Flutter cheat Sheet

State Management

State, in the context of a Flutter app, represents the information that can change over time and influences the appearance and behavior of the UI. Examples of state in a Flutter app include user input, network data, device orientation, and more. Effective state management involves handling and updating this data to keep the UI in sync with the application's logic.
In Flutter, the UI is typically divided into widgets, and state management revolves around how data is shared and updated between these widgets. Good state management ensures that when a change occurs in one part of the app, it accurately reflects in the other relevant parts of the UI

Types of State Management in Flutter

Flutter offers a range of state management solutions to cater to various project complexities and developer preferences. Here are some common types of state management in Flutter:

- **Provider State Management**
- **Bloc State Management**
- **Cubit State Management**
- **GetX State Management**

Flutter cheat Sheet

State Management

- **Provider Package :-**

A well-liked state management package for Flutter is Provider. It allows you to create and manage providers that hold the application's state. Widgets can then listen to these providers for changes in the state. Provider is particularly useful for managing global or shared state.

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

class MyWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
 final counter = Provider.of<Counter>(context);

 return Column(
 children: <Widget>[
 Text('Counter: ${counter.value}'),
 ElevatedButton(
 onPressed: () => counter.increment(),
 child: Text('Increment'),
 ), ], ); } }

class Counter with ChangeNotifier {
 int _value = 0;
 int get value => _value;

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

State Management

- **Cubit State Management :-**

Cubit is a state management solution built specifically for Flutter applications. It follows the principles of the BLoC (Business Logic Component) pattern and combines them with simplified semantics to provide a streamlined approach to state management.

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

class CounterCubit extends Cubit<int> {
 CounterCubit() : super(0);

 void increment() => emit(state + 1);
 void decrement() => emit(state - 1); }


final counterCubit = BlocProvider.of<CounterCubit>(context);

Text( 'Counter Value:',  style: TextStyle(fontSize: 24),  ),
BlocBuilder<CounterCubit, int>(
builder: (context, count) {
return Text(  '$count', style: TextStyle(fontSize: 48), );  },  ),
Row(
mainAxisAlignment: MainAxisAlignment.center,
children: [ FloatingActionButton( onPressed: () {
counterCubit.increment(); },
child: Icon(Icons.add),  ),
FloatingActionButton( onPressed: () {
counterCubit.decrement();  }, child: Icon(Icons.remove), ), ], )
```

State Management

- **Bloc State Management:-**

The Bloc (Business Logic Component) pattern is commonly used for state management in Flutter, especially for complex applications. It separates the UI from the business logic and uses streams to manage and propagate state changes. The `flutter_bloc` package is widely adopted for implementing the Bloc pattern.

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

class CounterCubit extends Cubit<int> {
 CounterCubit() : super(0);

 void increment() => emit(state + 1);  }

class MyWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
 final counterCubit = context.read<CounterCubit>();

 return Column(  children: <Widget>[
 BlocBuilder<CounterCubit, int>(
 builder: (context, state) {  return Text('Counter: $state');
 },  ),
 ElevatedButton(
 onPressed: () => counterCubit.increment(),
 child: Text('Increment'),  ),  ],  );  }
}
```

- **GetX State Management:-**

GetX is a powerful and lightweight state management library for Flutter. It provides a simplified and high-performance way to manage state, handle navigation, and inject dependencies. GetX is known for its simplicity and efficiency.

```
class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
 return GetMaterialApp( title: 'Flutter Demo',
 theme: ThemeData(
  primarySwatch: Colors.blue, ),
 home: First(),
 debugShowCheckedModeBanner: false, ); } }

class First extends StatelessWidget {
@override
Widget build(BuildContext context) {
 return Scaffold( appBar: AppBar(  title: Text("WsCubeTech"),
  centerTitle: true,  backgroundColor: Colors.green, ),
 body: Center(  child: Container(  child: ElevatedButton(
  child: Text("Go to next screen"),   onPressed: () {
  Get.to(Second());    }  ),  ), ), ); } }
class Second extends StatelessWidget {
@override
Widget build(BuildContext context) {
 return Scaffold( appBar: AppBar(
  title: Text("WsCubeTech"),  centerTitle: true,
  backgroundColor: Colors.green, ),
 body: Center( child: Container(
  child: ElevatedButton(   child: Text("Go to first screen"),
   onPressed: null  ),  ), ), ); } }
```

