CMPS 312

Data Layer

Dr. Abdelkarim Erradi

CSE@QU

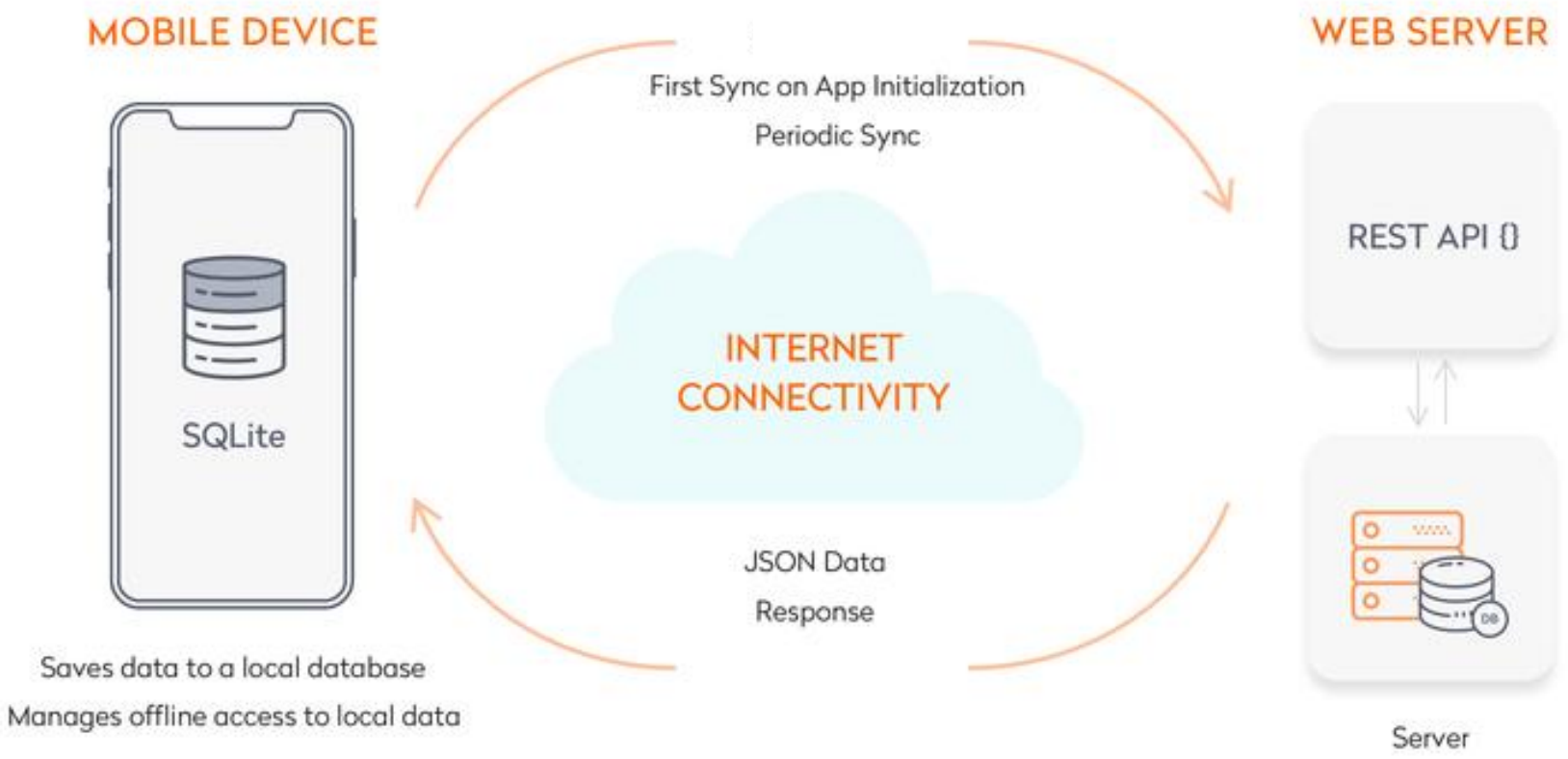
Outline

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

Data persistence options for Mobile Apps

Back

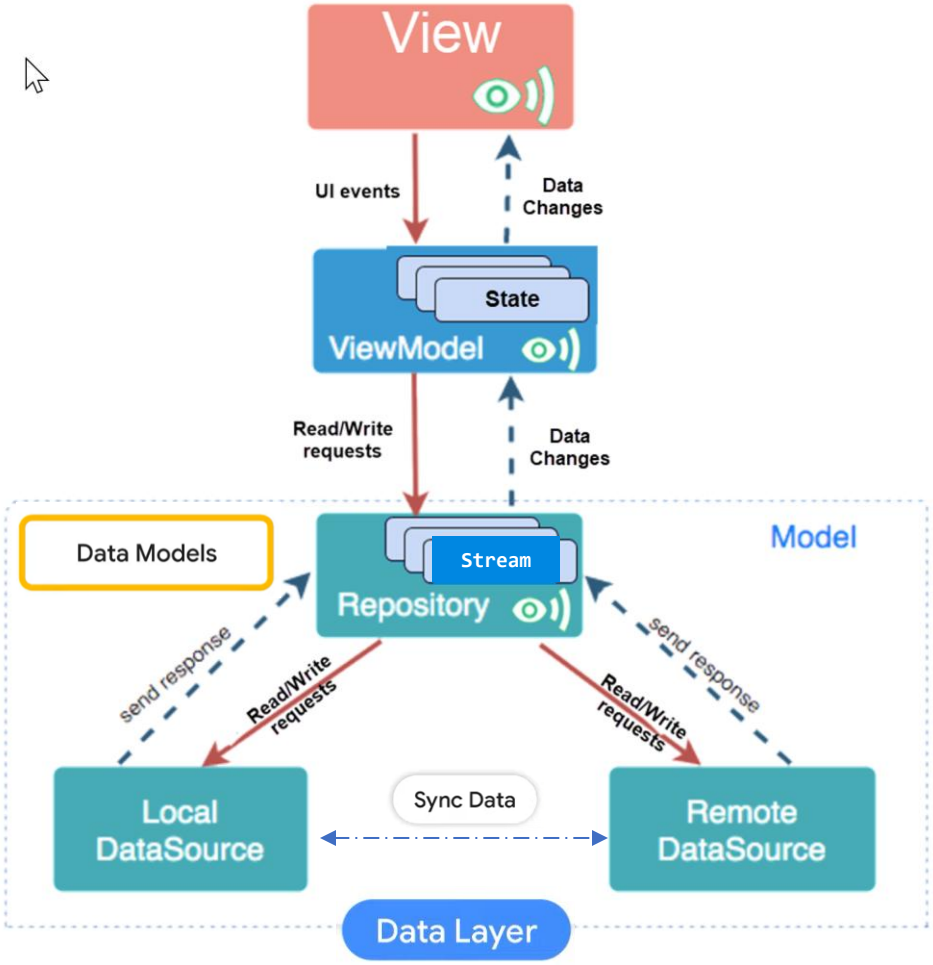
Offline Functionality with Data Sync



- **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



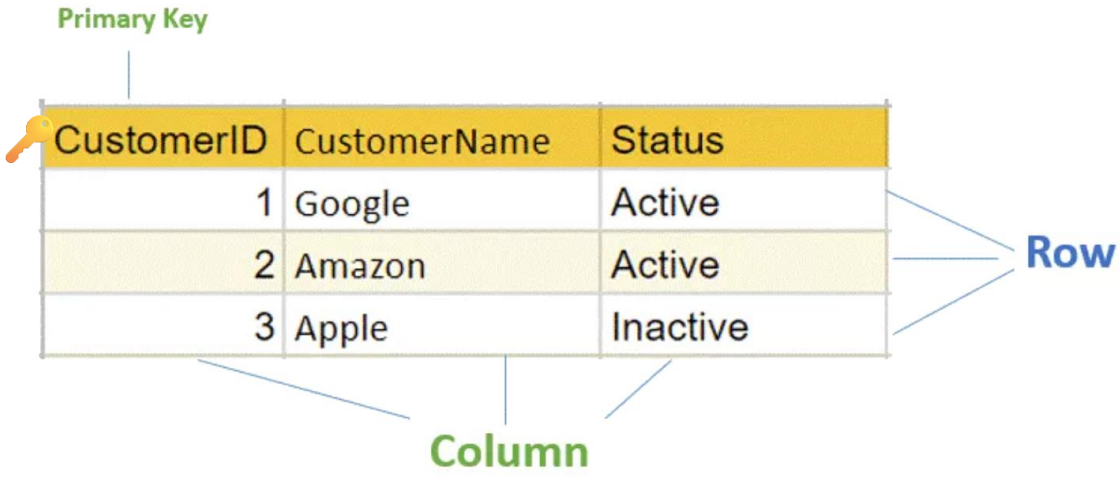
Data Storage Options for Mobile Apps

- **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:

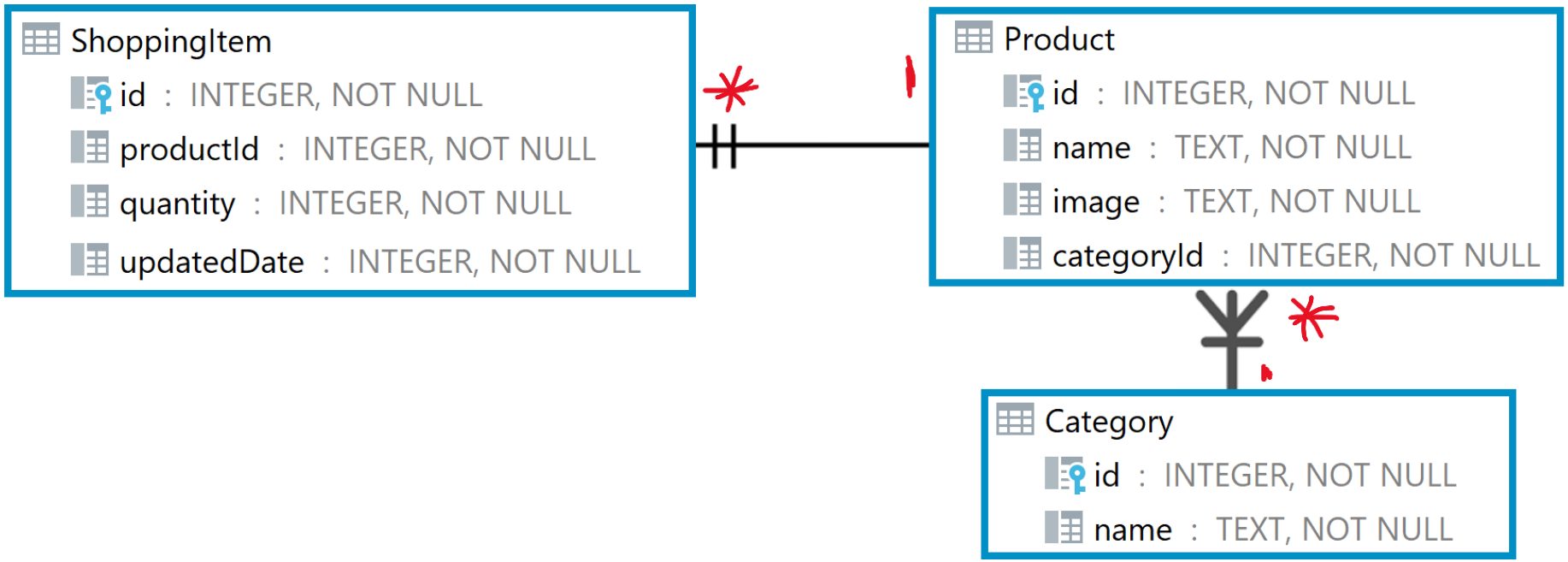
```
UPDATE User SET firstName = "Ali" where id = 2
```

