UNIVERSITY OF RWANDA
COMPUTER AND SOFTWARE ENGINEERING
REG NO: 223009343
MOBILE Application System and Design
20th Feb 2026

Assignment

TOPIC : STATE MANAGEMENT IN FLUTTER

1.

- provider is a state management solution built on top of flutter's inheritedWidget. It is officially recommended by the flutter team for simple to medium apps. It works by exposing a state object to the widget tree, and any widget that "listens" to it will automatically rebuild when the states change via notifyListeners().

Strengths:
- Simple to learn and use, especially for beginners
- officially supported and well-documented by the flutter team
- light weight with minimal boilerplate.
- Works well for small to medium-sized applications

Limitations
- As apps grow very large, the number of providers can become hard to manage.
- providers must be placed above the widgets that use them in the tree, which can sometimes lead to deeply nested provider structures.

2. Riverpod : is an improved and more flexible version of provider, created by the same author. It fixes several limitations of provider, such as being compile-time safe, not depending on the widget tree for provider access, and supporting multiple providers of the same type. It is well-suited for medium to large applications.

Strengths
- supports multiple providers of the same type without workarounds
- Completely independent of the widget tree, state can be accessed anywhere
- Scales well from small to large enterprise applications.

Limitations
- Steeper learning curve compared to provider, especially for beginner
- more boilerplate than GetX for simple use cases

3.- <u>Bloc</u> (Business logic Component) Bloc is a pattern that separates business logic from the UI using Streams and Events. The UI sends Events to the Bloc, the Bloc processes them and emits new States, and the UI rebuilds based on those States. It enforces a strict, predictable architecture and is best for large or enterprise-scale applications.

It Built on top of Dart's Streams, which are sequence of asynchronous data.

<u>Strength</u>

- Very strict and predictable architecture (Ideals for large teams)
- Built-in support for async operations via Streams
- Excellent testability: you can test every state transition in isolation

<u>Limitation</u>

- Steeper learning curve compared to provider, Riverpod or GetX
- can feel over-engineered for small or solo projects

4. <u>GetX</u> is light weight, all-in-one flutter package that handles state management, navigation, and dependency injection. It uses reactive variables (Rx types) and controllers to manage state with very little boilerplate. It is popular for fast development but is less strict in architectural enforcement.

GetX is a complete ecosystem that handles three major concerns in one package:

