Assignment #1

State Management in Flutter

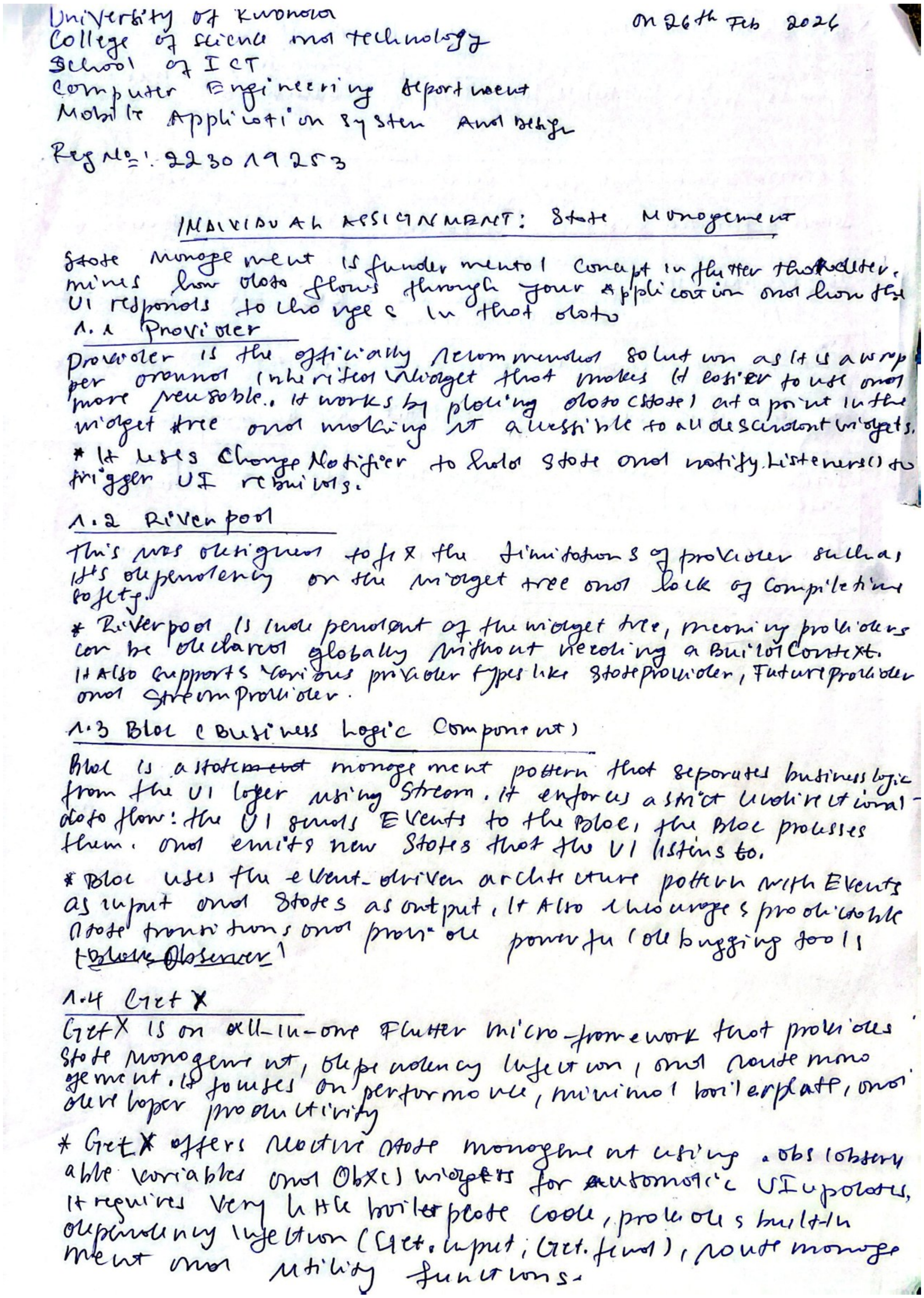
Year 3 CSE

Student Name: MBABAZI Patrick Straton

Student ID:223019253

Date: on 27 Feb 2026

Question 1: Explain State Management Approaches



University of Kibungo / Rwanda   On 26th Feb 2026
College of science and technology
School of ICT
Computer Engineering department
Mobile Application System And Design

Reg No: 223019253

INDIVIDUAL ASSIGNMENT: State Management

State management is fundamental concept in flutter that determines how data flows through your application and how the UI responds to changes in that data

1.1 Provider

provider is the officially recommended solution as it is a wrapper around inherited widget that makes it easier to use and more reusable. It works by placing data (state) at a point in the widget tree and making it accessible to all descendant widgets.

