

Biomedical Wearable Technologies
for Healthcare and Wellbeing

Persistence – Part 2

A.Y. 2021-2022

Giacomo Cappon



Outline

- **Recap**

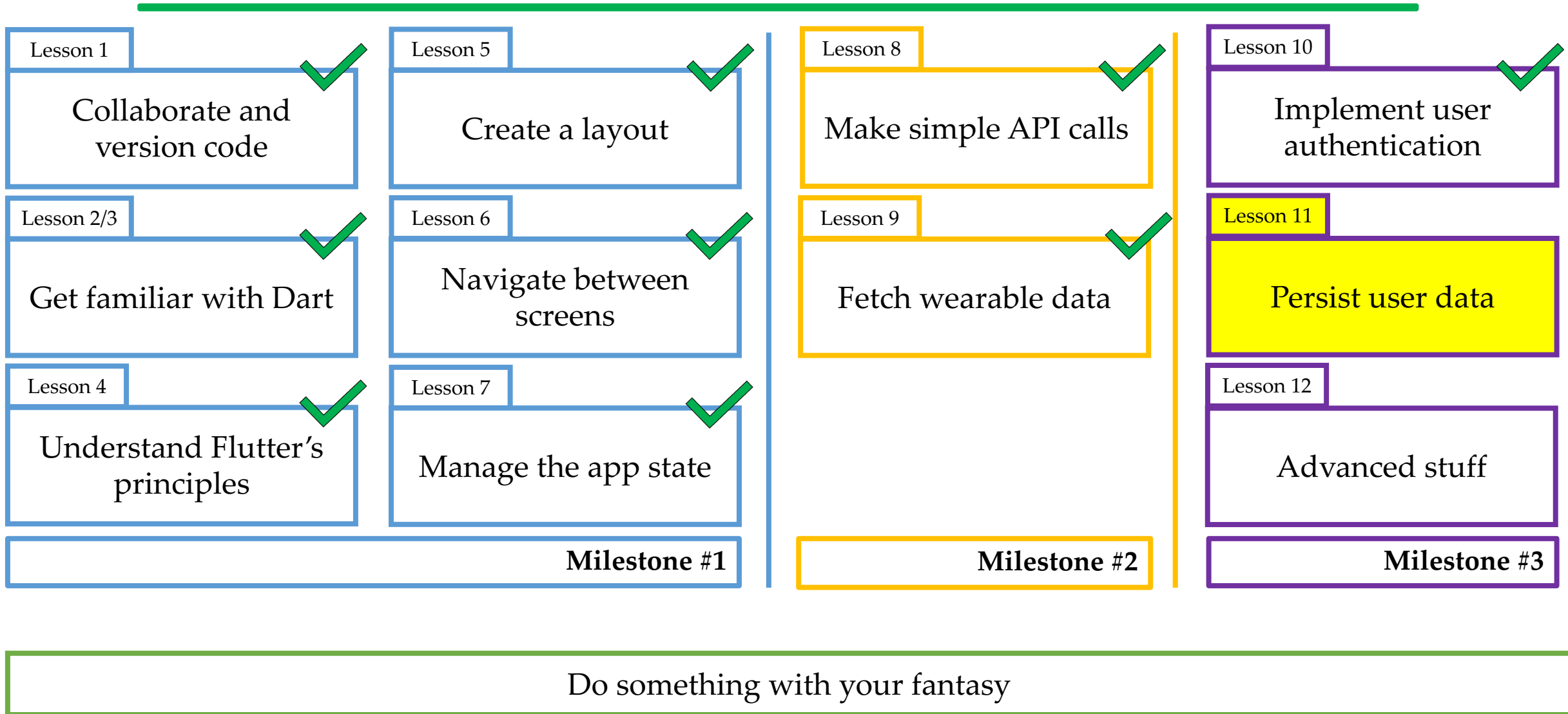
- The Floor package
- Code generation
- Case study and how to integrate a DB with Provider
- Two tips for the project

- Exercise

- Homework

- Resources

Recap



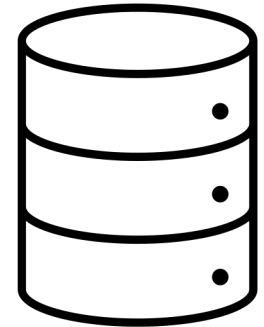
Outline

- Recap
- **The Floor package**
- Code generation
- Case study and how to integrate a DB with Provider
- Two tips for the project

- Exercise
- Homework
- Resources

Persist data in a database

- Today, we will learn how to persist data into our application.
- In particular, we will learn how to persist those data into a **local** relational database located in the mobile phone
- In Flutter, the local database is commonly a SQLite DB (a lighter version of SQL)
- Normally, a LOT of repetitive code is required in order to perform common operations, e.g.:
 - Map data model to tables and viceversa;
 - Querying the DB;
 - Open and close the DB;
 - Manage null-safety;
 - ...



