UNIVERSITY OF PADOVA
DEPARTMENT OF INFORMATION ENGINEERING

Biomedical Wearable Technologies
for Healthcare and Wellbeing

State Management

A.Y. 2022-2023

Giacomo Cappon

DEPARTMENT OF
INFORMATION
ENGINEERING
UNIVERSITY OF PADOVA

Outline

- **State management concepts**
- Provider
- Case study
- Other Provider classes

- Exercise
- Homework
- Resources

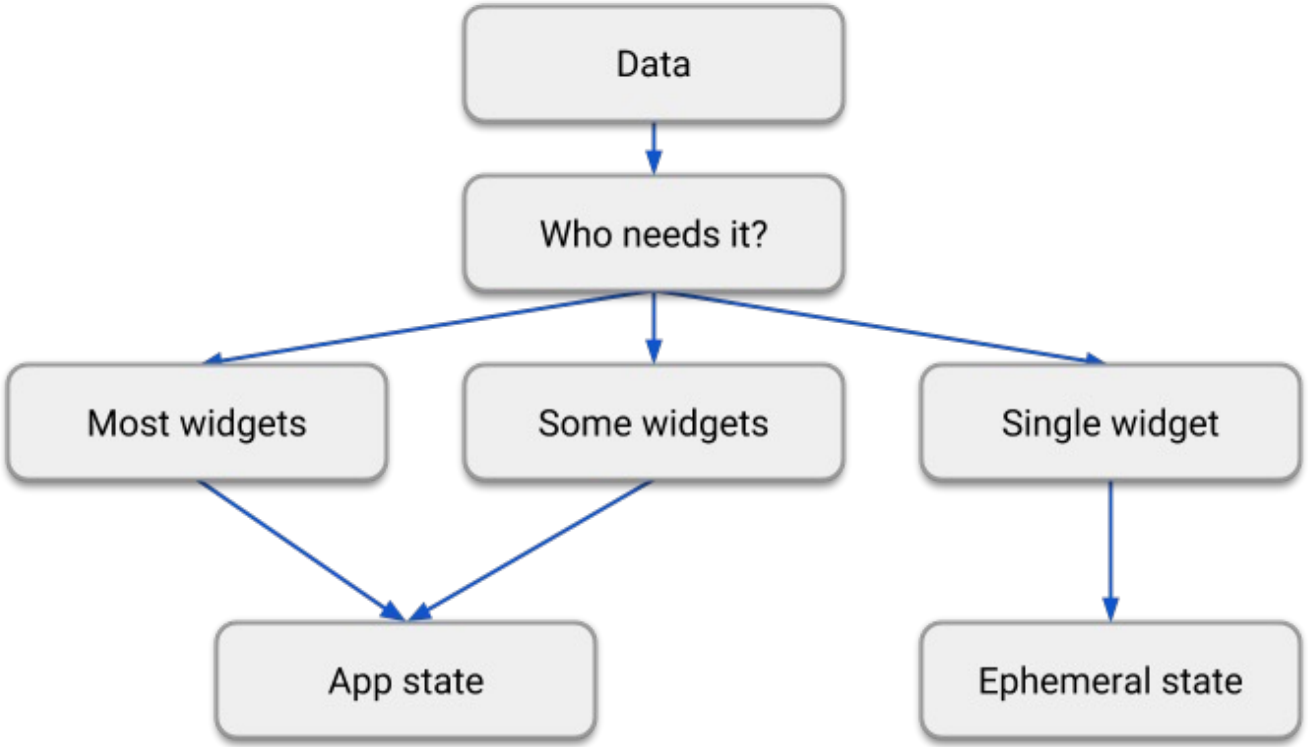
State

- State stands for everything that is necessary to define how the app and its screen behave and look at some point in time:
  - Assets
  - Variables
  - Fonts
  - …

- Conceptually can be divided in:
  - **Ephemeral state**: sometimes called *local state*, what can be striclty contained in a Widget
  - **App state**: sometimes called *shared state*, things that you want to share across many parts of the app

State

- There is no universal rule to chose if a variable is part of the ephemeral state or the app state. A diagram that can help:



Remember: Flutter is declarative

- Flutter is a declarative framework

- State is changed? Build methods are called and the UI is refreshed.

So far…

- So far, we had a grasp on state management:
  - Stateful widgets: their state can change through time
  - Stateless widgets: their state cannot change though time

- We need to understand how to change state when something occurs (e.g., a button is pressed, data change,…) and reflect those changes to the UI (rebuilding it).

- In lesson 4, we used setState():

So far…

- In lesson 4, we used setState():

```
...
@override
Widget build(BuildContext buildContext){
  return Column(
    mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text('Hello, $_word!'),
        ElevatedButton(onPressed: _changeWord, child: const
Text('Press me')),
      ],);
}//build

void _changeRandomWord(){
  setState(() {
    _word = WordPair.random().first;
  });
}//_changeRandomWord
...
```

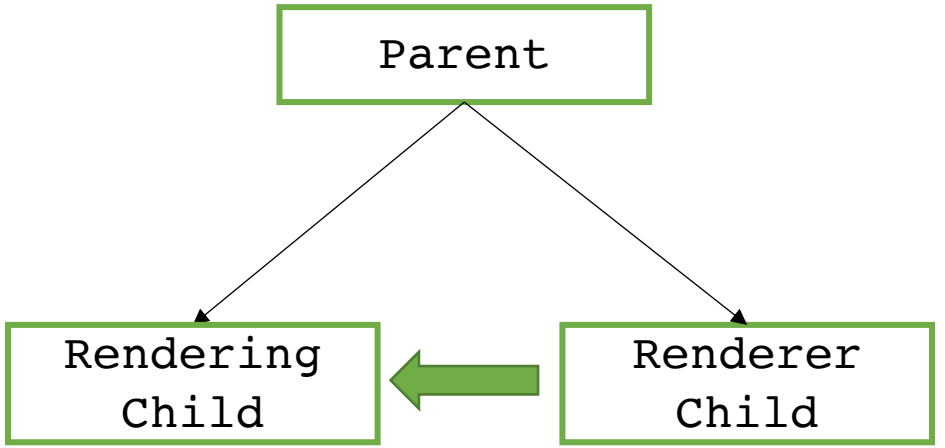
The limitation

- Why we cannot fully depend on this approach?