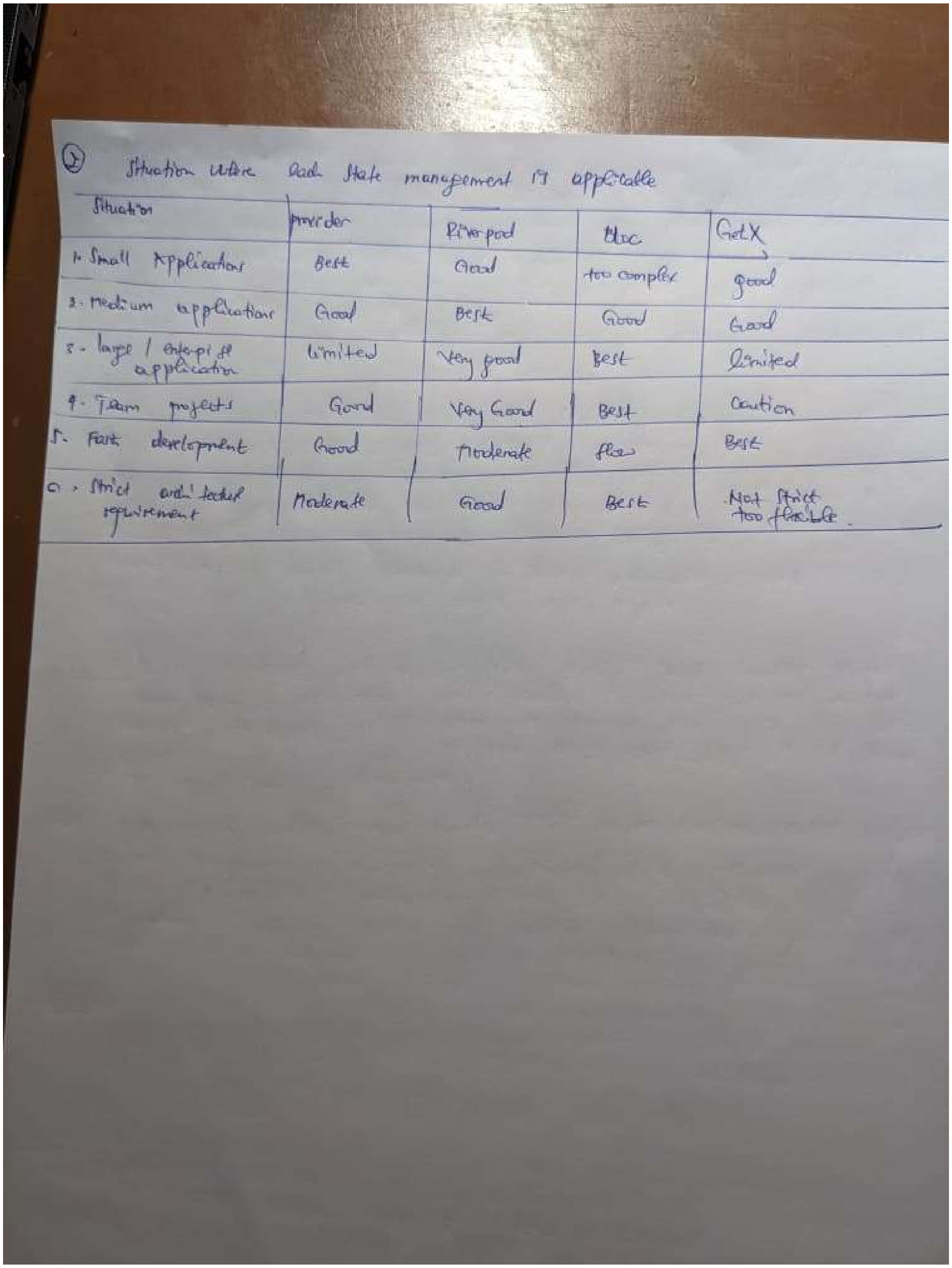
- State Management: managing and reacting to data changes
- Route Management: navigation between screens without BuildContext
- Dependency Injection: creating and injecting dependencies anywhere in the app.

<u>Strengths</u>

- No BuildContext required for most operations
- Gentle learning curve
- Very performant - granular rebuilds with Obx

<u>limitation</u>

- less architecture strictness
- can be harder to test compared to Bloc or Riverpod
- Mixing many responsibilities (state + routing + DI) in one package can cause tight coupling.