Flutter cheat Sheet

State Management

- **MultiProvider :-**

The MultiProvider uses providers property to take a list of Providers and inject them into widget-tree at the same level, so that all these providers will be available i.e. can expose their values (data) to all the widgets below the level, where the MultiProvider is placed; in the widget-tree.

```
class AppHomePage extends StatelessWidget {
 const AppHomePage({Key? key}) : super(key: key);  @override
 Widget build(BuildContext context) {
      // providers placed here with the help of MultiProvider
return MultiProvider(
      // providers property takes a list of provider,
     // all these provers are available to widget HomeBody
     // and below widgets in th widget tree       providers: [
    // first provider
    Provider<UserDetails>(
       create: (_) => UserDetails(  userName: "andy1234",
          name: "Andy Rousslet", themeColor: "Blue",
          roleId: 5)),
    //second provider
    Provider<UserLastLoginDetails>(
       create: (_) => UserLastLoginDetails(
          userName: "andy1234",
          lastLoginDate: "Mar 10, 2023",
          lastLoginTime: "10:30 AM"),   ),   ],
     child: const HomeBody());  } }
```

State Management

- **Consumer :-**

The provider is using state management in Flutter, Consumer is playing a specific role. Whenever any value update in ChangeNotifier class or we can whenever notifyListeners() is called in ChangeNotifier class then we get the updated value from Consumer block on UI screen. We can call it a listener which listens to updated values

```
class Box1 extends StatelessWidget {
        static int _rebuildCount = 0;
 @override Widget build(BuildContext context) {
            _rebuildCount++;
return Consumer<Notifier<int>>(
                builder: (context, count, child) =>
                Container( height: 100, width: 150, child:
Center(child:Text('Rebuild:
$_rebuildCount\n''${count.value.toString()}'),
), decoration: BoxDecoration(
border: Border.all( color: Colors.blue, width: 5, ),
                                    ), ),
                                        );
                                            }
                                                }
```

Flutter cheat Sheet

State Management

- **MultiBlocProvider, BlocBuilder, BlocListener, BlocConsumer, Manging States :-**

In Flutter, managing states efficiently is crucial for building robust and maintainable applications. The BLoC (Business Logic Component) pattern is a popular state management solution that helps to separate business logic from UI components. The `bloc` package in Flutter provides various tools to implement the BLoC pattern effectively. Here's how you can use `MultiBlocProvider`, `BlocBuilder`, `BlocListener`, and `BlocConsumer` to manage states in Flutter:

1. **MultiBlocProvider**: `MultiBlocProvider` allows you to provide multiple BLoCs at the root of your widget tree. This is useful when you have multiple BLoCs in your application.

```dart
MultiBlocProvider(
providers: [
BlocProvider<BlocA>(create: (_) => BlocA()),
BlocProvider<BlocB>(create: (_) => BlocB()),
// Add more BlocProviders as needed
],
child: MyApp(),
)
```

State Management

2. **BlocBuilder**: `BlocBuilder` is a widget that listens to the state of a BLoC and rebuilds its child when the state changes.

```dart
BlocBuilder<BlocA, BlocAState>(
builder: (context, state) {
// Rebuild UI based on BlocAState
return YourWidget();
},
)
```

3. **BlocListener**: `BlocListener` is a widget that listens to state changes in a BLoC but does not rebuild its child widget. It's useful for reacting to state changes without rebuilding the entire UI

```dart
BlocListener<BlocA, BlocAState>(
listener: (context, state) {
// React to state changes, e.g., show a SnackBar
Scaffold.of(context).showSnackBar(SnackBar(content
: Text('State changed'))); },
child: YourWidget(), )
```

State Management

4. **BlocConsumer**: `BlocConsumer` is a combination of `BlocBuilder` and `BlocListener`. It allows you to both listen to state changes and rebuild UI based on the new state.

```dart
BlocConsumer<BlocA, BlocAState>(
listener: (context, state) {
// React to state changes
},
builder: (context, state) {
// Rebuild UI based on state
return YourWidget(); }, )
```

By using these widgets along with the BLoC pattern, you can effectively manage the state of your Flutter applications, ensuring separation of concerns and maintainability.