- **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

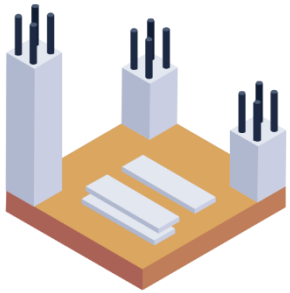


Querying Multiple Tables with Joins

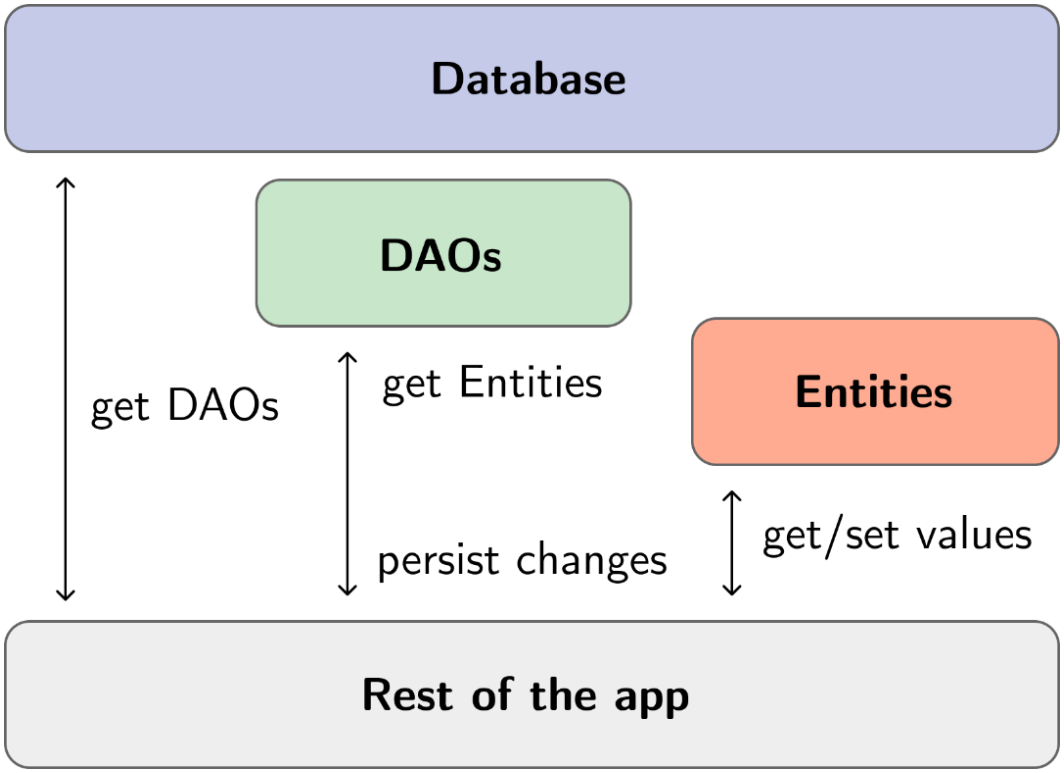
- Combine rows from multiple tables by matching common columns
  - For example, return rows that combine data from the **Product** and **Category** tables by matching the **Product.categoryId** foreign key to the **Category.id** primary key

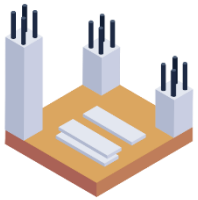
```
select p.id, p.name, p.image, c.name category
    from Product p join Category c on p.categoryId = c.id
    where p.categoryId = '1'
```

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

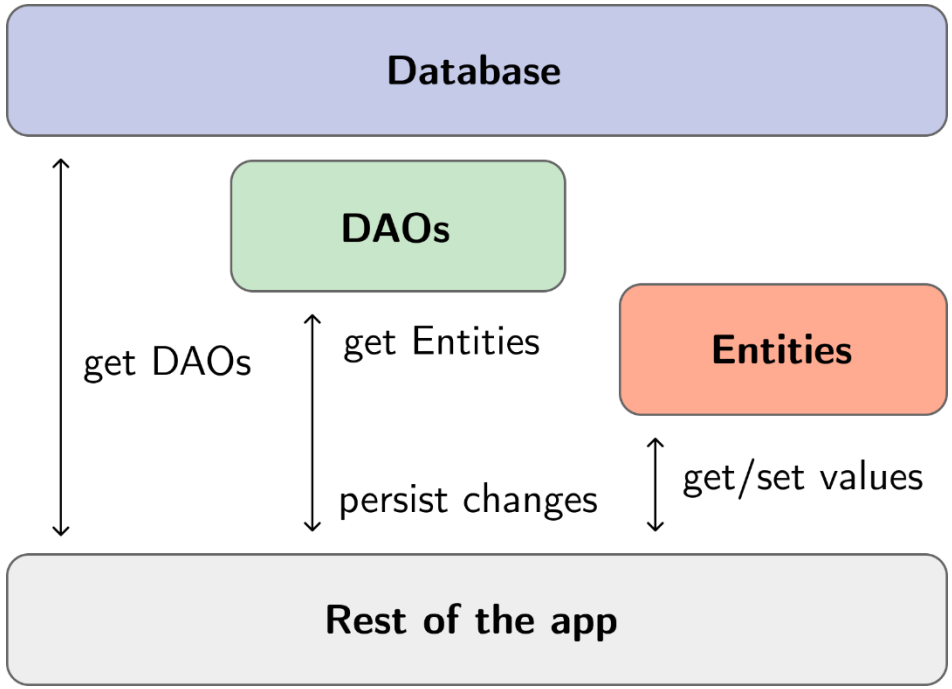
1. Model DB Tables as regular **Entity Classes**
2. Create Data Access Objects (DAOs)