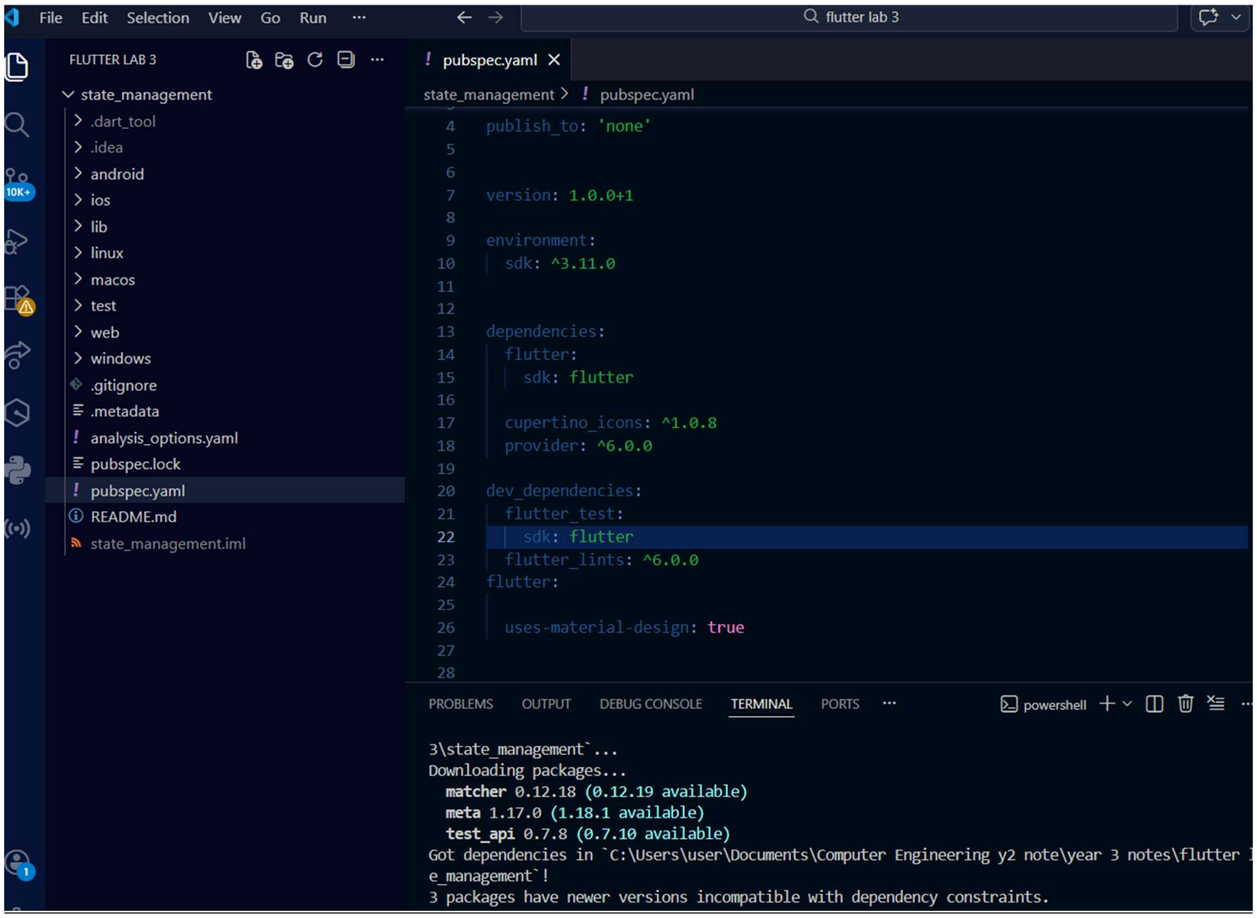
② Situation where each State management is applicable

Situation	provider	Riverpod	Bloc	GetX
1. Small Applications	Best	Good	too complex	good
2. Medium applications	Good	Best	Good	Good
3. large / enterprise application	limited	Very good	Best	limited
4. Team projects	Good	Very Good	Best	Caution
5. Fast development	Good	Moderate	slow	Best
6. Strict architectural requirement	Moderate	Good	Best	Not strict too flexible.

<u>Question 3</u>

1. **<u>Adding dependency</u>**

   The first step is to add the provider package to my project's pubspec.yaml file. This registers the package so Flutter can use it. we run flutter pub get afterward to install it. Without this, none of the Provider classes will be available in my app.

