

Name: NIYONKURU Jean De La Goix

Reg N°: 223003235

Year 3

Semester II

MOBILE DESIGN AND SYSTEM

Assignment #1

1) Flutter Offers several state management approaches, each with different trade-offs in complexity, scalability, and ease of use. Below are four widely used solutions.

1.1. Provider

Provider is officially recommended by the flutter team as the starting point for state management. It is built on top of InheritedWidget, a core flutter mechanism, but with a much simpler API. It uses ChangeNotifier to hold state and notifies listeners when data changes. Widgets subscribe to state changes ~~and~~ using Consumer<T> or Provider.of<T>(context). It is best for beginners, small to medium-sized applications, straight forward state.

1.2 Riverpod

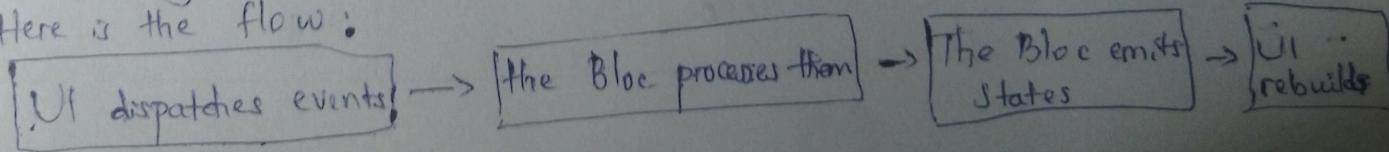
It was created by the same author as Provider (Remi Rousselet) as an evolved, improved alternative. It is completely independent of the widget tree (providers can be declared globally and tested in isolation), and offers compile-time safety which means using a ~~compi~~ provider incorrectly causes a compile error, not a runtime crash.

It also supports multiple types like StateProvider, FutureProvider, StreamProvider, StateNotifierProvider, and many more, and it best suits medium to large apps, teams that want strong testability and type safety.

1.3. Bloc - (Business logic Component)

This is a design pattern that enforces strict separation between UI and business logic.

Here is the flow:



It actually uses Streams internally, making it reactive and suitable to complex asynchronous data flows. It is suitable for large enterprise applications with complex feature

1.4 Getx

This is an all in one Flutter package that handles state management, routing, and dependency injection together.

It uses reactive variables (Rx types) and `obx()` widgets to rebuild only what changed. It has the so called Minimal boilerplate which significantly less setup code compared to Bloc or even Provider. It does not enforce a strict architecture, giving developers more flexibility (but less structure).

It is best for rapid prototyping, solo developers, apps that need speed over strict patterns.

2.) When to use each state management.

The table below summarises the recommended state management solution based on the type of project or requirement.

Situation	Provider	Riverpod	Bloc	Getx
Small Applications	Best choice due to low overhead.	Good choice as it supports isolation	Overtkill due to much setup.	Great choice with minimal code
Medium applications	Good choice	Best choice and recommended	Good	Good due to faster set up.
Large / Enterprise applications	Limited i.e. works but not ideal	Good	Best and recommended	Partial - works but not ideal.
Team projects	Good (always good for starting up)	Good	Best and highly recommended due to UI and business logic separation	Less ideal and not recommended for this scenario
Fast development	Good starting point	Good as it is an evolution of provider	Slow and not recommended	Best option with few codes
Strict architecture requirement	Partial and therefore not ideal choice	Good due to isolation	Best as it separates UI from business logic	Not recommended for this situation

Flutter State Management

Section 3 - How Provider is Used in Flutter

Provider is a wrapper around Flutter's InheritedWidget that makes it easier to share and manage state across the widget tree. The following steps walk through a complete Provider implementation using a simple counter application.

Step 1 - Adding the Dependency

Open your project's pubspec.yaml file and add the provider package under dependencies:

```
30  dependencies:
31    flutter:
32      sdk: flutter
33    provider: ^6.1.1
34
```

Then run the following command in your terminal to install the package:

```
C:\Users\CTrader\Desktop\3 rd year\Semester 2\Flutter\dart_programming_provider\my_app>flutter pub get
Resolving dependencies... (4.8s)
Downloading packages... (52.4s)
  async 2.12.0 (2.13.0 available)
```

Console output

```
  vm_service 14.3.1 (15.0.2 available)
Changed 2 dependencies!
15 packages have newer versions incompatible with dependency constraints.
Try `flutter pub outdated` for more information.
```

Step 2 - Creating a State Class

Create a Dart class that extends ChangeNotifier. This class is responsible for holding the application state and exposing methods to modify it. When state changes, notifyListeners() is called to inform all listening widgets.