- In a typical widget tree, when two children of a parent are not in the same class as of the parent, rendering one child from another becomes impossible until and unless you involve the parent for the same.

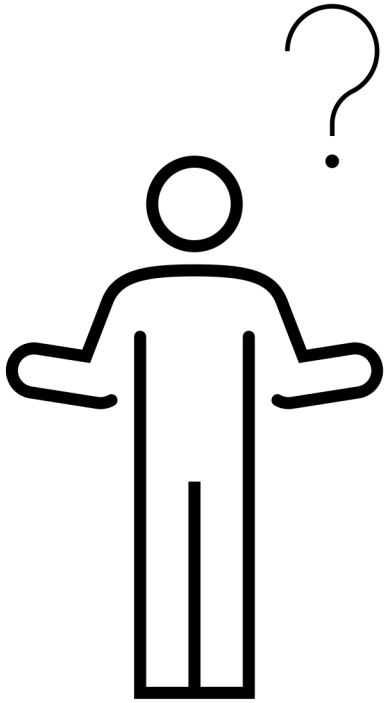
- You either have to call a setState() from parent which in turn renders both children or pass a call-back function from one child to another child via the Parent.

- In other words, app state is very messy to handle



So, how to manage state?

- There is no such a thing as a "*universal way to manage state*"



- Actually, there are a lot of possible approaches:
  - Provider
  - Riverpod
  - Redux
  - BLoC
  - …
- Every approach has its PROs and CONs. So?
- Here, we will discuss **Provider**, the recommended approach by the Flutter community.

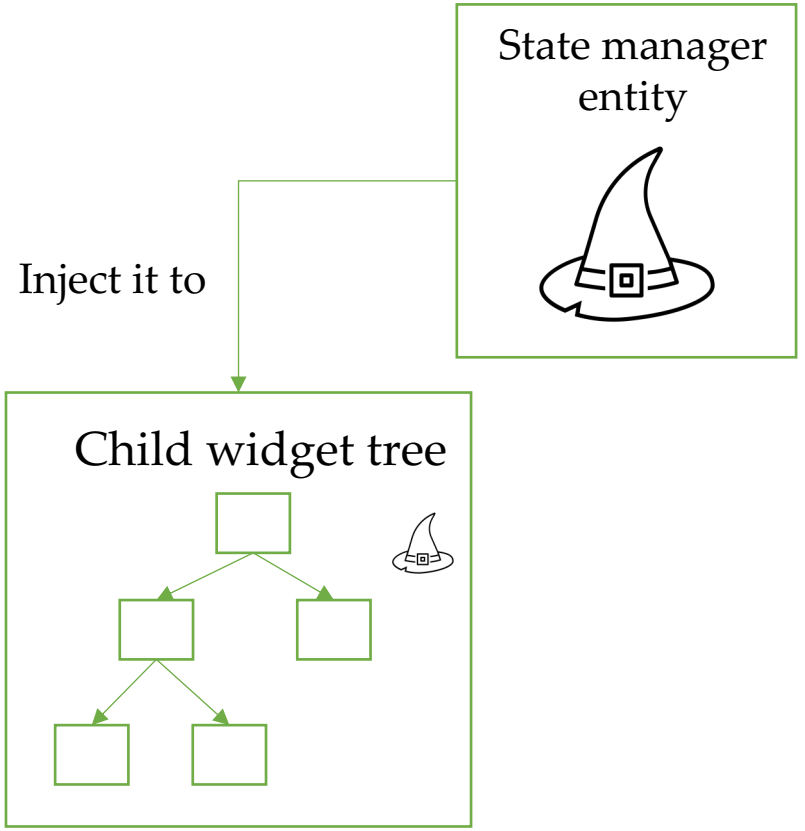
Outline

- State management concepts
- **Provider**
- Case study
- Other Provider classes

- Exercise
- Homework
- Resources

Provider core idea

- Provider wants to provide!
- The core idea is to provide the entity (one or more classes) in charge of maintaining the state down through the widget tree
- Each child will be able to access to the entity and react to state changes.



