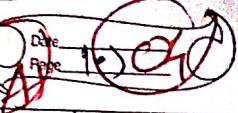


Name - Haran Krishna Div - D15B

Roll No - 33 MPL Assignment



Q1)

Ans) Key Features and Advantages of Flutter for Mobile App Development.

Single codebase for Multiple Platforms
One code to run Web, iOS, Android, etc.

• Hot Reload: Instantly see changes in app without losing state, speeding up development and debugging.

• Rich UI with customizable Widgets: Provides a wide range of pre-designed widgets and allows full customization to build complex, responsive UIs.

• High Performance: Built on Dart, Flutter compiles to native ARM code, eliminating the need for a Javascript bridge.

• Access to Native Features: Flutter offers plugins and platform channels to access native device features like GPS, camera, sensors, etc.

• Google Support: Backed by Google, Flutter receives regular updates, stability improvements, and stable long-term support.

3b) Rendering Approach: Traditional frameworks rely on native components, but Flutter draws everything itself using Skia. This eliminates dependency on OEM widgets and prevents UI inconsistencies.

• Cross-Platform without Sacrificing Performance: Unlike frameworks like React Native, which use a bridge to communicate with native components, Flutter's direct compilation to native code avoids performance bottlenecks.

• Declarative UI: Flutter uses a declarative UI approach where UI automatically updates when app state changes - similar to modern front-end frameworks.

• Reduced Maintenance Effort: A single codebase means fewer bugs and easier updates, as developers don't need to maintain separate code for iOS and Android.

• Faster Development Cycles: Hot reload and a vast library of ready-to-use components drastically cut down development and testing time.

- In Flutter, everything is a widget - buttons, text, images, layouts, and even entire app itself. These widgets are organized in a tree structure called the widget tree.
- Widget Tree: Represents the visual structure and hierarchy of the app UI. Each widget is a node in the tree, defining part of the UI. When the app runs, Flutter builds this tree and renders it on the screen.
 - Widget Composition: Flutter encourages building UIs by combining small, reusable widgets into more complex ones instead of creating large, monolithic components. You compose smaller, widgets, each handling a specific responsibility.

Ex -

```
class Myapp extends StatelessWidget {  
  Widget build(BuildContext context) {  
    return MaterialApp( //  
      home: Scaffold( //  
        appBar: AppBar(title: //  
          body: Column( //  
            children: [ //  
              Text('') //  
            ] //  
          ) //  
        ) //  
      ) //  
    ); //  
  } //  
}
```

ElevatedButton(),
),
),
);
});
});
});
});

b) Commonly Used Widgets and Their Roles.

Structural Widgets:

- 1.) Scaffold : Provides a basic page structure, with an app bar, floating action button, etc.
 - 2.) App Bar : A toolbar for titles, icons and actions.
 - 3.) Drawer : A side navigation panel.

Layout Widgets

- 1.) Container : Adds padding, margins, borders and backgrounds.
 - 2.) Row and column : Arrange widgets horizontally and vertically.
 - 3.) Stack : Overlaps widgets on top of each other.

List View: creates scrollable lists

Display Widgets

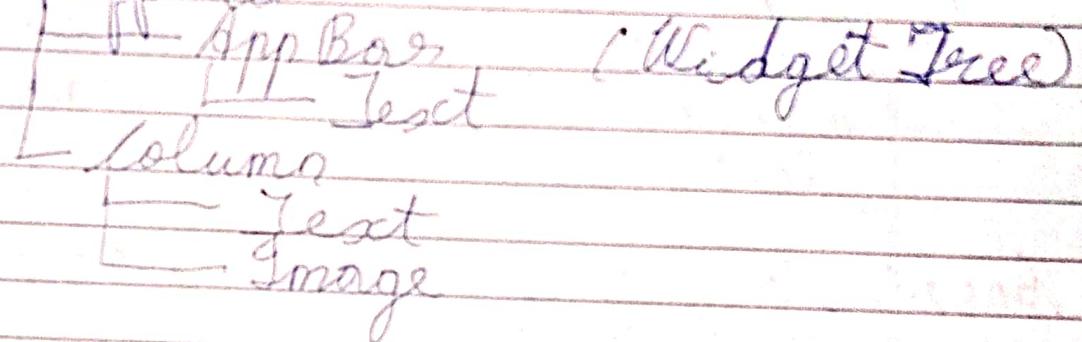
- 1.) Text: Displays static or dynamic text.
2.) Image: Displays images from assets in work or memory.