The screenshot shows the VS Code interface with the Explorer sidebar on the left. Under the 'DART_PROGRAMMING_PROVIDER' folder, files like 'my_app', '.dart_tool', '.idea', 'lib', 'counter_state.dart', 'main.dart', and others are listed. The 'counter_state.dart' file is selected and open in the main editor area. The code defines a state class 'CounterState' that extends 'ChangeNotifier'. It has a private '_count' variable and public getters and setters for it. The 'increment()' method increments the count and notifies listeners, while 'reset()' sets the count back to zero and notifies listeners.

```
my_app > lib > counter_state.dart
1 // counter_state.dart
2 import 'package:flutter/foundation.dart';
3
4 class CounterState extends ChangeNotifier {
5   int _count = 0;
6
7   // Getter to expose private state
8   int get count => _count;
9
10 void increment() {
11   _count++;
12   notifyListeners(); // Triggers UI rebuild
13 }
14
15 void reset() {
16   _count = 0;
17   notifyListeners();
18 }
19 }
```

Step 3 - Providing the State

Wrap the root widget (or a subtree) with `ChangeNotifierProvider`. This makes the state available to all descendant widgets in the widget tree. The `create` callback instantiates the state class once.

The screenshot shows the VS Code interface with the Explorer sidebar on the left. Under the 'DART_PROGRAMMING_PROVIDER' folder, files like 'my_app', '.dart_tool', '.idea', 'lib', 'counter_page.dart', 'counter_state.dart', and 'main.dart' are listed. The 'main.dart' file is selected and open in the main editor area. The code imports necessary packages and defines the `main()` function. Inside `main()`, a `ChangeNotifierProvider` is created with a `create` callback that returns an instance of `CounterState`. The provider's child is the `MyApp` widget.

```
my_app > lib > main.dart > ...
1 import 'package:flutter/material.dart';
2 import 'package:provider/provider.dart';
3 import 'counter_state.dart';
4 import 'counter_page.dart';
5
6 Run | Debug | Profile
7 void main() {
8   runApp(
9     ChangeNotifierProvider(
10       create: (context) => CounterState(),
11       child: MyApp(),
12     ), // ChangeNotifierProvider
13   );
14
15
16 class MyApp extends StatelessWidget {
17   const MyApp({super.key});
18 }
```

Step 4 - Accessing the State

There are two main ways to read the state inside a widget:

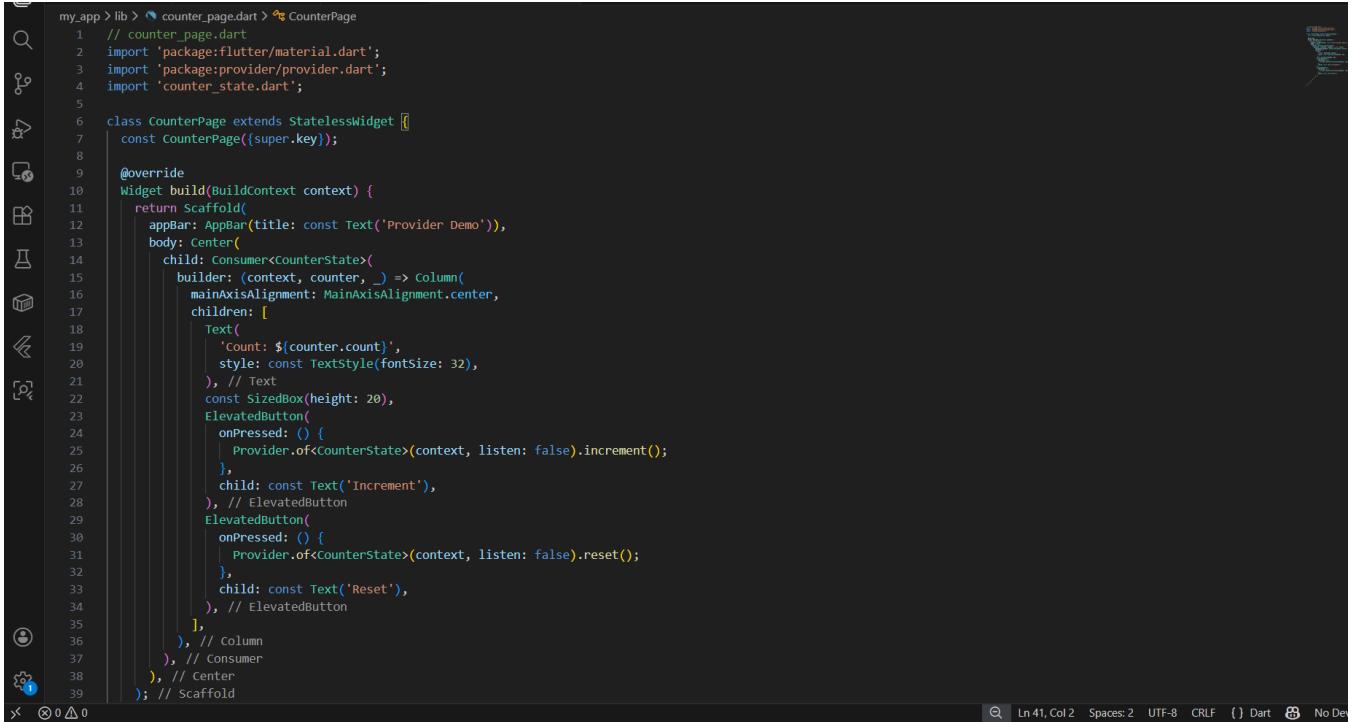
- `Provider.of<T>(context)` — reads the state directly, rebuilds the widget on every change.
- `Consumer<T>` — wraps only the widget that needs to rebuild, for better performance.

The screenshot shows the VS Code interface with the Explorer sidebar on the left. Under the 'DART_PROGRAMMING_PROVIDER' folder, files like 'my_app', '.dart_tool', '.idea', 'lib', 'main.dart', and others are listed. The 'main.dart' file is selected and open in the main editor area. The code defines the `build` method of the `MyApp` widget. It returns a `MaterialApp` widget with a title of 'Provider Demo' and a home page of `CounterPage()`.

```
main.dart
18
19   @override
20   Widget build(BuildContext context) {
21     return MaterialApp(
22       title: 'Provider Demo',
23       home: CounterPage(),
24     ); // MaterialApp
25   }
26 }
```

Step 5 - Updating the State

To update the state, call methods defined in the state class through the provider. Use `listen: false` when you only need to trigger an action and do not need the widget to rebuild.



```

my_app > lib > counter_page.dart > CounterPage
  1 // counter_page.dart
  2 import 'package:flutter/material.dart';
  3 import 'package:provider/provider.dart';
  4 import 'counter_state.dart';
  5
  6 class CounterPage extends StatelessWidget {
  7   const CounterPage({super.key});
  8
  9   @override
 10   Widget build(BuildContext context) {
 11     return Scaffold(
 12       appBar: AppBar(title: const Text('Provider Demo')),
 13       body: Center(
 14         child: Consumer<CounterState>(
 15           builder: (context, counter, _) => Column(
 16             mainAxisAlignment: MainAxisAlignment.center,
 17             children: [
 18               Text(
 19                 'Count: ${counter.count}',
 20                 style: const TextStyle(fontsize: 32),
 21               ), // Text
 22               const SizedBox(height: 20),
 23               ElevatedButton(
 24                 onPressed: () {
 25                   Provider.of<CounterState>(context, listen: false).increment();
 26                 },
 27                 child: const Text('Increment'),
 28               ), // ElevatedButton
 29               ElevatedButton(
 30                 onPressed: () {
 31                   Provider.of<CounterState>(context, listen: false).reset();
 32                 },
 33                 child: const Text('Reset'),
 34               ), // ElevatedButton
 35             ],
 36           ), // Column
 37         ), // Consumer
 38       ), // center
 39     ); // Scaffold