Provider classes

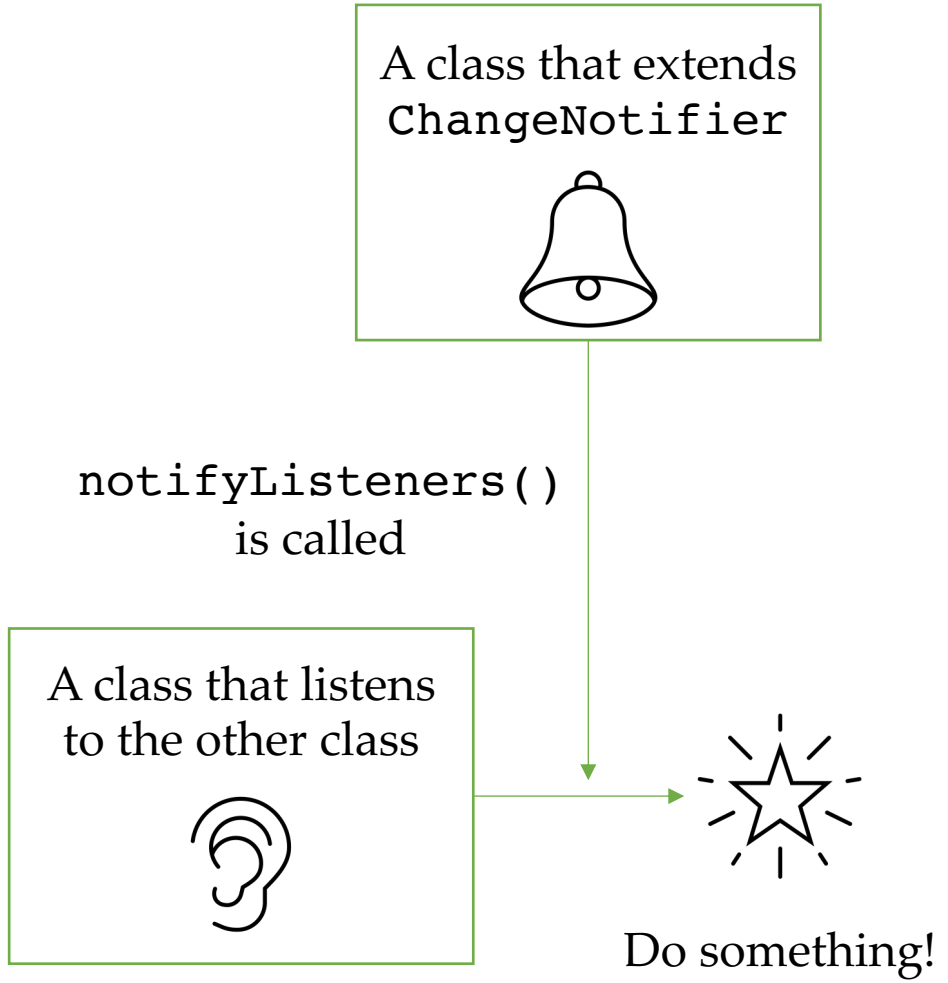
- Provider implement a set of classes, here's the most important:
  - `ChangeNotifier`
  - `ChangeNotifierProvider`
  - `Consumer`
  - `FutureProvider/StreamProvider`
  - `MultiProvider`
  - `ProxyProvider`

Provider classes

- Provider implement a set of classes, here's the most important:
  - **`ChangeNotifier`**
  - **`ChangeNotifierProvider`**
  - **`Consumer`**
  - `FutureProvider/StreamProvider`
  - `MultiProvider`
  - `ProxyProvider`

Provider classes: `ChangeNotifier`

- `ChangeNotifier` is a class that can notify **listeners** of any changes in the state.
- You usually extend the `ChangeNotifier` for models so you can send notifications when your model changes.
- When something in the model changes, you call **`notifyListeners()`** and whoever is listening can use the newly changed model.



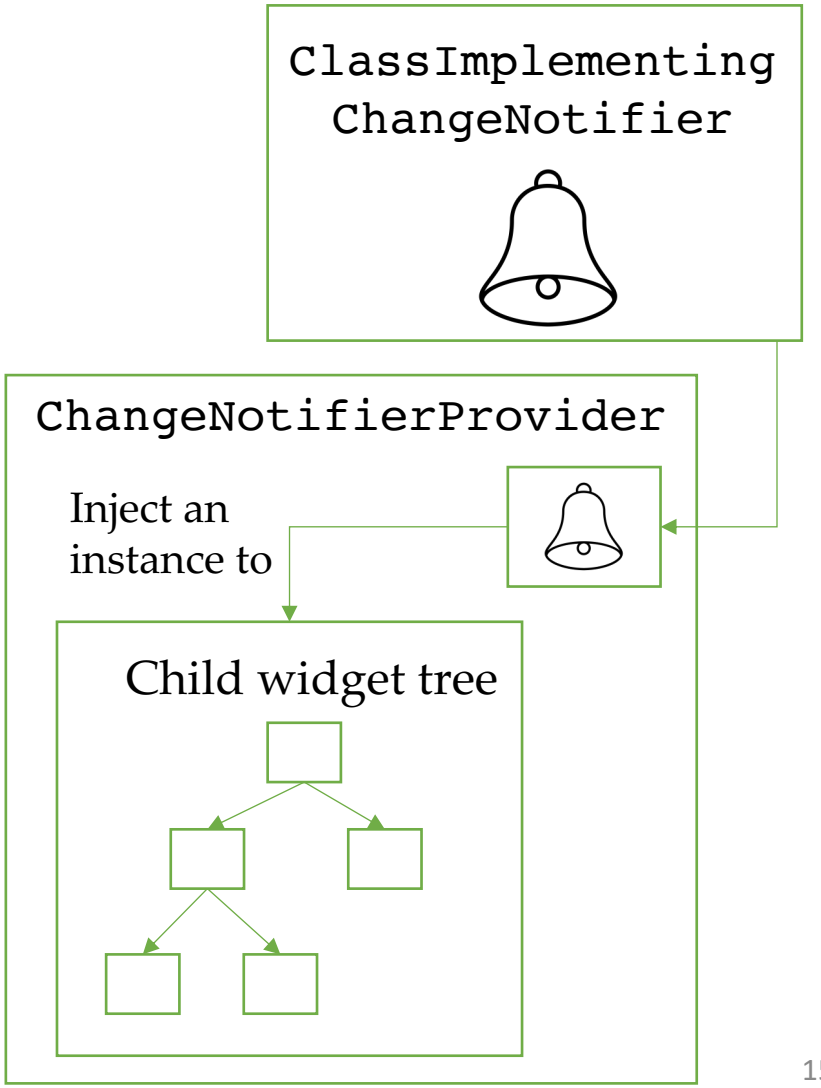
Provider classes: `ChangeNotifierProvider`

- `ChangeNotifierProvider` is a class that wraps a class that implements `ChangeNotifier` and **provide it** to its descendants.

- Now the widget tree can access to it (and use it!)