   - **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



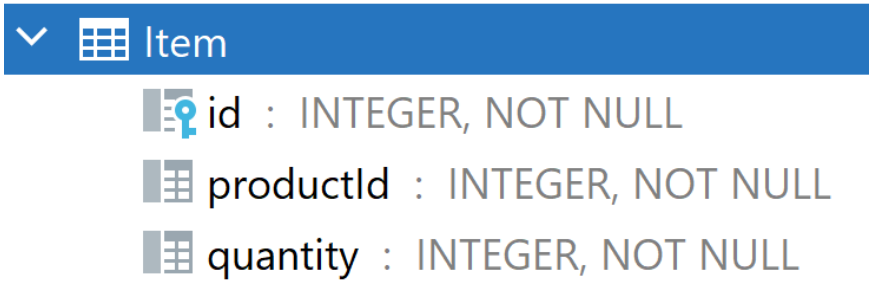
Floor main components

- **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

- Establish a one-to-many relationship by adding a foreign key to the many-side entity
- Foreign keys enforce data integrity (e.g., a pet can only belong to an existing owner)
- Support *cascading deletes* so related child records are automatically removed when the parent is deleted

```
@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

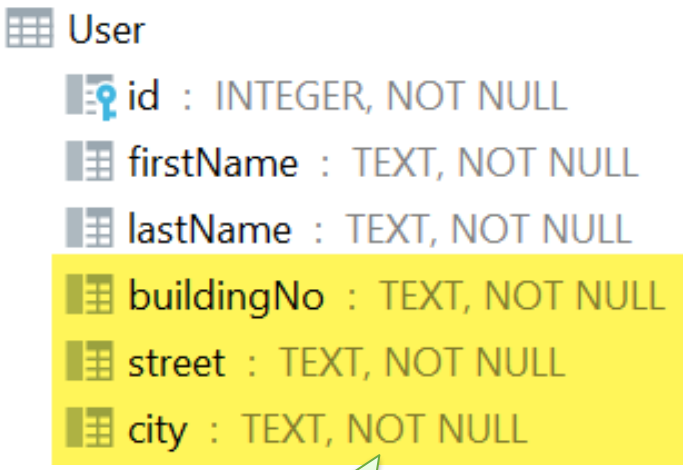
When an owner is deleted then auto-delete associated pets

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 { ... }
```