```

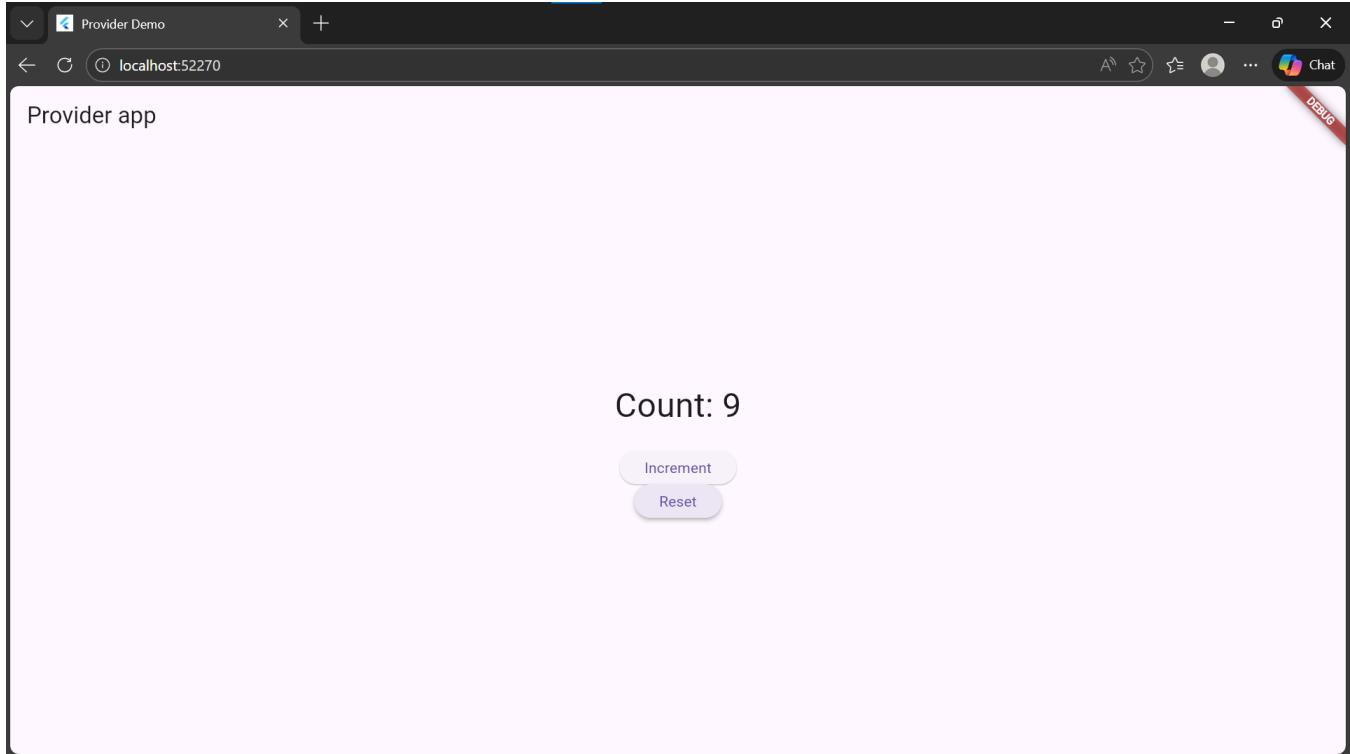
Ln 41, Col 2 Spaces: 2 UTF-8 CRLF {} Dart No Dev

Step 6 - How UI Rebuild Happens

When increment() is called, it modifies _count and calls notifyListeners(). Provider intercepts this notification and triggers a rebuild of every widget that is listening to CounterState. The diagram below illustrates the data flow:



Console Output (demonstrating notifyListeners() behavior):



Summary - Provider Key Concepts

Concept	Description
<code>ChangeNotifier</code>	Base class for state objects; call <code>notifyListeners()</code> to trigger UI updates.
<code>ChangeNotifierProvider</code>	Wraps the widget tree and makes state accessible to descendants.
<code>Consumer<T></code>	Rebuilds only its subtree when the state changes — preferred for performance.
<code>Provider.of<T></code>	Reads state directly; use <code>listen: false</code> when only calling methods.
<code>notifyListeners()</code>	Signals Provider to rebuild all widgets listening to this state object.

— End of Assignment #1 —