- Synthax:

```
ChangeNotifierProvider(
    create: (context) =>
    ClassImplementingChangeNotifier(),
    child: <widget_tree>,
);
```

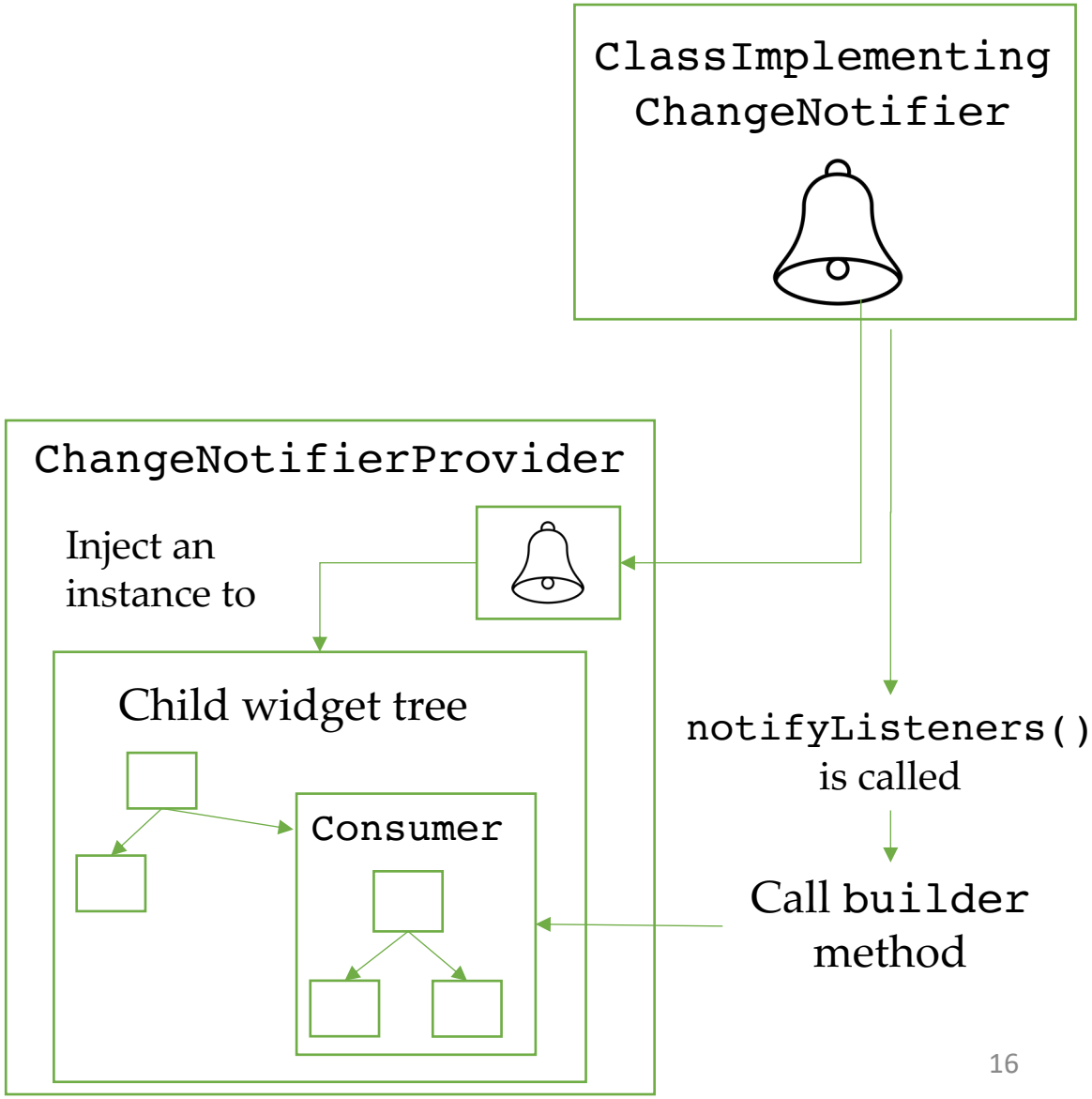


Provider classes: `Consumer`

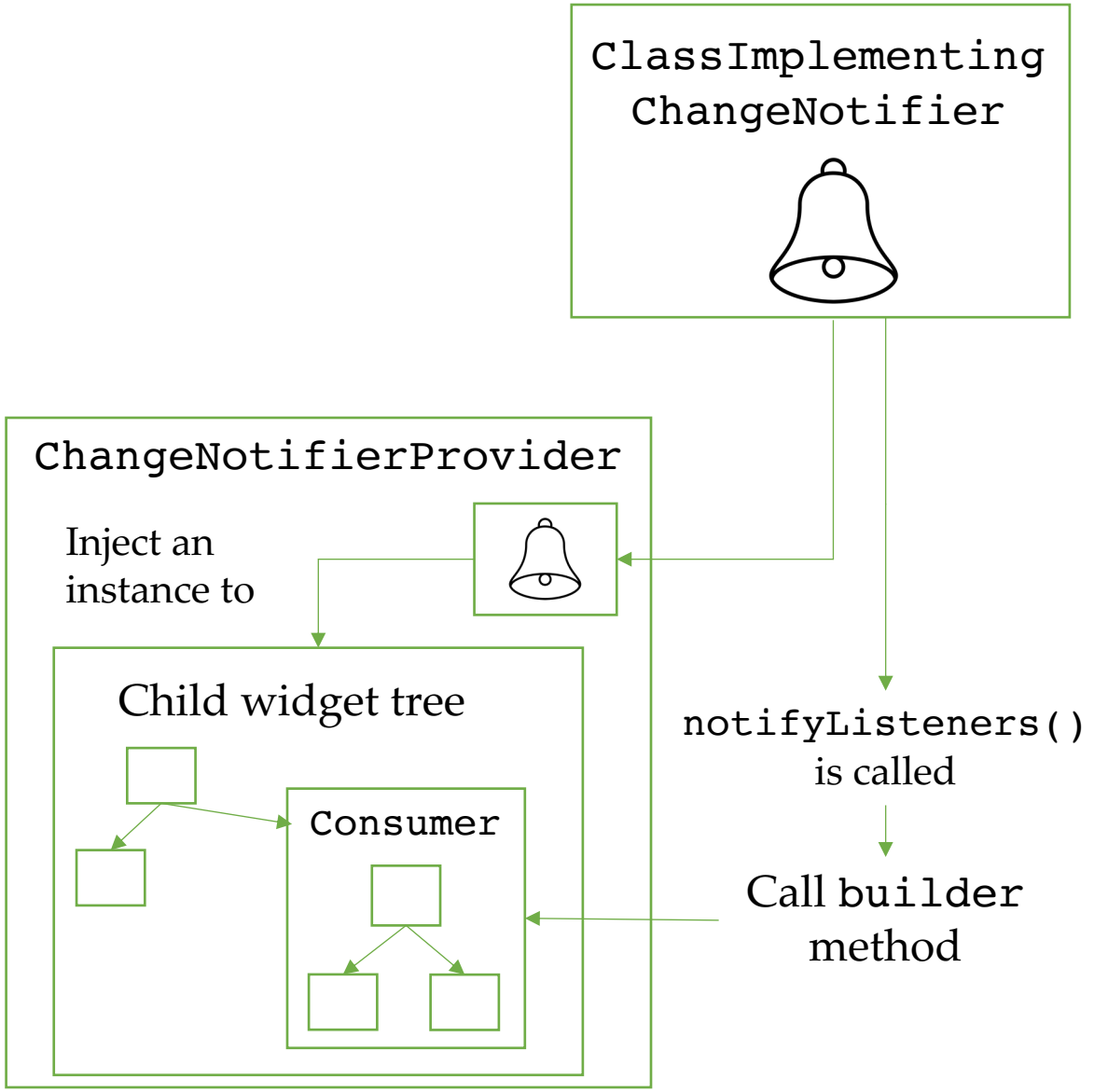
- `Consumer` is a widget that listens for changes in a class that implements `ChangeNotifier`, then rebuilds the widget tree below itself when it finds any.

