	Mango		Fruits
7	Red Apple		Fruits
8	Green Apple		Fruits
9	Pear		Fruits
10	Peach		Fruits



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

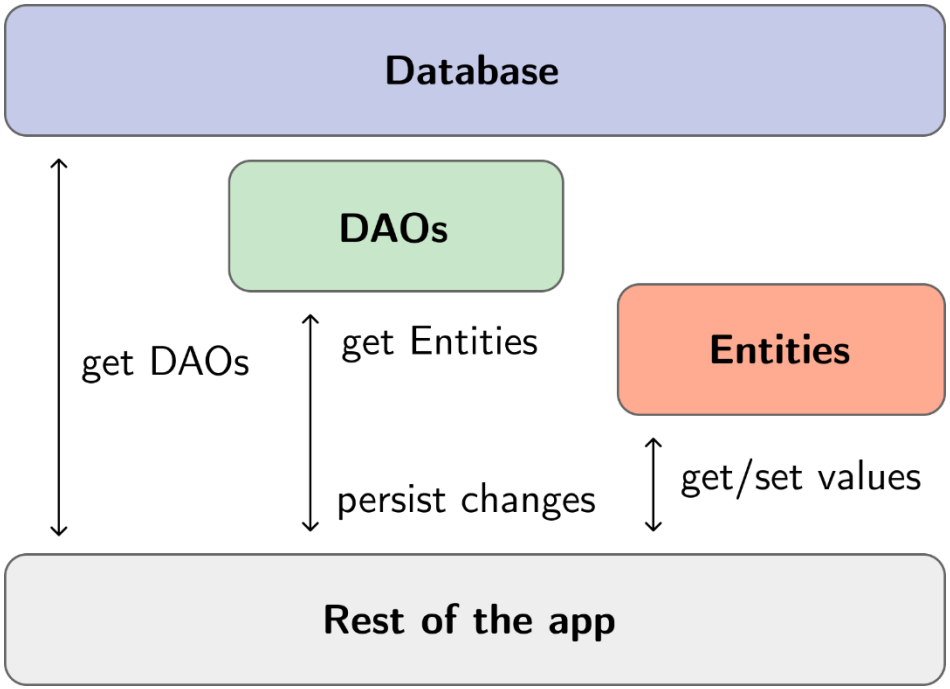
1. Model DB Tables as regular **Entity Classes**
2. Create Data Access Objects (DAOs)

   - **DAO abstract class** methods are annotated with queries for Select, Insert, Update and Delete

   - DOA implementation is auto-generated by the complier

   - DOA is used to interact with the database
3. **FloorDatabase** → holds a connection to the SQLite DB and all the operations are executed through it

3 major components in Floor



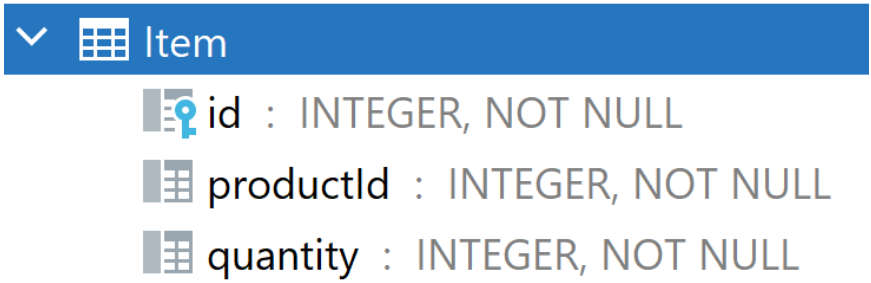
Floor main components

- **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`**

```
@Entity
class Item {
    @PrimaryKey(autoGenerate = true)
    long id;
    long productid;
    int quantity;
}
```



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 **@ignore**

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")
    Stream<List<User>> getUsers();
    @Query("select firstName from User")
    Future<List<String>> getFirstNames();
    @Query("select * from User where firstName = :fn")
    Future<List<User>> getUsersByFn(String fn);
    @Query("delete from User where lastName = :ln")
    Future<int> deleteUsers(String ln);
}
```

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 isolate

```
@dao
abstract class UserDao {
    @insert
    Future<int> add(User user);
    @insert
    Future< List<int>> addList(List<User> users);

