Reg no: 92300678
Computer Engineering
Mobile Applications Systems and Design
27/09/2026.

Q1. State management used in flutter application development.

① provider: is a simple and widely used state management solution in flutter. It helps share data between different widgets in an application without passing the data manually through many widgets levels.

- How it works:
  provider store the state in a separate class and allows widgets to listen for changes. when the data changes, the widgets that depend on it automatically rebuild. (data is provided from a higher level in the widget tree and accessed by lower widgets.

- Key features
  - * Reduces unnecessary code
  - * Efficient UI updates
  - * Recommended for small to medium applications.

- Use cases example
  - * Managing login information
  - * Theme switching
  - * Sharing app settings across screens.

```
import 'package:flutter/material.dart';
class Counter with ChangeNotifier {
  int _count = 0; // Variable to store
  void increment() {
    _count++; // Increase the store
    notifyListeners(); // Refreshing UI
  }
}
Consumer<Counter>(
builder: (context, counter, child) {
  return Column(
    children: [
      Text('${counter.count}', style: ...
      ElevatedButton(
        onPressed: counter.increment,
        child: Text("Add"),
      ),
    ],
  );
},
);
```

Q2. Riverpod

Riverpod is an improved and more advanced version of provider. It was created to solve some limitations of provider and make state management safer and more flexible.

- How it works:
  Riverpod manages state outside the widget tree allows developers to access data from anywhere in the application.
  (State is stored in providers that can be accessed globally in a controlled and safe way).

- Key features
  - more reliable and scalable
  - Compile-time safety
  - works well with large applications.
  - Easier testing and debugging

- Example use cases
  - Applications with complex data flow
  - Apps that require strong architecture
  - large flutter projects.

Q1.3. Bloc (Business logic component)

Bloc is a structured state management pattern used to separate business logic from the user interface.

- How it works:
  Bloc uses events and states
  + The user performs an action (event)
  + The Bloc processes the event
  + A new state is produced and sent to the UI
  (UI and business logic are completely separated).
- Flow of Bloc
  User Action → Event → Bloc → New state → UI update
- Key features
  - Clear separation of responsibilities
  - Scalable for large applications
  - Predictable data flow
  - Good for team projects.
- Use cases example
  - Enterprise flutter applications
  - Apps with complex business logic
  - Applications with many features.

Q1.4. GetX

GetX is a fast and lightweight framework used for state management, navigation, and dependency injection.

- How it works
  GetX uses reactive programming, where the UI automatically updates whenever the state changes.
- Main idea
  State changes are observed automatically, and the interface without needing much code.
- Key features
  - Very simple and fast
  - Requires less code
  - High performance
  - Combines multiple features in one package
- Use cases example
  - Quick application development
  - Medium to large apps
  - projects that need simple and fast state updates.

Q9.

Situation	provider	Riverpod	Bloc	Provider
Small applications	Very suitable and easy to implement	Suitable but sometimes more than needed	Not usually necessary	Very suitable and quick
Medium applications.	Good and commonly used	Very good and scalable	Suitable	Very good
large Enterprise applications	Can work but may became complex	Highly recommended	Best choice	Can work but depends on architecture
Team projects	Good but needs organization	Very good for teamwork	Excellent because of clear structure	Sometimes harder to manage in large teams.
Faster development	Good	Good	Slower because it requires more setup	Best option for fast development
Strict architecture requirement	limited architecture control	Strong architecture support	Best for strict and structured architecture	less strict archi

Question 3.

1. Adding Dependency

First, you need to add the Provider package to your Flutter project.

Open **pubspec.yaml** and add:

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

Then run:

```
flutter pub get
```

This installs Provider so it can be used in the project.

2. Creating a State Class

Next, create a class that will store and manage the state of the application.

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

// This class manages the state
class CounterProvider extends ChangeNotifier {
 int count = 0; // State variable

 // Function to update the state
 void increment() {
  count++;
  notifyListeners(); // Notify widgets to rebuild
 }
}
```

Explanation:

- ChangeNotifier allows the class to notify the UI when data changes.
- count stores the state.
- increment() updates the state.

- notifyListeners() tells Flutter to refresh widgets using this state.

3. Providing the State

Now the state must be made available to the application using **ChangeNotifierProvider**.

Usually this is done in main.dart.

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

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

Explanation:

- ChangeNotifierProvider provides the state to the widget tree.
- Any widget inside MyApp can access this state.

4. Accessing the State

Widgets can read the state using **Consumer** or **Provider.of**.

Example using **Consumer**:

```
Consumer<CounterProvider>(
  builder: (context, counter, child) {
    return Text(
      counter.count.toString(),
      style: TextStyle(fontSize: 24),
    );
  },
);
```

Explanation:

- Consumer listens to changes in CounterProvider.

- When the state changes, this widget rebuilds.

5. Updating the State

The state is updated by calling a method from the provider class.

Example with a button:

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

Explanation:

- The button calls increment().
- The state value changes.
- notifyListeners() is triggered.

6. How UI Rebuild Happens

UI rebuild happens automatically when the state changes.

Process:

1. User presses a button.
2. Provider function updates the state.
3. notifyListeners() is called.
4. Widgets listening to the provider rebuild.
5. Updated data appears on the screen.

Example showing the full UI part:

```
Column(
 mainAxisAlignment: MainAxisAlignment.center,
 children: [
  Consumer<CounterProvider>(
   builder: (context, counter, child) {
```

```
        return Text("Count: ${counter.count}");
      },
    ),
    ElevatedButton(
      onPressed: () {
        Provider.of<CounterProvider>(context, listen: false).increment();
      },
      child: Text("Add"),
    ),
  ],
);
```