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 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 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:

UI dispatches events → the Bloc processes them → The Bloc emits states → UI rebuilds

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

1.4 GetX

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

It user 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	Overkill due to much set up.	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:

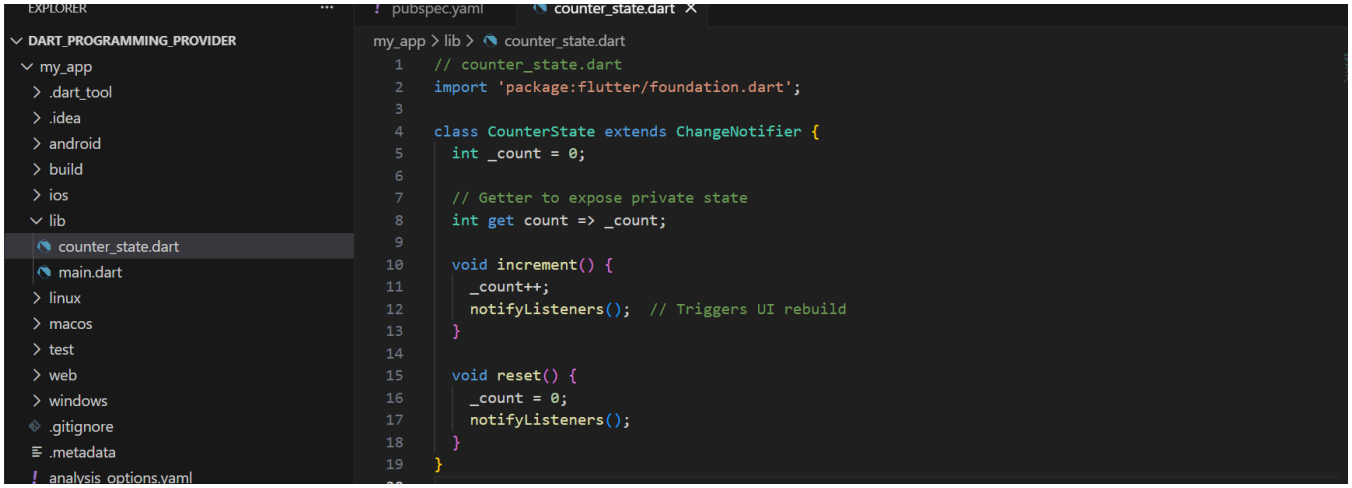
```
C:\Users\CTrader\Desktop\3 rd year\Semester 2\Flutter\dart_programming_provider\my_app>flutter pub get
Resolving dependencies... (4.8s)
Downloading packages... (52.4s)
  async 2.12.0 (2.13.0 available)
```