* It uses ChangeNotifier to hold state and notifyListeners() to trigger UI rebuilds.

1.2 Riverpod

This was designed to fix the limitations of provider such as it's dependency on the widget tree and lack of compile-time safety.

* Riverpod is independent of the widget tree, meaning providers can be declared globally without needing a BuildContext. It Also supports various provider types like StateProvider, FutureProvider and StreamProvider.

1.3 Bloc (Business Logic Component)

Bloc is a ~~state management~~ management pattern that separates business logic from the UI layer using Stream. It enforces a strict unidirectional data flow: the UI sends Events to the Bloc, the Bloc processes them and emits new States that the UI listens to.

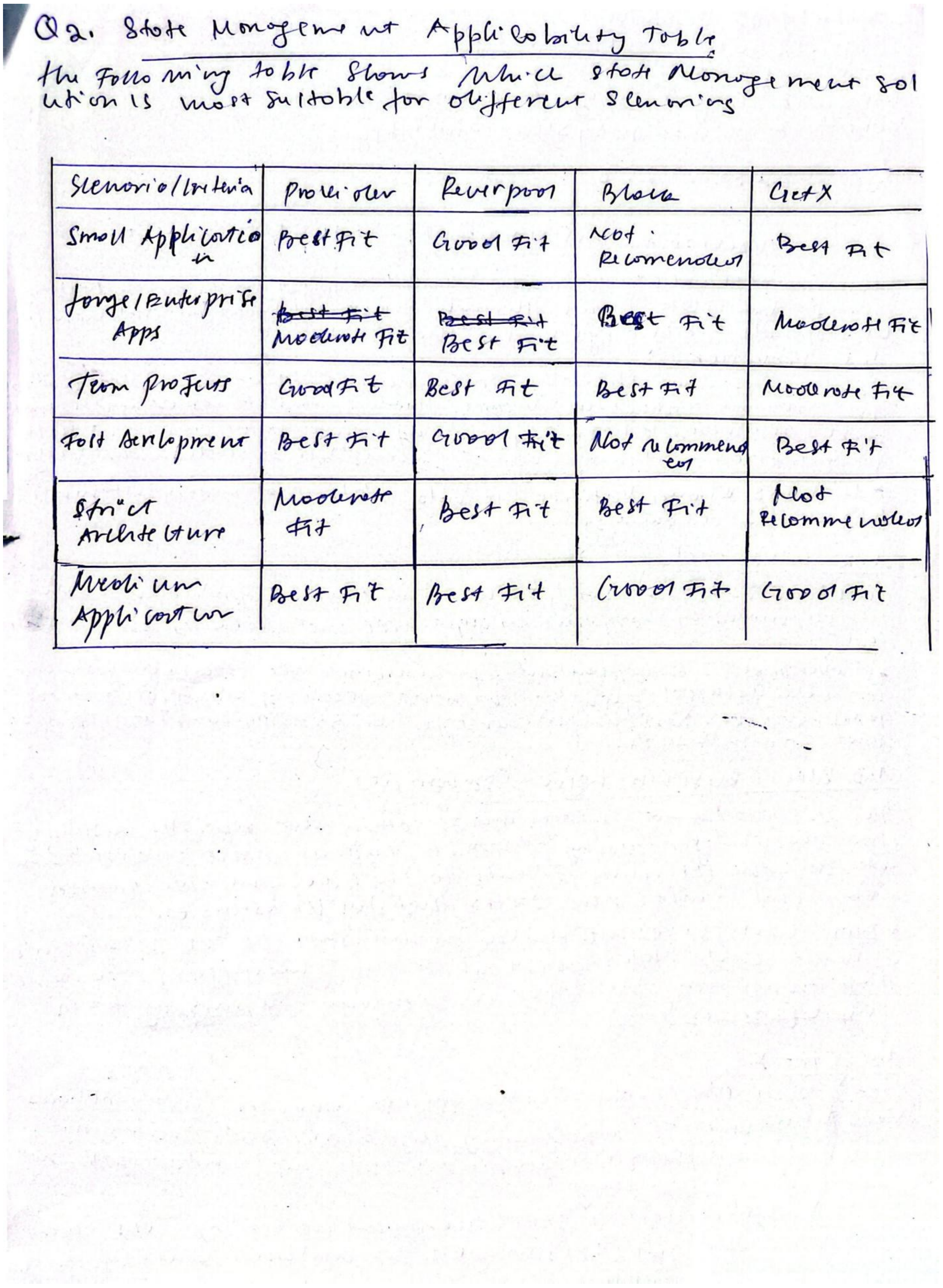
* Bloc uses the event-driven architecture pattern with Events as input and States as output, It Also encourages predictable state transitions and provide powerful debugging tools (~~BlocObserver~~)

1.4 GetX

GetX is an all-in-one Flutter micro-framework that provides state management, dependency injection, and route management. It focuses on performance, minimal boilerplate, and developer productivity

* GetX offers reactive state management using .obs (observable variables and Obx() widgets for automatic UI updates, It requires very little boilerplate code, provides built-in dependency injection (Get.put, Get.find), route management and utility functions.