Floor

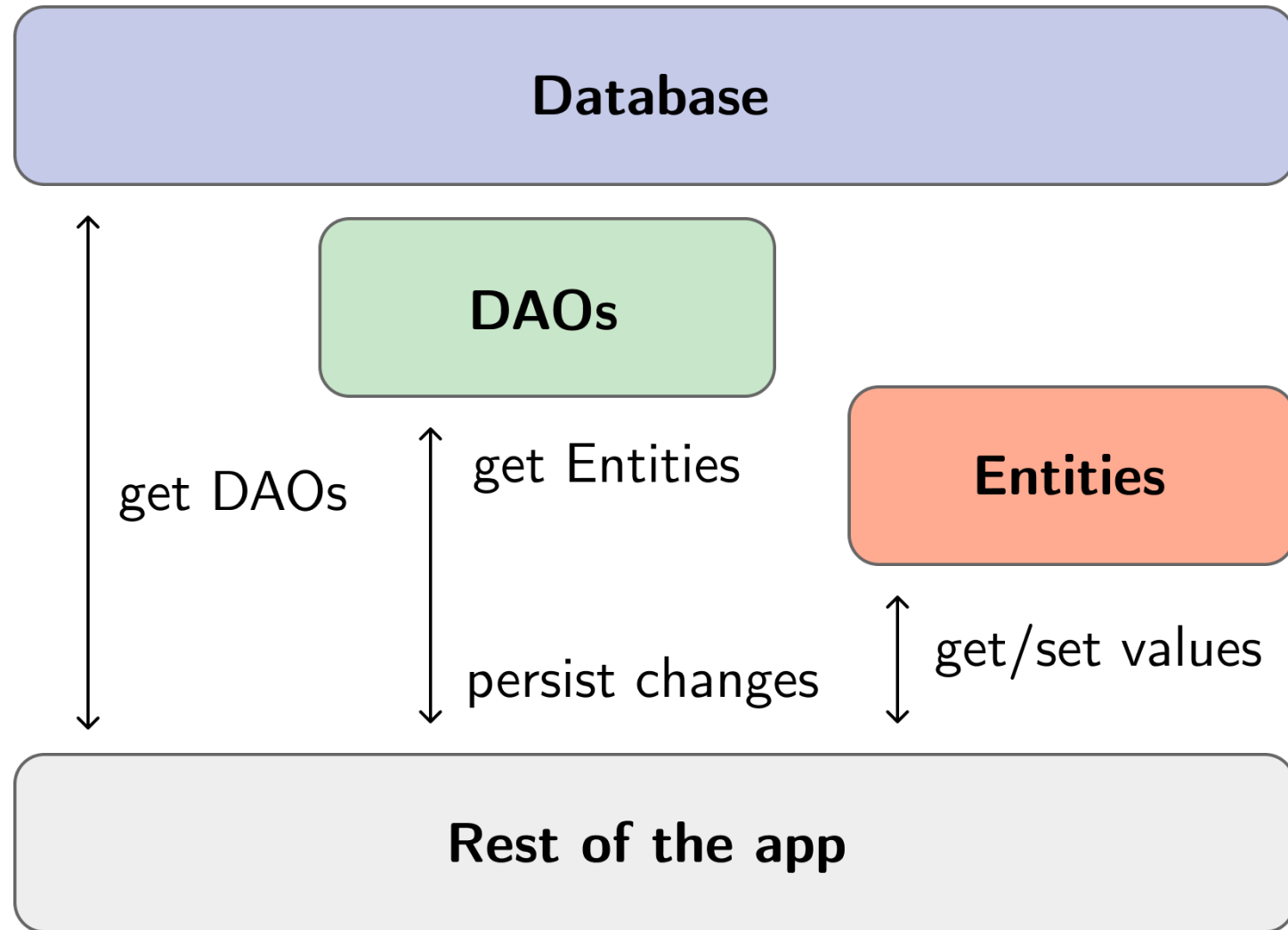
- Fortunately, there exist several packages that make our life easier when dealing with DB in Flutter.
- In this course, we will use Floor: a great package that automatically maps classes to database tables and offers full control of the database with the use of SQL:
 - <https://floor.codes/>
- As such, it will be possible to define a table structure via a Dart class and make queries via Dart methods.
- **Big plus** → Floor use the code generation concept: you define the specifics of your queries and DB structure, and Floor will generate the boilerplate code for you!