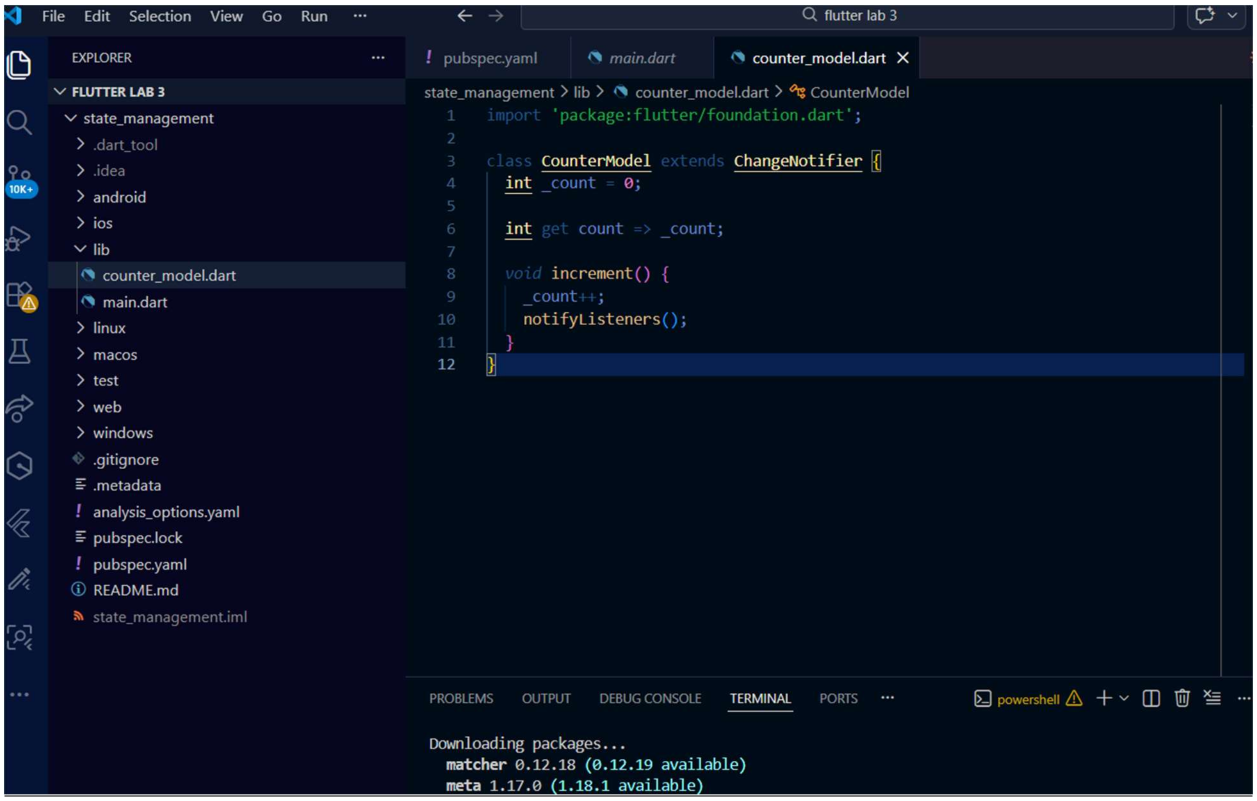


```yaml
publish_to: 'none'


version: 1.0.0+1

environment:
  sdk: ^3.11.0


dependencies:
  flutter:
    sdk: flutter

  cupertino_icons: ^1.0.8
  provider: ^6.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^6.0.0
flutter:

  uses-material-design: true

```

```
3\state_management`...
Downloading packages...
  matcher 0.12.18 (0.12.19 available)
  meta 1.17.0 (1.18.1 available)
  test_api 0.7.8 (0.7.10 available)
Got dependencies in `C:\Users\user\Documents\Computer Engineering y2 note\year 3 notes\flutter l
e_management`!
3 packages have newer versions incompatible with dependency constraints.
```

<u>2.Creating state class</u>

You create a class that extends ChangeNotifier. This class holds your app's state (data) and contains the logic to modify it. ChangeNotifier gives your class the ability to notify listeners when something changes via notifyListeners().



```dart
import 'package:flutter/foundation.dart';

class CounterModel extends ChangeNotifier {
  int _count = 0;

  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}
```