- Whenever `notifyListeners()` is called, the `Consumer's builder` function is called. Synthax:

```
return
Consumer<ClassImplementingChangeNotifier>(
 builder: (context, cart, child) {
  return Text("Total price:
    ${cart.totalPrice}");
  },
);
```



Wrapping up: Provider (core) flow



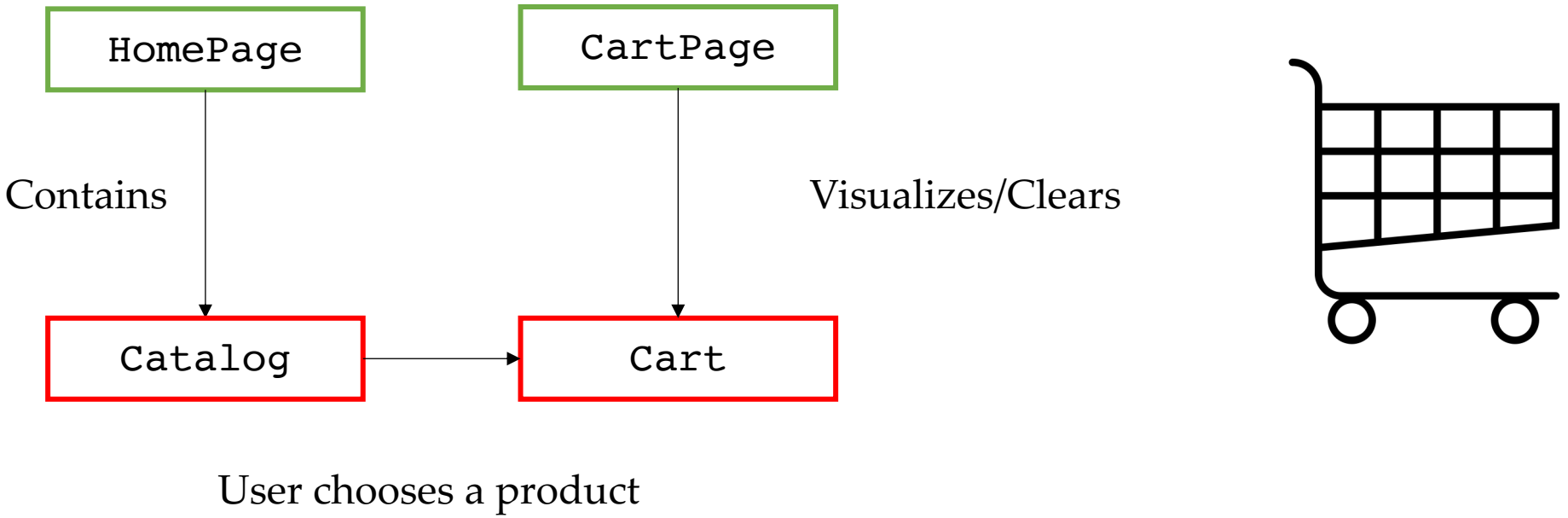
Outline

- State management concepts
- Provider
- **Case study**
- Other Provider classes

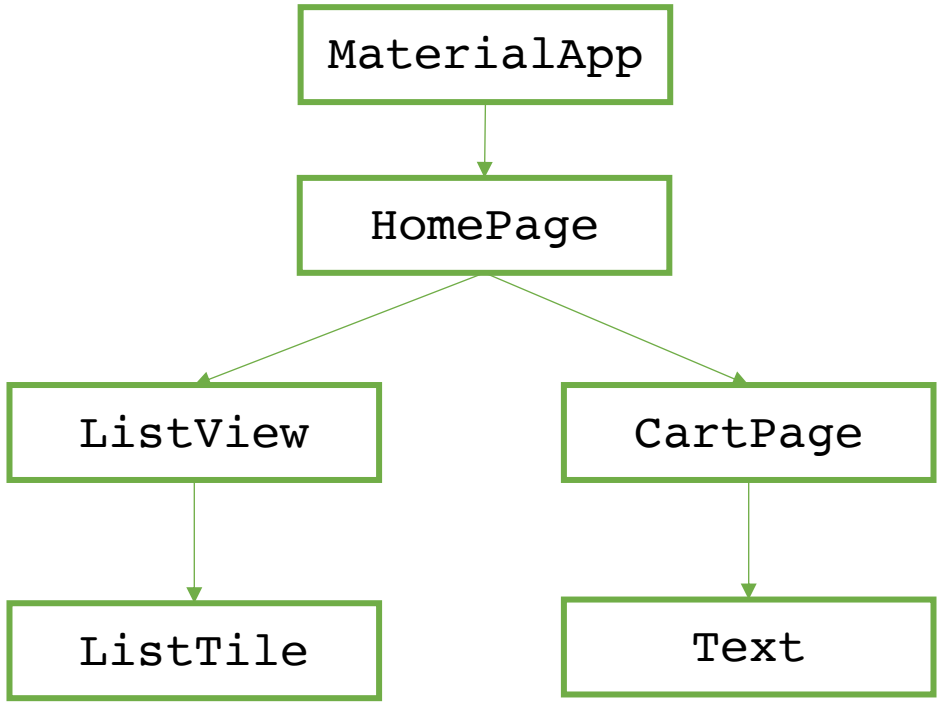
- Exercise
- Homework
- Resources

Case of study