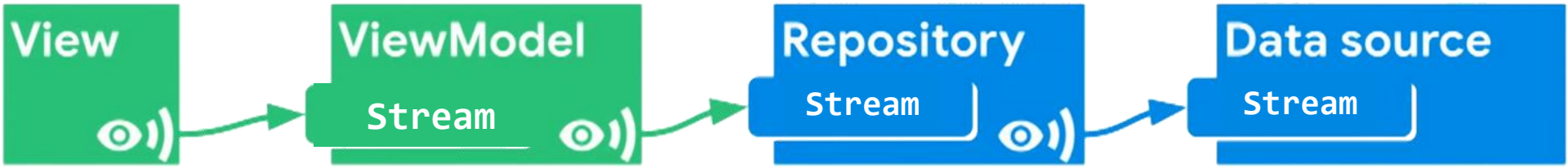
TypeConverter

- 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

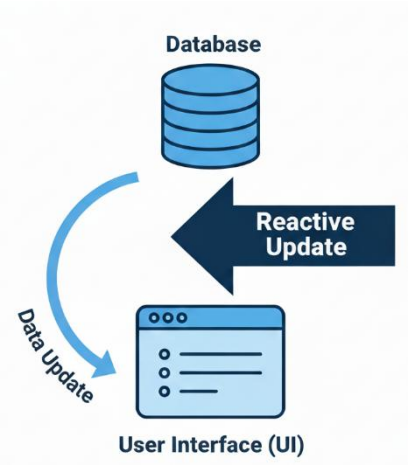


Observable Queries with Streams

- 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();
```

Data Flow Between App Layers

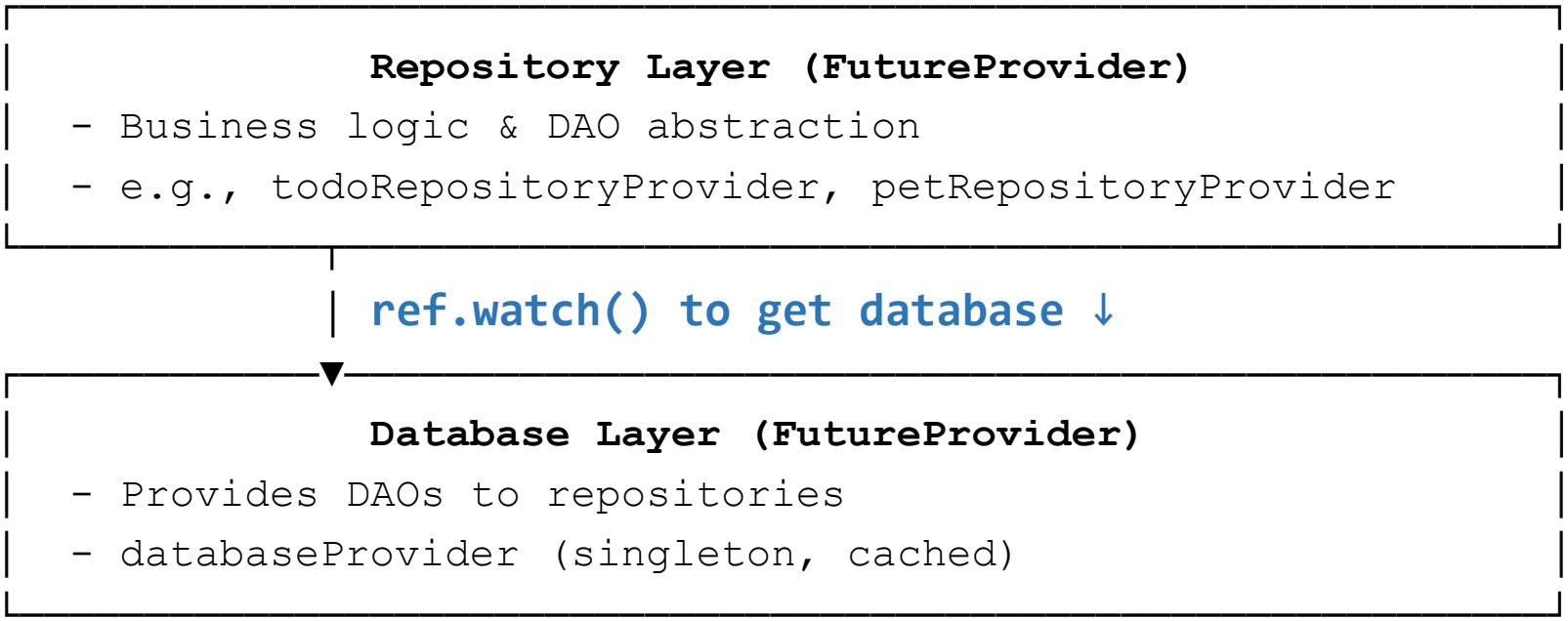
Back

Reactive Data Flow in MVVM Architecture

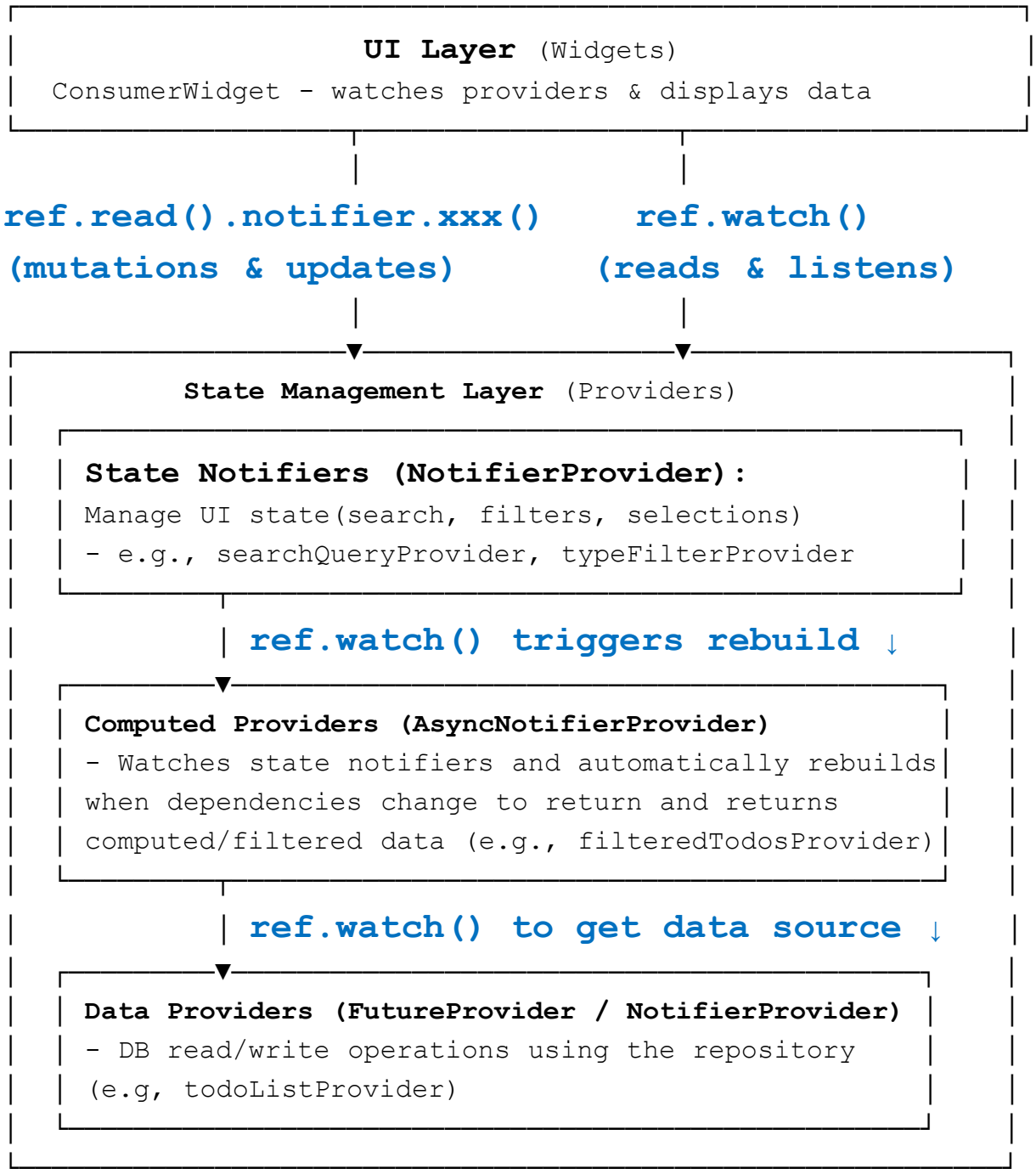
- Riverpod enables **state management** in a layered architecture:
  Database → Repository → Providers → UI
- **Key Idea:** Providers communicate through **dependency watching** (`ref.watch`) to create a **reactive data flow**
- **Benefits:**
  - Clear separation of concerns
  - Automatic UI updates when data changes
  - Scalable and testable architecture

Repositories and Database as Providers

- Repositories and the database can be exposed as Riverpod providers instead of being instantiated manually.
  - **Database Provider**: Creates and caches the database instance (e.g., SQLite via Floor)
  - **Repository Provider**: Depends on the database provider and exposes business logic and DAO access
  - This makes them part of the reactive architecture, so other providers can watch them



Communication Flow



Reactive Data Flow – Search ToDos (1 of 2)

Flow:
searchQueryProvider state changes

↓

ref.watch() detects change

↓

filteredTodosProvider.build() called automatically

↓

New database query executed

↓

AsyncValue<List<Todo>> updated

↓

UI displays new data

1 User Interaction → State Update