Floor

typesafe, reactive, lightweight, SQLite

```
dependencies:  
  flutter:  
    sdk: flutter  
  floor: ^1.2.0  
  
dev_dependencies:  
  floor_generator: ^1.2.0  
  build_runner: ^2.1.2
```

Overall architecture



Entity

- An Entity defines the model of a table in the DB. Floor automatically creates the mappings between the in-memory objects and DB table rows.
- It's possible to supply custom metadata to Floor by adding optional values to the `@Entity` annotation.
- It has the additional attribute of `tableName` which opens up the possibility to use a custom name for that specific entity instead of using the class name.
- `@PrimaryKey` marks the primary key column. This property has to be of type `int`. The value can be automatically generated by SQLite when `autoGenerate : true`.
- If you want a table's column to be nullable, mark the entity's field as nullable with `?`.



Entities

```
@Entity(tableName: 'person')
class Person {
    @PrimaryKey(autoGenerate: true
    final int id;

    final String name;

    Person(this.id, this.name);
}
```


Entity - Supported types and Primary Keys

- Floor entities can hold values of the following Dart types mapped to their corresponding SQLite types.
- In case you want to store sophisticated Dart objects you have to define a `TypeConverter` (see exercise 11.01 solution to see how).
- If you need to define a compound primary key (e.g., many-to-many relationship), you can use again the `@Entity` annotation

Dart	SQLite
int	INTEGER
double	FLOAT
String	TEXT
bool	INTEGER (0 = false, 1 = true)
Uint8List	BLOB

```
@Entity(primaryKeys: ['id', 'name'])  
class Person {  
    final int id;  
  
    final String name;  
  
    Person(this.id, this.name);  
}
```

Entity - Foreign Keys

- To define a foreign keys, add a list of `ForeignKeys` to the `@Entity` annotation of the referencing entity.
- `childColumns` define the columns of the current entity, whereas `parentColumns` define the columns of the parent entity.
- Foreign key actions can get triggered after defining them for the `onUpdate` and `onDelete` properties.

```
@Entity(  
    tableName: 'dog',  
    foreignKeys: [  
        ForeignKey(  
            childColumns: ['owner_id'],  
            parentColumns: ['id'],  
            entity: Person,  
        )  
    ],  
)  
class Dog {  
    @PrimaryKey()  
    final int id;  
  
    final String name;  
  
    @ColumnInfo(name: 'owner_id')  
    final int ownerId;  
  
    Dog(this.id, this.name, this.ownerId);  
}
```

DAO

- These components are responsible for managing access to the underlying SQLite database and are defined as abstract classes with method signatures and query statements.
- From the practical point-of-view, in the DAO, you will define all the queries you want to be able to execute



DAOs

```
@dao
abstract class PersonDao {
    @Query('SELECT * FROM Person')
    Future<List<Person>> findAllPersons();

    @Query('SELECT * FROM Person WHERE id = :
id')
    Stream<Person?> findPersonById(int id);

    @insert
    Future<void> insertPerson(Person person);
}
```

DAO - Insert

- `@insert` marks a method as an insertion method.
- When using the capitalized `@Insert` you can specify a conflict strategy. Else it just defaults to aborting the insert.
- These methods can return a `Future` of either
 - `void`: return nothing
 - `int`: return primary key of inserted item
 - `List<int>`: return primary keys of inserted items

Examples:

```
@Insert(onConflict:
OnConflictStrategy.rollback)
Future<void> insertPerson(Person person);
```

```
@insert
Future<List<int>> insertPersons(List<Person>
persons);
```

DAO - Update

- `@update` marks a method as an update method.
- When using the capitalized `@Update` you can specify a conflict strategy. Else it just defaults to aborting the update.
- These methods can return a `Future` of:
 - `void`: return nothing
 - `int`: return number of changed rows

Examples:

```
@Update(onConflict:
OnConflictStrategy.replace)
Future<void> updatePerson(Person person);
```

```
@update
Future<int> updatePersons(List<Person>
persons);
```

