

MPL Assignment

OS

API

Q-1 a)

Explain the key features and advantages of using flutter for mobile development.



- (i) Cross-Platform Development - Write once, run on multiple platforms (Android, iOS, web, desktop).
- (ii) Rich UI Components - Uses customizable widgets for a native-like experience.
- (iii) Fast Rendering - Uses skia graphics engine for smooth animations.
- (iv) Hot Reload - Allows real-time code changes without restarting the app.
- (v) Native Performance - Compiles to ARM code for high speed execution.

Advantages

- (i) Rapid Development - Hot reload and a single codebase speed up the process.
- (ii) Cost savings - Reduces development time and resource requirements.
- (iii) Consistent UI Across Devices - looks the same on Android and iOS.
- (iv) Reduced performance issues - The app runs natively without relying on intermediate bridges like in react native reducing lag.

b)

Discuss how the flutter framework differs from traditional approaches and why it has gained popularity in the developer community.



- (i) Single codebase vs Separate codebase

Traditional Approach: Developers need to write separate code for android (Java/Kotlin) and iOS (Swift).

Flutter uses a single Dart based codebase for both platforms reducing development time and effort.

(ii) Rendering Engine vs Native UI Components.

Traditionally, dev relied on platform native UI components which can lead to inconsistencies and platform performance issues. whereas flutter uses the Skia rendering engine to draw everything from scratch ensuring a consistent UI across device.

Flutter has gained popularity because:-

- (1) Faster development with Hot Reload. Developers can instantly see UI changes without restarting the app making the iteration process much quicker.
- (2) Consistent UI across devices since flutter does not rely on native components, the UI looks and behaves the same across different OS versions.
- (3) Cross platform efficiency: Business save time and resources by maintaining a single codebase for multiple platforms.
- (4) Improved performance: AOT compilation and direct access to GPU rendering ensure smooth animations and high performance.
- (5) Easy Integration with Backend Technologies: Works well with Firebase, REST APIs, GraphQL, and other backend technologies simplifying full stack development.

Q2 a) Describe the concept of the widget tree in flutter. Explain how widget composition is used to build complex user interfaces.

→ Widget tree in flutter.

In flutter, the widget tree is the fundamental structure that represents the UI of an application. It is a hierarchical arrangement of widgets where each widget defines a part of the user interface. Flutter's UI is entirely built using widgets which can be stateless or stateful.

The widget tree determines how the UI is rendered and updated when changes occur.

Widget composition in flutter.

Widget composition refers to building complex UIs by combining smaller, reusable widgets. Instead of creating large, monolithic UI components, Flutter encourages breaking the UI into smaller manageable widgets that can be reused and nested with each other.

Example : class ProfileCard extends StatelessWidget {

final String name;

final String imageUrl;

ProfileCard({required this.name, required this.imageUrl});

@override

Widget build(BuildContext context) {

return Card(

child: Column(

children: [

Image.network(imageUrl),

sizedBox(height: 10)

Text(name, style: TextStyle(fontsize: 20, fontWeight:

fontWeight.bold))

Advantages of widget composition

- 1) Reusability: Small widgets can be reused in different parts of the app.
- 2) Maintainability: Breaking UI into smaller widgets makes it easier to debug and update.
- 3) Performance: Flutter efficiently rebuilds only the necessary parts of the widget tree.

b) Provide examples of commonly used widgets and their roles in creating a widget tree.

→ 1) Structural Widgets: These widgets act as the foundation for building the UI.

↳ MaterialApp: The root widget of a flutter app that provides essential configurations.

↳ scaffold: Provides a basic layout structure, including an app bar, body floating action button etc.

↳ container: A versatile widget used for styling, padding, margin and background customization.

e.g.: MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('Flutter Widget Tree')),

body: Container(

padding: EdgeInsets.all(16.0),

child: Text("Hello."),

),

),

);

2) Input and Interaction Widgets.

↳ Textfield - Accept text input from users.

↳ Elevation Button - A button with elevation

Gesture Detector - Detects gestures like taps, swipes and long presses

Ex: Column (

children : [

Textfield (decoration: InputDecoration (labelText: "Enter name"

Elevated Button (

onPressed: () {

print ("Button Pressed"),

},

child: Text ("Submit"),

)

],

);

3) Display and Styling Widgets

↳ Text - displays text on the screen

↳ Image - shows images from assets network or memory

↳ Icon - displays icons

↳ Card - A material design Card with rounded corners and elevation

Ex: Column (

children : [

Text ("Welcome to flutter !", style: TextStyle (fontSize: 24,

fontWeight: FontWeight.bold)),

Image.network ("https://flutter.dev/images/flutter-logo-sharing.png"),

],

);

D.3 a) Discuss the importance of state management in flutter applications

→ In flutter, state refers to data that can change during the lifetime of an application. This includes:

- User input
- UI changes
- Network changes
- Animation states

There are two types of states.

1) Ephemeral state: small, UI specific state that doesn't affect the whole app.

2) App wide states - Data shared across multiple widgets

Importance of state management

- Efficient UI updates : Flutter's UI is rebuilt whenever state changes. Efficient state management ensures that only necessary widgets are updated, improving performance.
- Code maintainability & scalability: Managing state properly makes the code modular, readable and scalable for larger applications.
- Data consistency & synchronization →

Proper state management ensures that data remains consistent across different screens and widgets.

b) Compare and contrast the different state management approaches available in flutter, such as setState, Provider, and Riverpod. Provide scenarios where each approach is suitable.

- setState . Local state

Pros - simple built in easy to use

Cons - Not Scalable causes unnecessary re-renders

Best use cases - Small UI updates (e.g. toggle switch, counter)

Provider - App wide state

Pros - lightweight, recommended by flutter, efficient

Cons - Boilerplate code for nested providers.

Best use cases - Medium scale apps (e.g. authentication, themes, API data).

Riverpod App - wide state (More scalable than provider)

Pros - Eliminates Provider's limitations, improves performance

Cons - Requires learning new concepts

Best use cases: large apps needing global state (e.g: shopping cart, user sessions)

Scenarios for Each Approach

- Use setState when managing simple UI elements within a single widget like toggling dark mode in a setting screen
- Use Provider when sharing state across multiple widgets such as managing user authentication or theme changes
- Use Riverpod when building a complex, scalable app with global state management, like an e-commerce app with cart management

(Q4.a) Explain the process of integrating ~~Firebase~~ with a flutter application.

Discuss the benefits of using ~~Firebase~~ as a backend solution.

→ Firebase provides a powerful backend solution for flutter applications offering services like authentication, real time databases, cloud functions, storage and more.

Steps to integrate firebase with flutter

Step 1 : Create a firebase project

go to firebase console

click on "Add project" and enter a project name

Configure google analytics if needed, then click create.

Step 2: Register the flutter app with Firebase

- In the firebase project dashboard click "Add App" and select Android or iOS based on your platform
- for Android: enter the android package name and download the google services.json file and place it in android/app/
- for iOS

Enter the iOS Bundle Identifier

Download the GoogleService-Info.plist file and place it in ios/runners

Step 3: Install firebase dependencies

Add firebase dependencies in pubspec.yaml

firebase core

firebase auth

cloud firestore

Run: flutter pub get

Step 4: Configure Firebase for Android & iOS

for Android

- open android/build.gradle and ensure the following classpath 'com.google.gms:google-services:4.3.10'
- Open android/app/build.gradle and add at the bottom apply plugin: 'com.google.gms.google-services'

Step 5: Initialize Firebase in flutter

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

Benefits of using firebase

Firebase is a Backend as a Service (BaaS) that simplifies backend development for flutter apps. Here are some key benefits.

1) Easy to set up and scale.

- ↳ No need to manage backend infrastructure

- ↳ Scales automatically based on usage

2) Authentication

- ↳ Provides email/password, Google, Facebook and phone authentication

- ↳ Seamless integration with Firebase Authentication

3) Cloud storage

Secure file storage for images, videos and documents.

a) Push Notifications (firebase cloud messaging)

Send real-time notifications to users across different platforms.

b) Highlight the firebase services commonly used in flutter development and provide a brief overview of how data synchronization is achieved

→ Firebase provides a suite of backend services that simplify flutter app development

1) Firebase Authentication

Enable secure authentication using email/password, phone number and third party providers like google, facebook and apple.

2) Cloud firestore

Stores and sync data in real time across devices supports structured data, queries and offline access.

e.g:- `firestore.instance.collection('users').add({`

`'name': 'John Doe',`

`'email': 'JohnDoe@example.com',`

`});`

3) Realtime Database :

A realtime JSON-based database that automatically updates data across devices

Ex. Database Reference ref = FirebaseDatabase.getInstance().getReference("message");
ref.set("text": "Hello, Firebase!");

4) Firebase Cloud Messaging (FCM)

Enable push notifications and messaging between users

Ex. firebaseMessaging.getInstance().subscribeToTopic("news");

5) Firebase Analytics

Tracks user interactions and app performance

Ex. firebaseAnalytics.getInstance().logEvent(name: "button_clicked", parameters:
{"button": "subscribe"});

6) Firebase Hosting

Deploys and serves web Applications securely with automatic SSL.

Data Synchronization in Firebase.

Firebase insures real-time data synchronization across multiple devices and platforms using Firebase and Realtime Database.

1) Cloud Firestore Sync Mechanism

Using realtime listeners to update UI instantly when data changes
Ex. FirebaseFirestore.getInstance().collection('users').snapshots().listen

((Snapshot) {

for (var doc in snapshot.docs) {

print(doc['name']);

FOR EDUCATIONAL USE

2) Realtime Database Sync Mechanism

uses persistent websocket connection for live updates

Eg: DatabaseReference ref = firebaseDatabase.getReference().ref("message").
ref.addValueListener(event){
print(event.snapshot.value);
},

3) Offline Data Sync

firestore caches data locally and syncs changes when the device
is online.

Eg: FirebaseFirestore.getInstance().settings.Settings(persistenceEnabled: true),

4) Cloud functions for automated updates

Automate backend logic to trigger update when data changes.