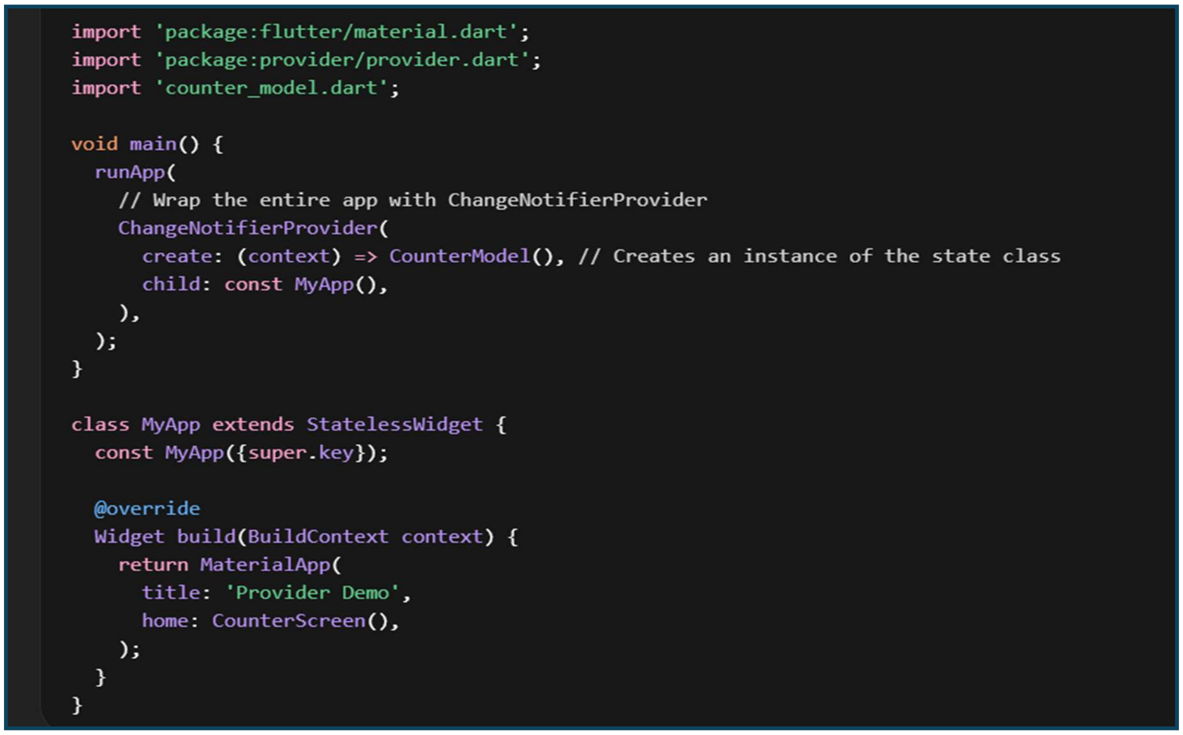
```
Downloading packages...
  matcher 0.12.18 (0.12.19 available)
  meta 1.17.0 (1.18.1 available)
```

3.Providing the state

we wrap our widget tree (usually at or near main()) with ChangeNotifierProvider. This makes the state class available to all widgets below it in the tree. It creates a single instance of my model and injects it into the widget tree.