```dart
// UI Layer: User types in search bar
SearchBar(
  onChanged: (value) =>
    ref.read(searchQueryProvider.notifier).setQuery(value),
    //    └─ ref.read() gets the notifier
    //       └─ .notifier accesses the class methods
    //          └─ .setQuery() updates internal state
)
```

Flow:
User Input → ref.read(provider.notifier).method() → State Updated

2 State Change → Automatic Rebuild

```dart
class FilteredTodosNotifier extends AsyncNotifier<List<Todo>> {
  @override
  Future<List<Todo>> build() async {
    // 👀 Watching state - rebuilds when searchQueryProvider changes
    final searchQuery = ref.watch(searchQueryProvider);

    // Get repository
    final repository = await ref.watch(todoRepositoryProvider.future);

    // Query database with current filters
    return await repository.searchTodos(
      searchQuery: searchQuery,
    );
  }
}
```

Reactive Data Flow – Search ToDos (2 of 2)

Flow:
filteredTodosProvider state changes
↓
Widget rebuilds automatically (ref.watch detected change)
↓
.when() handles AsyncValue states
↓
UI displays new data

3 Data Update → UI Rebuild

```
// UI watches computed provider
Widget build(BuildContext context, WidgetRef ref) {
  final todosAsync = ref.watch(filteredTodosProvider);
  //                           └─ listens for changes

  return todosAsync.when(
    loading: () => CircularProgressIndicator(),
    error: (e, st) => Text('Error: $e'),
    data: (todos) => ListView.builder(
      itemCount: todos.length,
      itemBuilder: (_, i) => TodoTile(todos[i]),
    ),
  );
}
```

Reactive Data Flow with Manual Refresh Cycle after Mutation

Flow:

User Action (Delete)

↓

ref.read().notifier.delete()

↓

Database mutation (DELETE query)

↓

ref.read().notifier.refresh()

↓

filteredTodosProvider.build() called

↓

New query fetches updated data

↓

UI automatically updates via ref.watch()

Mutation → Manual Refresh Cycle