- We will build a (too) simple e-commerce app. Users can choose, in the `HomePage`, products from a catalog and add them to the cart. In a different screen, `CartPage`, the number of items in the cart is visualized.



Case of study – UI widget tree



To visualize the `Catalog`

To visualize the number of items in the `Cart`

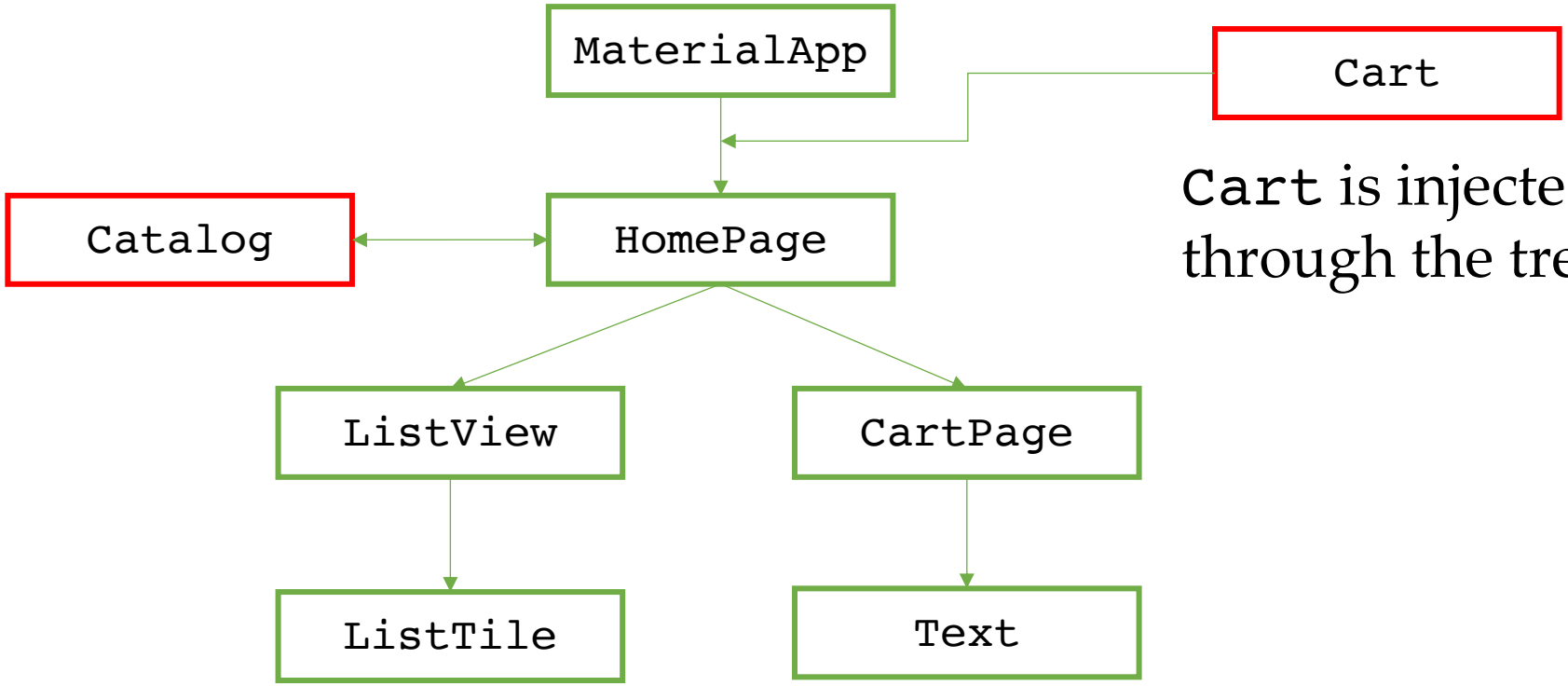
- How to manage the catalog and the cart? With two class (model).
  - No problems with the `Catalog` (Ephemeral state)…
  - On the other hand, `Cart` must be shared (App state) Where to put it?