    @delete
    Future<void> delete(user: User);
    @delete
    Future<void> deleteList(List<User> users);

    @update
    Future<void> update(user: User);
    @update
    Future<void> updateList(List<User> users);
}
```

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

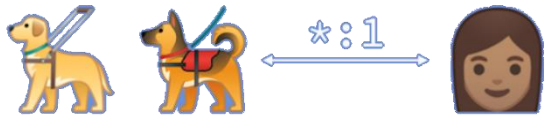
```
@Database(version: 1, entities: [Item, User])
abstract class ShoppingDB extends FloorDatabase() {
    UserDao get userDao;
}
```

- Then run this command to generate the database and DAO implementations

```
dart run build_runner build --delete-conflicting-outputs
```

One-to-many relationships
Views
Type Converters

One-to-many relationships



- Define one-to-many relationship between entities by defining a **foreign key** through the column references and performing joins in queries
  - Foreign key allows **integrity checks** (e.g., can insert pet only for a valid owner) & **cascading** deletes

```
@Entity
class Owner {
  @PrimaryKey(autoGenerate: true)
  final int id;
  final String name;
  Owner({required this.id, required this.name});
}
@Entity(
  foreignKeys: [
    ForeignKey(
      childColumns: ['ownerId'], // Column in this entity
      parentColumns: ['id'],     // Column in the one-side entity
      entity: Owner,
      onDelete: ForeignKeyAction.cascade,
    ),
  ],
  indices: [
    Index(value: ['ownerId']),
  ],
)
class Pet {
  @PrimaryKey(autoGenerate: true)
  final int id; // Primary key
  final String name;
  final int ownerId; // Foreign key linking to the Owner table
  Pet({required this.id, required this.name, required this.ownerId});
}
```

The **foreignKeys** parameter in the **@Entity** annotation specifies the one-side table (Owner), the linking columns

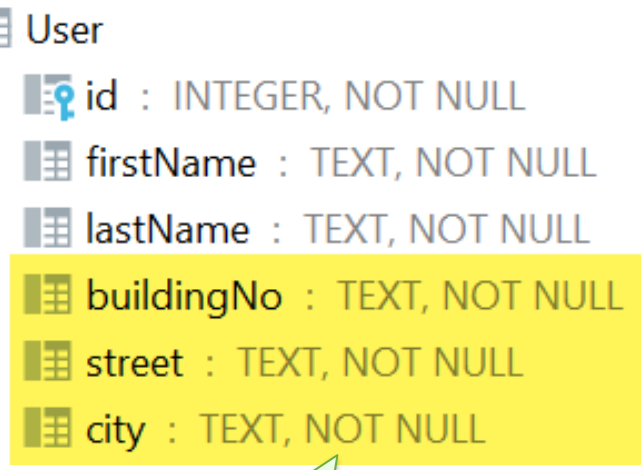
When an owner is deleted then auto-delete associated pets

An **index** is created on the ownerId column to improve query performance

Flattening Nested Objects with @Embedded

- **`@Embedded`** flattens fields of a nested object into the parent entity's table columns
  - No separate table for the nested object => simple design + no need for joins for small related data

```
// Embedded class - fields will be flattened into parent table
class Address {
  final String bulidingNo;
  final String street;
  final String city;
  Address({ ... });
}
@Entity()
class User {
  @PrimaryKey(autoGenerate: true)
  final int? id;
  final String name;
  @Embedded() // Flattens Address fields into users table
  final Address address;
  User({ ... });
}
```



User table will have a `buildingNo`, `street` and `city` columns

Views

- A database view is a virtual table created by defining a SQL query: a cleaner and reusable way to structure complex queries
  - This eliminates the need to write complex SQL queries repeatedly

```
@DatabaseView(
  'SELECT p.name as petName, o.name as ownerName FROM Pet p INNER JOIN Owner o ON p.ownerId = o.id',
   viewName: 'PetWithOwnerView',
)
class PetOwner {
  final String petName;
  final String ownerName;
  PetOwner({required this.petName, required this.ownerName});
}
```

- DAO for querying the View

```
@dao
abstract class PetDao {
  @Query('SELECT * FROM PetWithOwnerView')
  Future<List<PetOwner>> getPetsWithOwners();
}
```

- Add the class returned by the view to the @Database annotation as a view