DAO - Delete

- `@delete` marks a method as a deletion method.
- These methods can return a `Future` of:
 - `void`: return nothing
 - `int`: return number of deleted rows

Examples:

```
@delete  
Future<void> deletePerson(Person person);
```

```
@delete  
Future<int> deletePersons(List<Person>  
persons);
```

Database

- To create a Database, define an abstract class which extends `FloorDatabase`.
- Furthermore, it's required to add `@Database()` to the signature of the class.
- Make sure to add the created entity to the entities attribute of the `@Database` annotation.
- In order to make the generated code work, it's required to also add the highlighted imports in the example.
- Make sure to add `part 'database.g.dart';` beneath the imports of this file. It's important to note that 'database' has to get exchanged with the filename of the database definition. In this case, the file is named `database.dart`.

Database

```
// required package imports
import 'dart:async';
import 'package:floor/floor.dart';
import 'package:sqflite/sqflite.dart' as sqflite;

import 'dao/person_dao.dart';
import 'entity/person.dart';

part 'database.g.dart'; // the generated
code will be there

@Database(version: 1, entities: [Person])
abstract class AppDatabase extends
FloorDatabase {
    PersonDao get personDao;
}
```

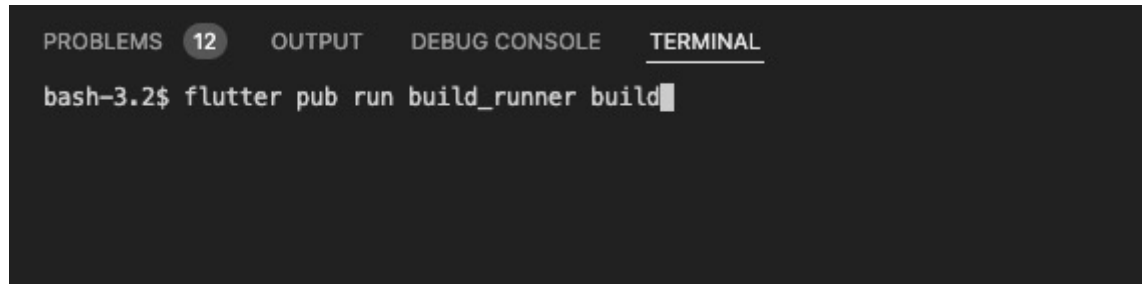
Outline

- Recap
- The Floor package
- **Code generation**
- Case study and how to integrate a DB with Provider
- Two tips for the project

- Exercise
- Homework
- Resources

The final step: generating all the code

- To generate the code simply open the terminal of VS Code and run
`flutter packages pub run build_runner build`



A screenshot of a VS Code terminal window. The terminal has tabs for 'PROBLEMS' (with a count of 12), 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active. The prompt 'bash-3.2\$' is visible, followed by the command 'flutter pub run build_runner build' with a cursor at the end.

- Remember that every time you change something in the code of the entities, the daos, or the database, you need to rerun the command!

Use the generated code

- For obtaining an instance of the database, use the generated `$FloorAppDatabase` class, which allows access to a database builder.
 - The name is composed by `$Floor` and the database class name.
 - The string passed to `databaseBuilder()` will be the database file name.
- For initializing the database, call `build()` and make sure to `await` the result.
 - You need to do it at the beginning of your app.

```
//Example on how to init the DB
final AppDatabase database = await
$FloorAppDatabase.databaseBuilder('app_database.db').build();
```

```
//Example on the use
final personDao = database.personDao;
final person = Person(1, 'Frank');
await personDao.insertPerson(person);
final result = await personDao.findPersonById(1);
```