`Catalog`

`Cart`

Lift the state up

- The idea is "Lift the state up"

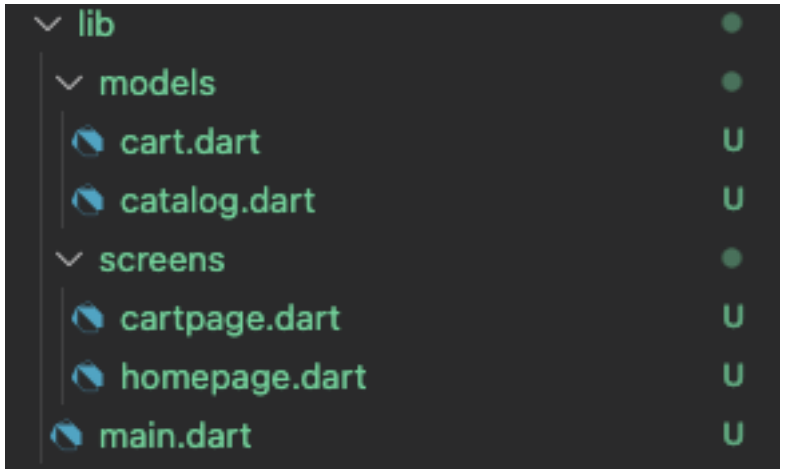


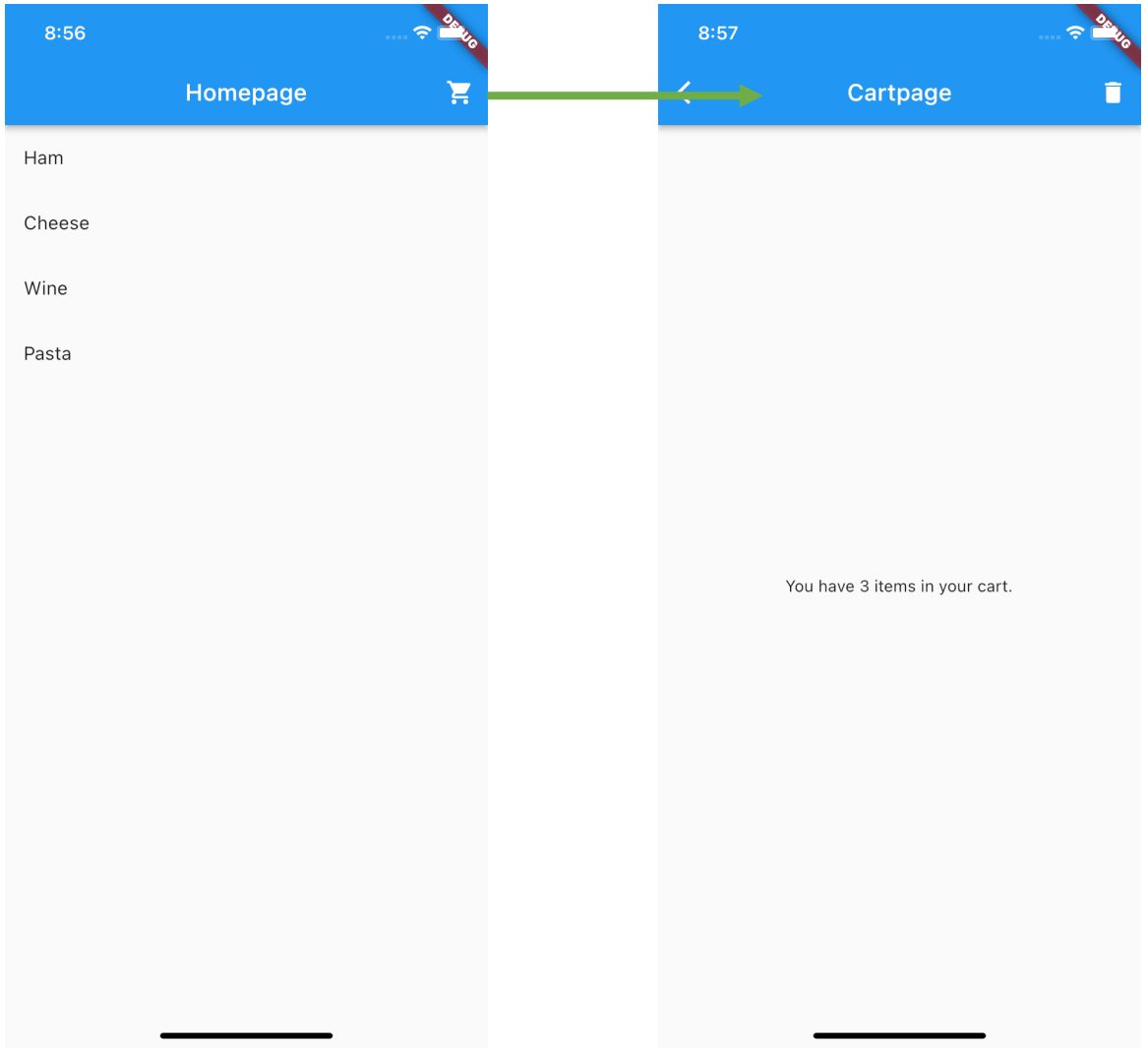
`Cart` is injected through the tree.

`Catalog` is just a variable of `HomePage`.

Case of study - Live

- Let's create a new project "shopper" that implements the case of study and shows how to use Provider.





Full example in lab_07-state_management/shopper/

Outline

- State management concepts
- Provider
- Case study
- **Other Provider classes**

- Exercise
- Homework
- Resources

Provider classes

- Provider implement a set of classes, here's the most important:
  - `ChangeNotifier`
  - `ChangeNotifierProvider`
  - `Consumer`
  - **`FutureProvider/StreamProvider`**
  - **`MultiProvider`**
  - **`ProxyProvider`**

Provider classes: `FutureProvider/StreamProvider`

- `FutureProvider` and `StreamProvider` are the same thing as `ChangeNotifier` but they work with `Futures` and `Streams` instead of `ChangeNotifier`.

- Synthax example:

```
FutureProvider(
  initialData: null,
  create: (context) => <get_a_future>,
  child: <widget>,
);
StreamProvider(
  initialData: null,
  create: (context) => <get_a_stream>,
  child: <widget>,
);
```

`initialData` is used as value while the Future is loading.

`create` specifies the `Future` object.

