Reg No: 2231125 61 Le 27/02/2026

Computer Engineering Y3

Question 1 State management explained briefly

* Provider : this is a simple and light weight state management solution built on top of Inherited widget. It allows data sharing and it uses changeNotifier to notify listeners when data changes.

* Riverpod : this is a state management that is more flexible compared to provider, it doesn't depend on the widget tree. it is more safe and more testable. Moreover it supports compile time error checking and better dependency management

* Bloc (Business Logic component): It is based on reactive programming using streams. It separate business logic from UI completely. Moreover it uses Event (inputs) and States (outputs). It is highly structured and predictable

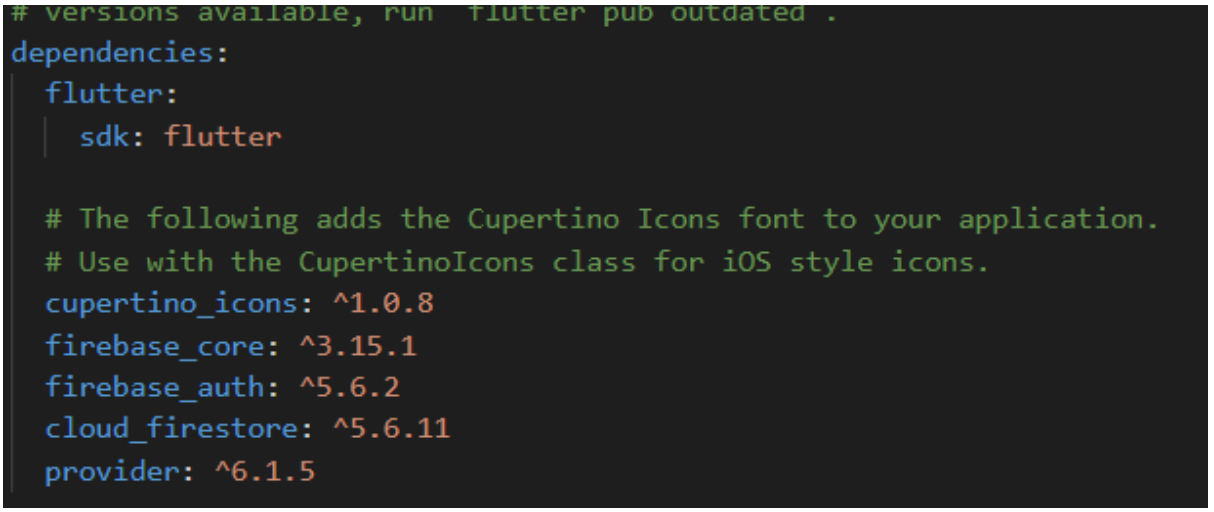
* Get x : This is a light weight state management solution that combines state management, dependency injection and routing. it has very simple syntax and it is very fast to implement though some developers argue it can encourage poor architectural habits if not used carefully due to it's "magic" approach

Scenario	Provider	River Pod	Bloc	Getx
Small Applications	Best (simple and quick)	work well	Overkill	Work well
Medium Applications	Good fit	Best choice	Good fit	Good fit
Large Enterprise Apps	Can struggle	Good fit	Best Choice	Not recommended
Team Projects	Manageable	Good fit	Best Choice	Can cause inconsistency
Fast development	moderate speed	Moderate speed	slow setup	Best Choice
Strict Architecture requirement	loosely enforced	well structured	Best choice	Too flexible/ loose

Question 3

How a provider is used in flutter you pass through these steps:

1. Add the provider with flutter pub add provider which Automatically adds the latest version of provider to your pubspec.yaml file. Runs flutter pub get afterward to install it



```
# versions available, run  flutter pub outdated .
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.8
  firebase_core: ^3.15.1
  firebase_auth: ^5.6.2
  cloud_firestore: ^5.6.11
  provider: ^6.1.5
```

Every dependency is placed under dependencies in public spec.yaml. And with flutter pub get you get the all dependencies.

2. **Creating a State Class :** You create a class that extends ChangeNotifier. This class holds the state and contains methods to update it. When the state changes, you call notifyListeners() to notify all listeners.

```dart
import 'package:note_system/repositories/notes_repository.dart';

class NotesProvider with ChangeNotifier {
  final NotesRepository _repository = NotesRepository();
  List<Note> _notes = [];
  bool _isLoading = false;
  String? _error;

  List<Note> get notes => _notes;
  bool get isLoading => _isLoading;
  String? get error => _error;

  Future<void> fetchNotes(String userId) async {
    _isLoading = true;
    notifyListeners();

    try {
      _notes = await _repository.fetchNotes(userId);
      _error = null;
    } catch (e) {
      _error = 'Failed to fetch notes';
    } finally {
      _isLoading = false;
      notifyListeners();
    }
  }

  Future<void> addNote(String userId, String text) async {
    try {
      await _repository.addNote(userId, text);
      await fetchNotes(userId); // Refresh the list
    } catch (e) {
      _error = 'Failed to add note';
      notifyListeners();
    }
  }
```

3. **Providing the State**

   You provide the state object at the top of the widget tree using ChangeNotifierProvider. This makes the state available to all descendant widgets. You do this in main

It might be one provider or multiple providers but in that app i developed there were multiple providers

Syntax for one provider:

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

```
Widget build(BuildContext context) {
  return MultiProvider(
    providers: [
      ChangeNotifierProvider(create: (_) => NotesProvider()),
      ChangeNotifierProvider(create: (_) => AuthenticationProvider()),
    ],
```

4. Accessing the State

There are several ways to access the state in a widget:

Provider.of<T>(context) – gives the object and listens to changes if listen: true (default).

Consumer<T> – rebuilds only the part inside it when the state changes.

context.watch<T>() – listens to changes.

context.read<T>() – gets the object without listening.

Example of Provide.of and consumer

```
Future<void> _loadNotes() async {
  final notesProvider = Provider.of<NotesProvider>(context, listen: false);
  await notesProvider.fetchNotes(_user.uid);
  if (!mounted) return;

  final error = notesProvider.error;
```

```
      ),
      body: Consumer<NotesProvider>(
        builder: (context, provider, child) {
          if (provider.isLoading && provider.notes.isEmpty) {
            return const Center(child: CircularProgressIndicator());
          }

          if (provider.notes.isEmpty) {
            return const Center(
              child: Text('Nothing here yet—tap + to add a note.'),
            );
          }

          return ListView.builder(
            padding: const EdgeInsets.all(8),
            itemCount: provider.notes.length,
```

4. **Updating the State**

   You update the state by calling methods on the provider object. This is usually done in response to user interaction.

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

Or using context.read:

```
onPressed: () => context.read<Counter>().increment(),
```

5. **How UI Rebuild Happens**

   When notifyListeners() is called in the ChangeNotifier, Provider automatically triggers a rebuild of any widgets that are listening to that provider.

   Widgets that use Consumer, context.watch(), or Provider.of with listen: true will rebuild.

   This ensures only the parts of the UI that depend on the changed state are updated, improving performance.