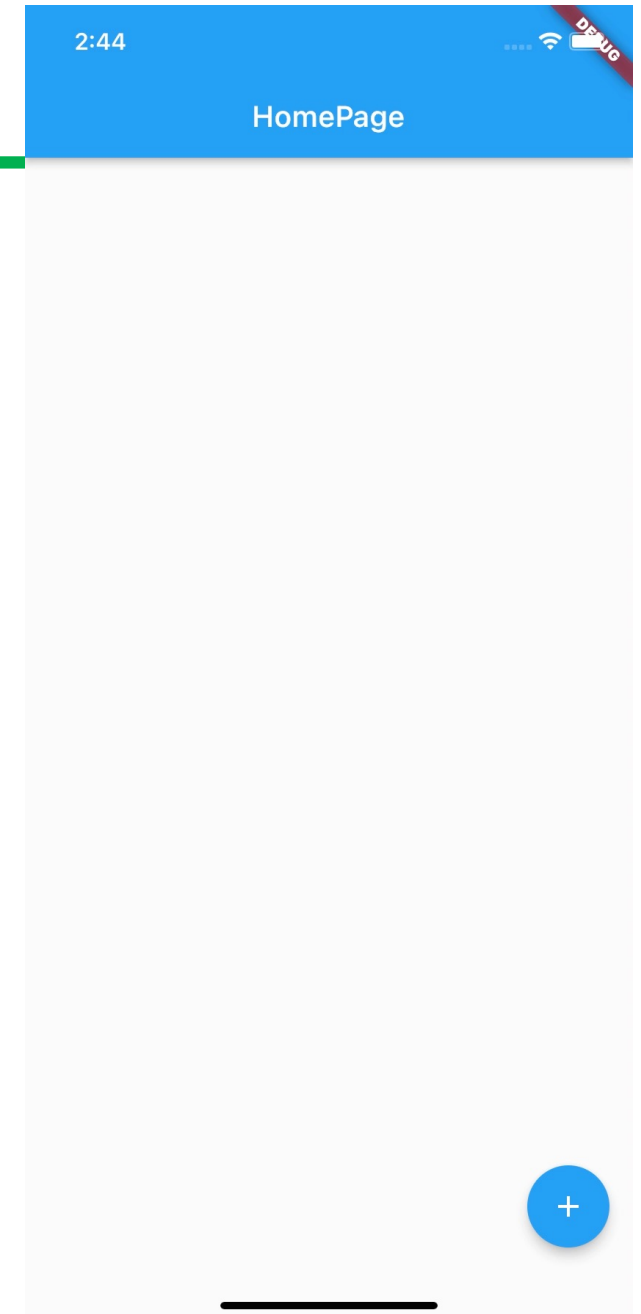
Outline

- Recap
- The Floor package
- Code generation
- **Case study and how to integrate a DB with Provider**
- Two tips for the project

