UNIVERSITY OF RWANDA

COLLEGE OF SCIENCE AND TECHNOLOGY
SCHOOL OF CCT
DEPARTMENT OF COMPUTER AND SOFTWARE ENGINEERING

ACADEMIC YEAR: 2025-2026

YEAR OF STUDY: 3

DATE: On Friday, February 27th, 2026

STUDENT DETAILS: Silas HAKUZWIMANA (223001019)
MODULE: MOBILE APPLICATION SYSTEMS AND DESIGN
ASSIGNMENT 1: STATE MANAGEMENT IN FLUTTER

Q1.

* Provider:
  - Currently the most popular and Google-recommended state management solution.
  - It is a wrapper around inheritedWidget to make them easier to use and more reusable.
  - It listens to a value and exposes it to the widget tree. When the value changes, all the widgets listening to it are built.
  - Best for: lifting state up and dependency injection.

* Riverpod:
  - Created by the same author as Provider (Remi Rousselet) to address some provider's limitations.
  - It is compile-safe (no run-time errors from forgetting to provide a value) and doesn't rely on Flutter's BuildContext.
  - It allows for consuming multiple providers of the same type without conflict.
  - Best for: Applications requiring more safety and complex provider hierarchies.

* Bloc (business logic component):
  - A design pattern that separates presentation from business logic.
  - Relies on Events and States. The UI sends Events (like buttonClick) and the Bloc emits new States (like LoadingState, LoadedState) in response.
  - Uses streams (Reactive programming). It is very predictable and testable.
  - Best for: large streams and complex applications where a strict, testable architecture is required.

*GetX:

- A micro-framework that is extremely lightweight and high performance
- Combines statemanagement, dependency injection, and route management in one package.
- Uses "Reactive State Manager" (GetBuilder/Obx) which updates only the specific part of UI that uses the variable.
- Known for its simplicity, minimal boilerplatecode, and zero context navigation.
- Best for: Fast development and small to medium apps where you want to minimize coding time.

Q2.

Situation	Recommended State management	Explanation
Small applications	Provider or GetX	Simple apps don't need complex streams. Provider is straightforward and GetX is very fast to implement
Medium applications	provider or Riverpod	As complexity grows, you need better organisation. Riverpod offers type safety, while Provider remains simple.
Large/Enterprise applications	Bloc or Riverpod	Enterprise apps need strict architecture, scalability, and testability. Bloc's event-driven pattern excels here.
Team projects	Bloc or Riverpod	clear structure prevents messy code. Bloc enforces a specific way of doing things making it easy for team members to understand each other's code.
Fast development	GetX	It has the least boilerplate. You can build an app in days with minimal code for navigation and state management.
Strict architecture requirement	Bloc	Bloc forces the separation of UI and Business logic completely. It follows the official Business logic component pattern from Google.

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

Detailed explanation of Provider usage

Provider is a wrapper around InheritedWidget to manage state efficiently. Here are the key steps to implement it:

1. Adding Dependency

First, you must include the Provider package in your project's configuration file.

- Open your pubspec.yaml file.
- Add provider under the dependencies section:

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

- Run flutter pub get in your terminal to install the package.

2. Creating a State Class (Model)

You need a class that holds your data and logic. This class must extend ChangeNotifier to allow it to broadcast updates.

- Define your variables and methods.
- Call notifyListeners() whenever the state changes to alert the UI.

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

class Counter with ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count++;
    notifyListeners(); // Essential for triggering UI updates
  }
}
```

3. Providing the State

To make the state accessible, you must wrap your widget tree with a ChangeNotifierProvider.

- This is typically done at the top level in main.dart.
- The provider must be placed above any widget that needs to consume the data.

```
// main.dart
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => Counter(), // Creates the Counter instance
      child: const MyApp(),
    ),
  );
}
```

4. Accessing the State

There are two primary ways to retrieve data from your provider:

- ✓ context.watch<T>(): This is used inside the build method to make the widget listen to changes in the state. When notifyListeners() is called, this widget will rebuild.
- ✓ context.read<T>(): This is used to access the state without listening for changes. It is ideal for one-time actions, such as calling an increment function inside an onPressed callback, because it avoids unnecessary UI rebuilds.
- ✓ Provider.of<T>(context): This is the classic approach. By default, it acts like watch (rebuilds the UI), but you can set listen: false to make it behave like read.
- ✓ Consumer<T>: This is a widget wrapper used for more granular rebuilds. It ensures that only the specific part of the widget tree inside the Consumer is rebuilt when the state updates, which helps optimize app performance.

5. Updating the State

State is updated by calling the methods defined in your ChangeNotifier class.

- When calling a method (like an increment function), use listen: false within Provider.of because you are performing an action rather than building a widget that needs to watch for changes.

```
// Example inside an ElevatedButton
onPressed: () {
  Provider.of<Counter>(context, listen: false).increment(); [cite: 20]
}
```

6. How UI Rebuild Happens

The rebuild process follows a specific reactive flow:

1. **Trigger:** A user interaction (like a button tap) calls a method in the state class.
2. **Logic:** The method updates the data and executes notifyListeners().
3. **Notification:** The ChangeNotifier sends a signal to all active listeners.
4. **Reaction:** ChangeNotifierProvider catches this signal.
5. **Rebuild:** Provider identifies widgets using Provider.of(listen: true) or Consumer and triggers their build() methods.
6. **UI Update:** Flutter renders the screen with the updated values.