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.
```

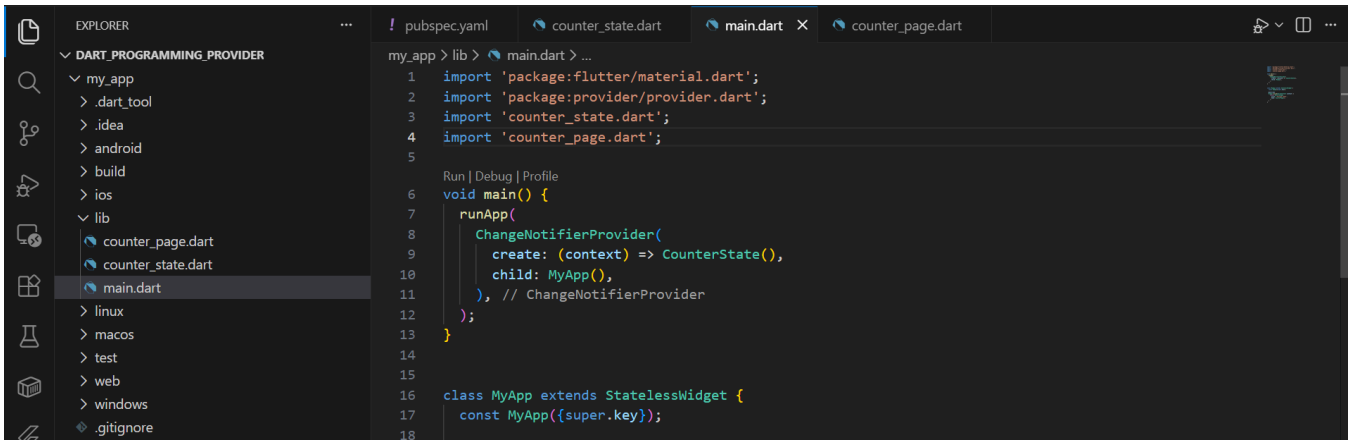
Step 2 - Creating a State Class

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.



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.



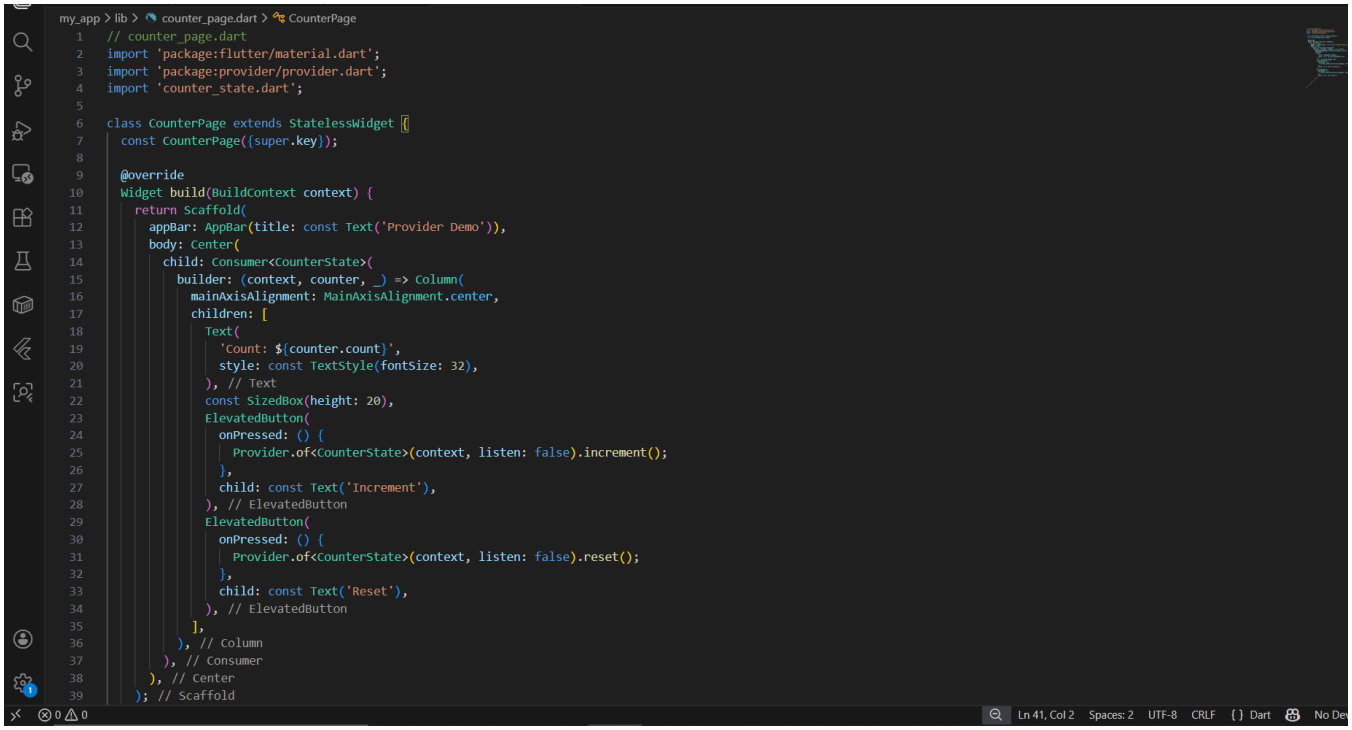
Step 4 - Accessing the State

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.

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.

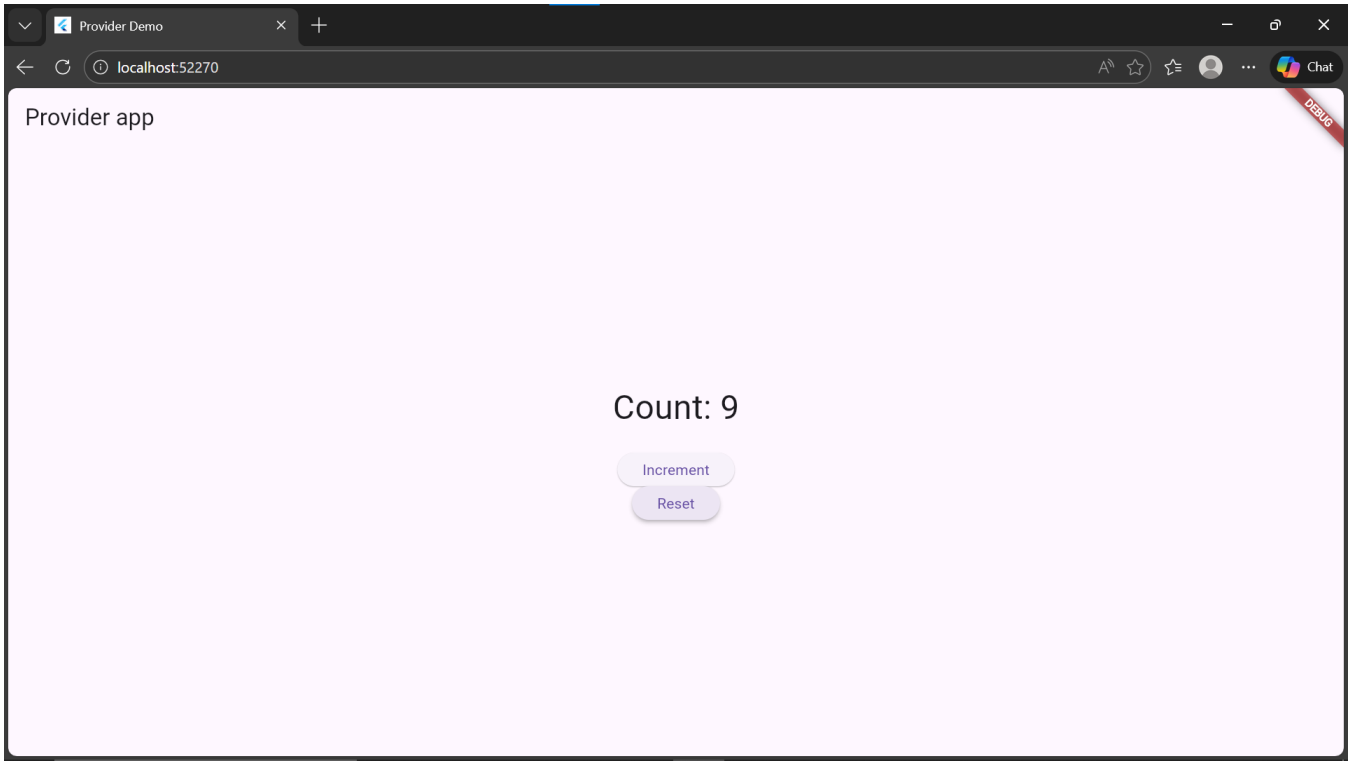


Step 6 - How UI Rebuild Happens

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:

```
User Taps Button
        |
        ▼
increment() called on CounterState
        |
        ▼
_count++  →  notifyListeners()
        |
        ▼
Provider notifies all listening widgets
        |
        ▼
Consumer<CounterState> → builder() called again
        |
        ▼
Text('Count: 1') rendered on screen  ✅
```

Console Output (demonstrating notifyListeners() behavior**):**



Summary - Provider Key Concepts

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

— End of Assignment #1 —