- Exercise
- Homework
- Resources

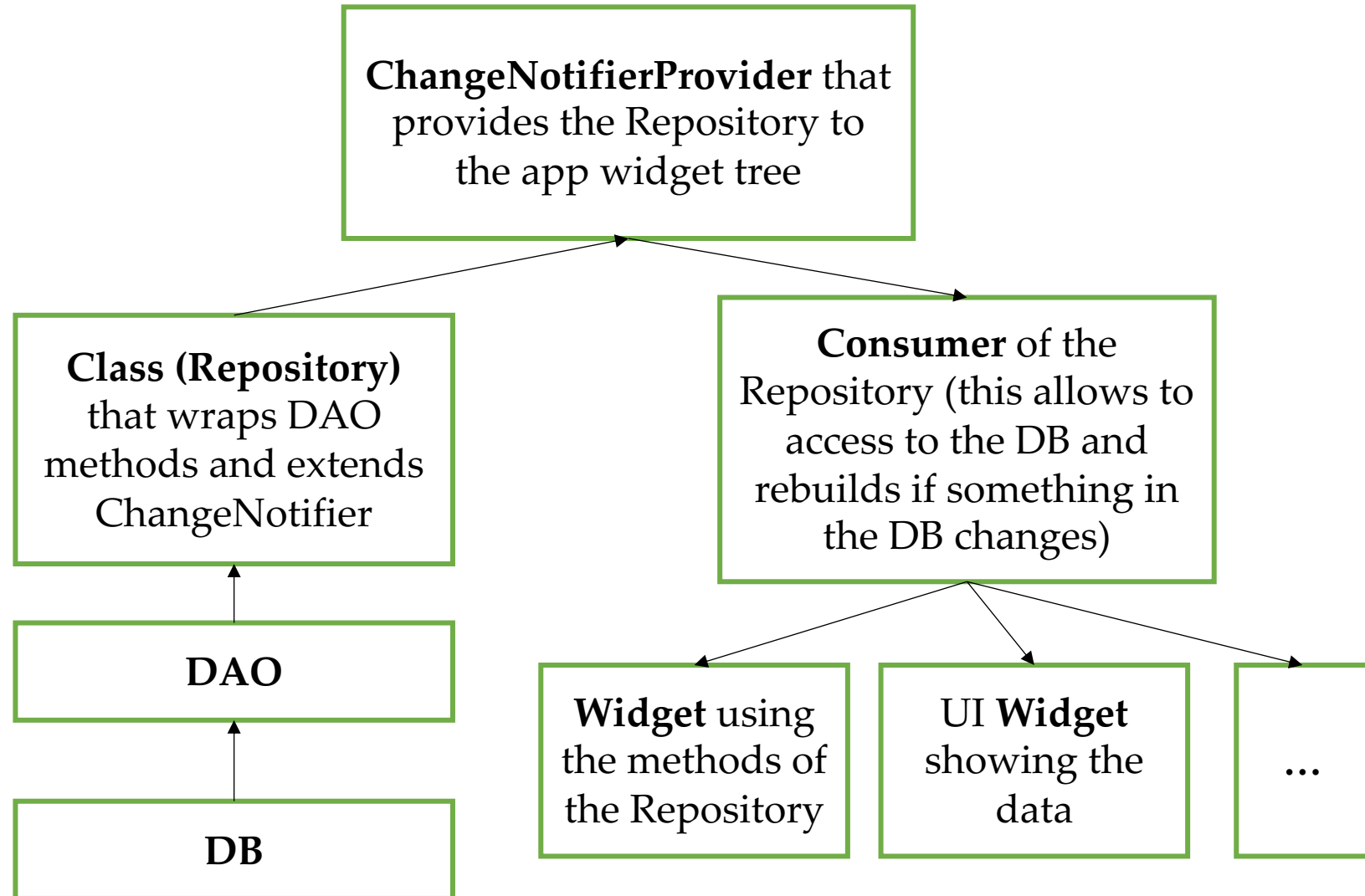
Case of study

- As simple use case, let's implement a very simple DB and a "TODO" simple app called 'busy_day'.
- The database will consist of only 1 table: Todo
- The app will consist of a simple `ListView` that visualizes, inserts and deletes the entries of the Todo table. So we have to "implement" the following three queries:
 - `SELECT` to obtain all the Todos in the table;
 - `DELETE` to remove a Todo from the table;
 - `INSERT` to create a Todo and put it in the table.



Case of study – State management with Provider

- How to “integrate” the DB in the business logic of an app using providers?
- We will take inspiration from the repository design pattern.
- What happens now if, for example, you have a ListView, that visualizes the entry of a table, and somebody deletes something from that table?



Outline

- Recap
- The Floor package
- Code generation
- Case study and how to integrate a DB with Provider
- **Two tips for the project**
- Exercise
- Homework
- Resources

Two tips for the project

During the project discussion it would be nice to see/discuss

- The ER diagram of the database
- How you decided to manage your data in terms of regulatory aspects

Outline

- Recap
- The Floor package
- Code generation
- Case study and how to integrate a DB with Provider
- Two tips for the project

- **Exercise**
- Homework
- Resources

Exercise

➤ Exercise 11.01

- Start from the code of exercise 07.01.
- Use Floor to define a proper entity, DAO, and DB to persist the meal data.
- Note that Floor does not support natively the DateTime type. As such, to define the entity of the meal you must use a TypeConverter. Try to follow the docs to implement it:
 - <https://floor.codes/type-converters/>
- If you are not able to solve the “TypeConverter issue”, you can find the solution in the lab repository.

Outline

- Recap
- The Floor package
- Code generation
- Case study and how to integrate a DB with Provider
- Two tips for the project

- Exercise
- **Homework**
- Resources

Homework

- Get familiar with Floor by taking a look at the docs
- Try to integrate Floor in your project. Here's a possible roadmap to follow:
 1. Start by defining (on paper) a simple DB made of 1 table able to store 1 "type" of data fetched from fitbitter, e.g., step counts.
 2. Create the entity that implement the table you designed
 3. Create the DAO for that entity. Again, I suggest to start easy: define just 3 queries: one for reading all the entries of the table (SELECT), one for inserting entries (INSERT), and one for deleting entries (DELETE)
 4. Create the DB and generate the code.
 5. Create the repository that wraps the DAO and extend ChangeNotifier
 6. Initialize the DB at the beginning of your code and provide the repository to the app.
 7. Try to use the DB.
 8. Go back to step 1 and add the other tables according to your needs.

Outline

- Recap
- The Floor package
- Code generation
- Case study and how to integrate a DB with Provider
- Two tips for the project

- Exercise
- Homework
- **Resources**

Resources

- Floor official docs
 - <https://floor.codes/>
- Repository design pattern
 - <https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30>