```dart
// User action on UI: Delete todo
onPressed: () async {
  // Step 1: Perform mutation via notifier
  await
ref.read(todoListProvider.notifier).delete(todoId);
  //        └─ ref.read() for one-time action (no listening)

}


class TodoListNotifier extends Notifier<void> {
  ...
  Future<void> delete(String id) async {
    final repository = await
           ref.read(todoRepositoryProvider.future);
    // Mutation
    await repository.deleteTodo(id);



    // Trigger refresh of filtered todos
    ref.read(filteredTodosProvider.notifier).refresh();
  }
  ...
}
```

`ref.watch()` vs. `ref.read()`

- Use `ref.watch()` when you want to **display** data or **react** to changes
- Use `ref.read()` when you want to **perform actions** (mutations, one-time operations)

Provider Types Used - FutureProvider

Purpose: One-time async data loading, cached result

Example: Database initialization, repository creation

```
final databaseProvider = FutureProvider<AppDatabase>((ref) async {
  // Initialize and return database instance
  final database = await
      $FloorAppDatabase.databaseBuilder(...).build();
  return database;
});


final todoRepositoryProvider =
FutureProvider<TodoRepository>((ref) async {
  final database = await ref.watch(databaseProvider.future);
  return TodoRepository(database.todoDao);
});
```

Key Points:

✅ Result is cached - only runs once

✅ Perfect for database/repository initialization

Provider Types Used - NotifierProvider

Purpose: Mutable state with methods to update it

Example: Search query, filters, selections

```dart
class SearchQueryNotifier extends Notifier<String> {
  @override
  String build() => '';
  void setQuery(String query) => state = query;
  void clear() => state = '';
}

final searchQueryProvider = NotifierProvider<SearchQueryNotifier,
String>(
  () => SearchQueryNotifier(),
);
```

Key Points:

- ✅ Holds simple state (String, int, bool, enums)
- ✅ Provides methods to update state
- ✅ UI reads state and calls notifier methods
- ✅ Other providers watch and react to changes

Provider Types Used - AsyncNotifierProvider

Purpose: Async computed state that rebuilds when dependencies change

Example: Filtered/searched data from database

```dart
class FilteredTodosNotifier extends AsyncNotifier<List<Todo>> {
  @override
  Future<List<Todo>> build() async {
    // Watch dependencies - rebuilds when any change
    final searchQuery = ref.watch(searchQueryProvider);
    // Get repository
    final repository = await ref.watch(todoRepositoryProvider.future);
    // Perform filtered query
    return await repository.searchTodos(...);
  }

  Future<void> refresh() async {
    state = const AsyncValue.loading();
    state = await AsyncValue.guard(() => build());
  }
}
```

.guard handles errors and sets state accordingly

Key Points:

- ✅ Automatically rebuilds when watched providers change
- ✅ Returns `AsyncValue<T>` with loading/data/error states
- ✅ Perfect for database queries with multiple filters
- ✅ Can be manually refreshed after mutations

Provider Types Used - NotifierProvider

```
/// Notifier for todo mutations (add, edit, toggle, delete)
/// Does NOT cache todos in memory - delegates display to
FilteredTodosNotifier
/// Performance benefit: Avoids loading all todos when dealing with large
datasets
class TodoListNotifier extends Notifier<void> {
  @override
  void build() {
    // No initial state - this is a mutation-only provider
  }


  Future<void> add(ToDo newToDo) async {
    final repository = await ref.read(todoRepositoryProvider.future);
    // Mutation
    await repository.addTodo(newTodo);


    // Trigger database query to refresh the filtered todos
    ref.read(filteredTodosProvider.notifier).refresh();
  }

...
}
```

Best Practices

- Use `FutureProvider` for one-time initialization (database, repositories)
- Use `NotifierProvider` for simple mutable state (search, filters)
- Use `AsyncNotifierProvider` for computed data with dependencies
- Watch only what you need - avoid unnecessary rebuilds
- Use `.future` when awaiting FutureProvider in async context
- Call methods on `.notifier` when updating state from UI
- Use `ref.invalidate()` to force provider rebuild
- Handle `AsyncValue` states in UI (.when, .maybeWhen)

Summary - Reactive data flow

The provider architecture creates a **reactive data flow** where:

1. UI updates state providers (search, filters)
2. State changes trigger computed providers to rebuild
3. Computed providers fetch fresh data from repositories
4. UI automatically updates with new AsyncValue states
5. Mutations trigger manual refresh/invalidate cycles

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
- Use reactive data flow between app layers

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