UNIVERSITY OF RWANDA
COLLEGE OF SCIENCE & TECHNOLOGY
SCHool OF ICT
DEpartMENT OF COMPUTER & SOFTWARE Eng.
Module : MobiLE Application & design.
Reg No : 223007703

TOPIC STATE MANAGEMENT IN FLUTTER.

Q1. explain briefly the following state management used in Flutter application development.

• **provider** : is the officialy recommended state management solution by the flutter team. It is built on top of inherited Widget, making it both simple and efficient.
=> provider works by wrapping parts of The widget tree with a provider widget that holds a state object.
It allows : - storing application state in one place.
- Listening to changes
- Rebuilding only necessary widgets.
It is beginner-friendly and good for small to median projects (Applications)
example: - A counter app
- Theme switching app
- simple E-commerce ~~app~~ cart.

• **Riverpod** : is an improved and more robust evalution of provider, created by the same author (Remi Rousselet). It solves some limitatio of provider like ~~on~~ dependency on Buildcontext.
Advantages: - safer and more flexible.
- work outside widget tree
- Better for testing.
- compile - time safety.
It is suitable for medium to large applications.
Example: - Apps with multiple APIs
- Authentication system
- complex logic.

• **Bloc** (Business Logic component): is a powerfull state management pattern based on The concept of streams. It strictly separates business logic from UI using three main concepts : Events (inputs by user actions), states (output representing The UI condition), and bloc class itself (which processes events and emits new states).

• **Bloc** (Business logic Management): is also more structured state management approach.

It uses: - events
- States
- Streams

Bloc sperates UI From business logic completely.

Advantages: - It very organized
- Easy to test
- Best for large application
- Good for team work.

Example: - Banking app
- Large entreprise system.
- hospital management system.

• **GetX**: is a light weight yet powerfull all-in-one Flutter package that combine state management, dependency injection and route management. It uses reactive programming through Rx Variables and Getx controller. GetX known for its minimal boilerplate code and Fast development speed. unlike Bloc or Riverpod, GetX handles navigation, snack bars, dialogs, and dependency injection in addition to state.

It provides: - State management
- Dependency injection
- Route management.

Advantage: - Very simple syntax
- Fast Development
- less boilerplate code.

It suitable for: - Fast development
- small to medium apps

Example: - startup app
- school project

Q9: Draw table explain the situation where each state management is applicable

Situation	Provider	Riverpod	Bloc	GetX
small applications	Very Good	Good	Too complex	very Good
Medium application	Good	very Good	Good	Good
Large/enterprise App	Limited	very Good	Excellent	Moderate
Team projects	Good	very Good	Excellent	Risk if not structured
Fast development	Good	Moderate	slow	Excellent
strict architecture requirement	limited	Good	Excellent	Not recommended

Question 3: Explain in detail the key steps on how Provider is used.

Provider is Flutter's officially recommended state management solution. It allows widgets to listen to a shared state and automatically rebuild when that state changes. Below are the key steps required to implement Provider in a Flutter application.

Step 1: Adding the Dependency

The first step is to add the provider package to the project's pubspec.yaml file under the dependencies section. After editing the file, run the command flutter pub get in the terminal to download and install the package.

pubspec.yaml

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.0.0
```

Run:

```
flutter pub get
```

Step 2: Creating a State Class:

A state class is a Dart class that holds the data you want to share across the app. It must extend ChangeNotifier, which gives it the ability to notify listeners (widgets) whenever data changes. The class contains the state variables and methods to update them. After updating the state, notifyListeners() is called to trigger UI rebuilds.

Code:

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

class CounterProvider extends ChangeNotifier {
  int _count = 0;

  int get count => _count;

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

Step 3: Providing the State

To make the state available to widgets in the app, the ChangeNotifierProvider widget is placed above the widgets that need access to it in the widget tree typically wrapping the MaterialApp or a specific subtree. The create parameter instantiates the state class.

Code:

```dart
import 'package:provider/provider.dart';

void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CounterProvider(),
      child: MyApp(),
    ),
  );
}
```

If multiple state classes need to be provided simultaneously, MultiProvider can be used to combine them cleanly without nesting multiple ChangeNotifierProviders.

Step 4: Accessing the State

Any descendant widget can read the state using Provider.of(context) or the more concise context.watch() and context.read() methods. context.watch() subscribes the widget to changes (it will rebuild on updates), while context.read() only reads the value once without subscribing (used for actions, not display).

<u>Method 1: Using Provider.of</u>

```dart
final counter = Provider.of<CounterProvider>(context);
Text('${counter.count}');
```

<u>Method 2: Using Consumer</u>

```dart
Consumer<CounterProvider>(
  builder: (context, counter, child) {
    return Text('${counter.count}');
  },
)
```

Step 5: Updating the State

State is updated by calling methods defined in the state class. This is typically done inside button callbacks or other event handlers. context.read() is preferred here because we only need to call the method, not listen for changes.

Call the method inside button:

```
ElevatedButton(
  onPressed: () {
    Provider.of<CounterProvider>(context, listen: false)
        .increment();
  },
  child: Text("Increment"),
)
```

Step 6: How UI Rebuilds Happen

When a method in the state class calls notifyListeners(), Provider internally notifies all widgets that called context.watch() or used Consumer. Flutter then schedules those specific widgets to rebuild during the next frame only the widgets that are listening rebuild, not the entire widget tree. This makes Provider efficient, as it avoids unnecessary rebuilds. The Consumer widget is an alternative approach to context.watch() that allows you to limit rebuilds to a smaller subtree:

```
Consumer<CounterState>(
  builder: (context, counter, child) {
    return Text(
      'Count: ${counter.count}',
    );
  },
)
```

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()* or *Consumer* for this model are rebuilt.
7. **UI updates:** The rebuilt widgets read the new state value and render the updated UI.

End.................