Question 2: State Management Applicability Table



Q2. State Management Applicability Table

the Following table shows which state Management sol ution is most suitable for different Scenarios

Scenario/Criteria	Provider	Riverpod	Bloc	GetX
Small Application	Best Fit	Good Fit	Not Recommended	Best Fit
Large/Enterprise Apps	~~Best Fit~~ Moderate Fit	~~Best Fit~~ Best Fit	Best Fit	Moderate Fit
Team Projects	Good Fit	Best Fit	Best Fit	Moderate Fit
Fast Development	Best Fit	Good Fit	Not recommended	Best Fit
Strict Architecture	Moderate Fit	Best Fit	Best Fit	Not Recommended
Medium Application	Best Fit	Best Fit	Good Fit	Good Fit

Question 3: How Provider is Used (Detailed Steps)

Step 1: Adding the Dependency

First, add the provider package to your Flutter project by including it in the pubspec.yaml file under dependencies:

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.1.1
```

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

```
flutter pub get
```

This downloads the Provider package and makes it available in your project. Provider is maintained by the Flutter community and is the officially recommended approach by Google's Flutter team.

Step 2: Creating a State Class

Create a class that extends ChangeNotifier. This class will hold your application state and provide methods to modify it. ChangeNotifier is a built-in Flutter class that provides change notification to its listeners.

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

class CounterState extends ChangeNotifier {
  int _count = 0;

  int get count => _count;

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

  void decrement() {
    _count--;
    notifyListeners();
  }

  void reset() {
    _count = 0;
    notifyListeners();
  }
}
```

The underscore prefix (_count) makes the variable private. The getter (count) exposes it as read-only. Every method that changes the state calls notifyListeners() to tell all listening widgets to rebuild.

Step 3: Providing the State

Wrap the part of the widget tree that needs access to the state with a ChangeNotifierProvider. This is typically done at the top level of your app so the state is available everywhere:

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


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

ChangeNotifierProvider creates an instance of CounterState and makes it available to all descendant widgets. For multiple providers, use MultiProvider:

```
MultiProvider(
  providers: [
    ChangeNotifierProvider(create: (_) => CounterState()),
    ChangeNotifierProvider(create: (_) => ThemeState()),
  ],
  child: const MyApp(),
)
```

Step 4: Accessing the State

There are three main ways to access the provided state in your widgets:

Method 1 - context.watch<T>(): Listens for changes and rebuilds the widget when the state updates.

```
Widget build(BuildContext context) {
  final counter = context.watch<CounterState>();
  return Text('Count: ${counter.count}');
}
```

Method 2 - context.read<T>(): Accesses the state without listening. Use this in callbacks like onPressed where you do not need to rebuild.

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

Method 3 - Consumer<T> widget: Provides more granular control by rebuilding only the widget inside the Consumer builder.

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

Step 5: Updating the State

To update the state, call the methods defined in your state class. This is typically done from event handlers like button presses:

```
// Using context.read (recommended for callbacks)
FloatingActionButton(
  onPressed: () {
    context.read<CounterState>().increment();
  },
  child: Icon(Icons.add),
)
```

Important: Always use context.read() (not context.watch()) inside callbacks and event handlers. Using context.watch() in these places would cause unnecessary rebuilds and potential errors.

Step 6: How UI Rebuild Happens

The UI rebuild process in Provider follows these steps:

1. State Change: A method in the ChangeNotifier class is called (e.g., increment()), which modifies the internal state variable.

2. Notification: The method calls notifyListeners(), which sends a signal to all registered listeners that the state has changed.

3. Widget Detection: Flutter's framework checks which widgets are listening to this specific ChangeNotifier (those using context.watch() or Consumer).

4. Selective Rebuild: Only the widgets that depend on the changed state are rebuilt. Other widgets in the tree remain untouched, ensuring optimal performance.

5. UI Update: The rebuilt widgets reflect the new state values on screen.

This selective rebuild mechanism is what makes Provider efficient. Instead of rebuilding the entire widget tree, only the affected widgets are updated. Using Consumer widgets or context.select() can further optimize this by narrowing down exactly which parts of the UI need to respond to specific state changes.

```
// Example: context.select rebuilds ONLY when 'count' changes
final count = context.select<CounterState, int>(
  (counter) => counter.count,
);
```