For application that requires multiple provides ,

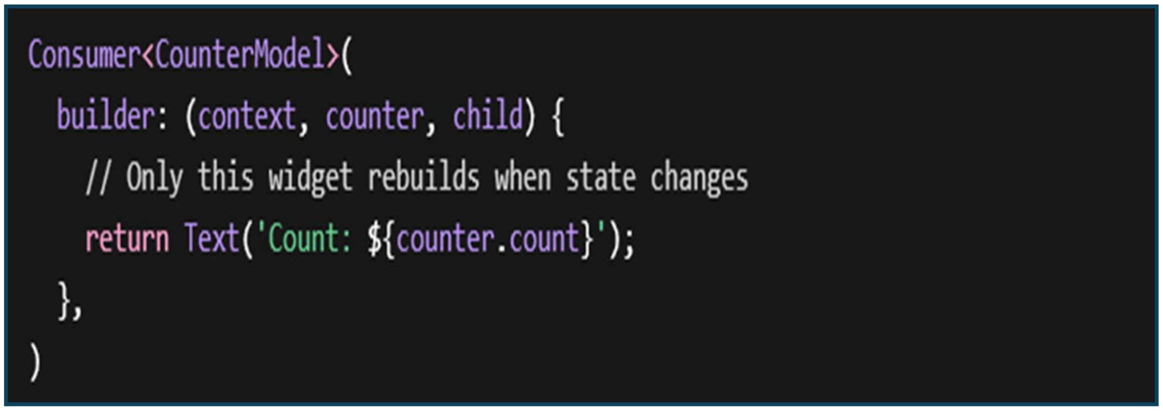
When your app grows, you'll have multiple state classes (e.g., UserModel, CartModel, ThemeModel). Instead of nesting multiple ChangeNotifierProviders (which gets messy), you use MultiProvider — it takes a list of providers and makes all of them available to the entire widget tree below it.



```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'counter_model.dart';

void main() {
  runApp(
    // Wrap the entire app with ChangeNotifierProvider
    ChangeNotifierProvider(
      create: (context) => CounterModel(), // Creates an instance of the state class
      child: const MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Provider Demo',
      home: CounterScreen(),
    );
  }
}
```

4. Accessing the state

Inside a widget, use Consumer, context.watch() or context.read() to get the state.



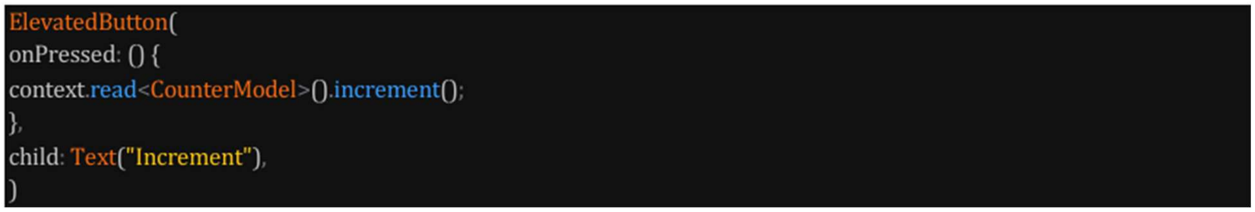
```dart
Consumer<CounterModel>(
  builder: (context, counter, child) {
    // Only this widget rebuilds when state changes
    return Text('Count: ${counter.count}');
  },
)
```

• Consumer listens for changes in CounterModel

• Rebuilds only this widget when state changes

5**. Updating the state**

To call a method that changes state (like incrementing a counter), you use Provider.of<T>(context, listen: false) or context.read<T>(). The listen: false flag is important here — it means this call won't trigger a rebuild by itself; it just performs the action, which then calls notifyListeners() internally to trigger rebuilds elsewhere.



```
ElevatedButton(
onPressed: () {
context.read<CounterModel>().increment();
},
child: Text("Increment"),
)
```

6.<u>**How UI Rebuild Happens in Provider**</u>

Understanding the rebuild mechanism is essential for writing efficient Provider-based apps. The flow works as follows:

1. **User triggers an action** : e.g., taps an 'Increment' button in the UI.
2. **Method is called on the model** : context.read<CounterModel>().increment() is invoked.
3. **State is modified internally** : The private _count variable is incremented inside the model.
4. **notifyListeners() is called** : This method (inherited from ChangeNotifier) broadcasts a change notification to all registered listeners.
5. **Provider intercepts the notification** :ChangeNotifierProvider receives the signal and marks the relevant widgets as "dirty" (needing a rebuild).
6. **Flutter schedules a rebuild** : On the next frame, all widgets using context.watch<T>() or Consumer for this model are rebuilt.
7. **UI updates** : The rebuilt widgets read the new state value and render the updated UI.

Provider is simply a way to **share and manage data** across your Flutter app without passing it manually through every widget.