```
@Database(version: 1, entities: [Owner, Pet], views: [PetOwner])
abstract class PetDatabase extends FloorDatabase { ... }
```

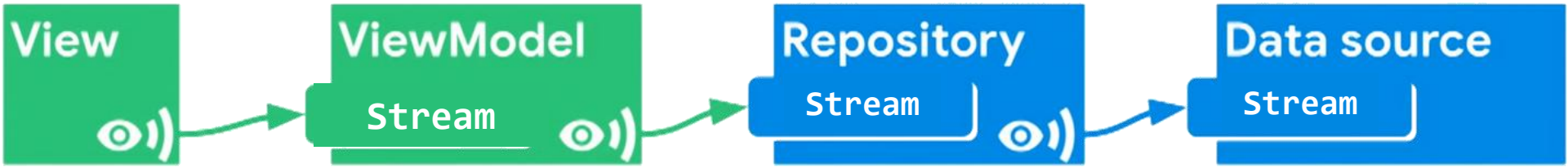
TypeConverter

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

```
class TodoTypeConverter extends TypeConverter<TodoType, String> {
  @override
  TodoType decode(String databaseValue) {
    return TodoType.values.firstWhere((e) => e.name == databaseValue);
  }
  @override
  String encode(TodoType value) {
    return value.name;
  }
}
```

```
class Todo {  ...
  @TypeConverters([TodoTypeConverter])
  final TodoType type;
```

Observable Queries using Stream

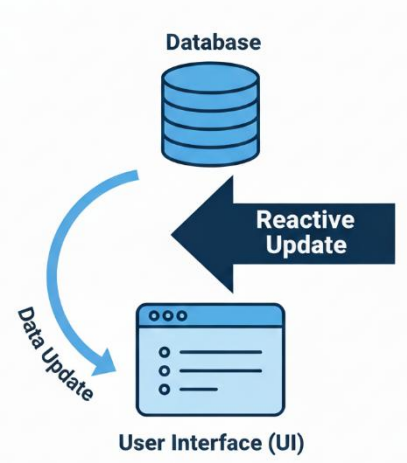


Observable Queries with Streams

- Observable queries automatically notify the app when data changes in the database
- Implemented by returning **Stream** from DAO methods.
- UI observes the stream and updates reactively.

✅ Android/iOS SQLite implementations support change notifications

⚠️ Windows/Desktop doesn't fully support automatic DB change detection. Streams don't emit new values when data changes




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

AsyncNotifierProvider

AsyncNotifierProvider

AsyncNotifierProvider combines async operations with mutable state, allowing **custom methods to refresh, update, or mutate async data**

💡 **When to Use**

✅ **CRUD operations** with async data (calls DAO to read/write to DB)

✅ **Refreshable** remote data

🌍 **Real-World Scenarios**

- Shopping cart (add/remove with server sync)
- User profile with update capabilities
- Todo list synced with backend

🆚 **vs FutureProvider**

Feature	FutureProvider	AsyncNotifierProvider
Custom Methods	❌ No	✅ Yes
Mutations	❌ No	✅ Yes
When to Use	Read-only	Need mutations

💻 Example: Shopping Cart with Server Sync

```
class ShoppingCartNotifier extends AsyncNotifier<List<CartItem>> {
 @override
 Future<List<CartItem>> build() async { // Load cart from server
    return await CartRepository().fetchCart();
  }
 Future<void> addItem(Product product) async { // Optimistic update
    final currentCart = state.valueOrNull ?? [];
    state = AsyncValue.data([...currentCart, CartItem.fromProduct(product)]);
    await CartRepository().addToCart(product.id);
  }
 Future<void> removeItem(String itemId) async {
    final currentCart = state.valueOrNull ?? [];
    state = AsyncValue.data(currentCart.where((item) => item.id != itemId).toList());
    await CartRepository().removeFromCart(itemId);
  }
 Future<void> refresh() async {
    state = const AsyncValue.loading();
    state = await AsyncValue.guard(() => CartRepository().fetchCart());
  }
}

final shoppingCartProvider = AsyncNotifierProvider<ShoppingCartNotifier, List<CartItem>>(
  () => ShoppingCartNotifier(),
);
```

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://pinchbv.github.io/floor/
- Persist data with SQLite
  - https://docs.flutter.dev/cookbook/persistence/sqlite