Provider classes: `MultiProvider`

- What if you need to inject more than one provider through the widget tree? Use `MultiProvider.`

- Synthax example:

List of providers.

```
MultiProvider(
  providers: [
ChangeNotifierProvider(...),
FutureProvider(...),
  ],
  child: <widget_tree>,
);
```

Provider classes: `ProxyProvider`

- What if you have two models that you want to provide, but one of the models depends on the other one? In that case you can use a `ProxyProvider`.

- A `ProxyProvider` takes the value from one provider and lets it be injected into another provider.

- Synthax example:

```
MultiProvider(
  providers: [
    ChangeNotifierProvider<MyModel>(
      create: (context) => MyModel(),
    ),
    ProxyProvider<MyModel, AnotherModel>(
      update: (context, myModel, anotherModel) =>
AnotherModel(myModel),
    ),
  ],
```

Provider classes – Much more

- Of course, the Provider package can do much more
  - `ListenableProvider`
  - `ValueListenableProvider`
  - `ChangeNotifierProxyProvider`
  - `Selector`
  - `...`
- To learn more, take a look at this useful article:
  - Making sense of all those Flutter Providers
    - https://medium.com/flutter-community/making-sense-all-of-those-flutter-providers-e842e18f45dd

Outline

- State management concepts
- Provider
- Case study
- Other Provider classes

- **Exercise**
- Homework
- Resources

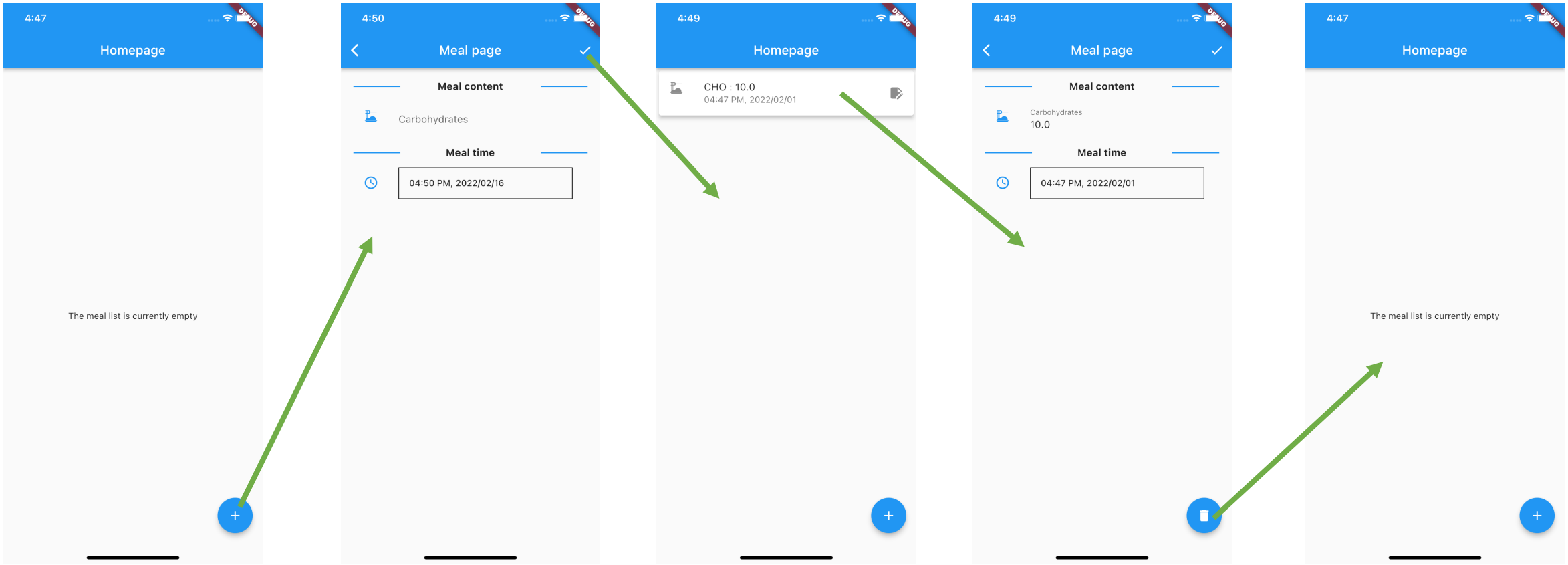
Exercise

- Exercise 07.01

  - Create a new project ‘meal_manager’, an app that stores the meal intakes of a user in terms of carbohydrate content and meal timing.

  - The app needs to implement the following functionalities:
    - When a user opens (or restarts) the app, an empty list is showed;
    - By tapping a button, the user navigates to another page where he can add a new meal (CHO content and timing). Once he/she is done, the user taps a button and navigates back to the home page. The home page must show the updated meal list with the new meal just created;
    - The user can select a meal from the list. If he/she does that, he/she navigates to another page where he/she can edit or delete the meal entry from the list.

Exercise: here's a possible idea



Homepage is initially empty

When the button is tapped the user navigates to another page where he/she can add a new meal

When the user is done the new meal is added to the list

When the meal is tapped, the user navigates to a new screen where he/she can edit or delete it

If the meal is edited/deleted the list is updated.

Outline

- Recap
- State management concepts
- Provider
- Case study
- Other Provider classes

- Exercise
- **Homework**
- Resources

Homework

- Get familiar with Provider
- Take a look to other approaches, e.g., Riverpod or BLoC. Maybe there is something that fits better your way of thinking!
- Take a look at my solution. You will find some useful code.

Outline

- State management concepts
- Provider
- Case study
- Other Provider classes

- Exercise
- Homework
- **Resources**

Resources

- State management
  - https://docs.flutter.dev/development/data-and-backend/state-mgmt/intro
- Making sense of all those Flutter Providers
  - https://medium.com/flutter-community/making-sense-all-of-those-flutter-providers-e842e18f45dd
- List of state management approaches
  - https://docs.flutter.dev/development/data-and-backend/state-mgmt/options