3.) Interactive Widgets:

- Elevated Button: A button with elevation
- Textfield: Input field for text entry
- Checkbox, Switch, Slider: Interactive UI elements.

MaterialApp

Scaffold



Q3:

Ans.a)

~~State management is crucial in Flutter because it controls how data flows and how UI updates in response to changes.~~

In Flutter, state refers to data that affects the appearance or behavior of a widget. Managing state correctly ensures that UI stays in sync with underlying data.

Why State Management is Important

UI Reactivity: Automatically rebuild the UI when the state changes, providing a dynamic user experience.

Performance Optimization: Efficiently rebuild

only the necessary widgets, preventing unnecessary renders.

- Data consistency: Maintain consistent data across different parts of the app even with complex UI structures
- Separation of concerns: Helps keep business logic separate from UI, making apps more maintainable and testable.

Q8(b) Comparing State Management Approaches in Flutter:

1.) Approach - setState

Description - Built-in simple way to manage state by calling `setState` to rebuild a widget.

Pros:

- Easy to implement
- Good for small, local state

Best for: Small apps, local state (e.g. toggling a button, updating text fields)

2.) Provider

Description - Inherited widget-based approach to pass data down the widget tree.

Pros:

- Lightweight and flexible

good for medium
easy to learn and widely supported
Best for: Medium to large apps

3) Approach - Riverpod

Description: A more robust, compile
safe state management solution that
fixes Provider's limitations.

- Pros - No need for context lookups.
 - Supports auto-disposal
 - More scalable and test friendly
- Best for: Complex, large-scale apps

~~When to Use Each Approach~~

~~setState:~~

- For small, local state changes within a single widget.

Example: A counter app.

~~Provider~~

- When you need to share state across multiple widgets.

Example: Managing user login status, theme switching, global settings.

~~Riverpod~~:

- For large, complex apps with multiple independent states.

Ex: - E-commerce apps with cart.

Q4)

Ans (ii) Integrating Firebase with a Flutter App Application

1. Steps to integrate Firebase in Flutter:
 1. Create a Firebase Project;
 2. Go to Firebase console - create a new project
 3. Add an App to Firebase:
 - Choose platform:
 - for Android: Download the google-service.json file and place it in android/app directory.
 4. Update Flutter Project:
 - 1. Add necessary Firebase packages in pubspec.yaml;
 - 2. Run flutter pub get to install dependencies
 5. Initialize Firebase:
 - 1. Add Firebase initialization in main.dart:
 - 2. import 'package:firebase_core/firebase_core.dart';
 - 3. void main() async {
 - 1. WidgetsFlutterBinding.ensureInitialized();
 - 2. await Firebase.initializeApp();

run App (MyApp()));

Configure Native Platforms:

For Android: Update android/app/build.gradle with Google services plugin

Benefits of Using Firebase:-

- 1.) Fully Managed Backend
- 2.) Real Time DataBase and Sync
- 3.) Authentication and Security
- 4.) Scalability
- 5.) Cloud Functions

Ans:-) Commonly Used Firebase Services:

- 1.) Firebase Authentication user login / sign up or social logins
- 2.) Cloud Firebase: NoSQL database with real-time synchronization -
- 3.) Realtime Database: JSON based database that syncs data in real time -
- 4.) Firebase Analytics: Tracks user interactions and app events.
- 5.) Firebase Storage: Stores and serves user-generated content like

images, videos, etc.

Data synchronization in Firebase
import 'package:cloud_firestore/cloud_firestore.dart';

Stream Builders

stream: FirebaseDatabase.getInstance().collection("messages").snapshots(),
builder: (context, snapshot) {
if(snapshot.hasData) return
CircularProgressIndicator();
else message = snapshot.get("text")
};

```
private Message messages = snapshot.data();
return ListView.builder()
    .itemCount(messages.length)
    .itemBuilder((context, index) {
        return ListTile(
            title: Text(messages[index].text),
            subtitle: Text(messages[index].id)));
    })
    .build();
```

~~2) j~~

1

Snapshots() → Stream Real-time updates

Teacher's Signs:

Teacher's Signs: