Prayag Jaiswal
D15B
24

Assignment no. 1 (MAD)

A

Q.1) (a) Explain the key features and advantages of using flutter for mobile app development.

→ Flutter, developed by Google, is an open-source UI toolkit for building natively compiled applications for mobile, web and desktop from a single codebase. Key features and advantages are as follows

(1) Single codebase: Write once and run on both android and iOS, reducing development time and effort.

(2) Fast development: Instantly see changes without restarting the app

(3) UI with customization: uses its own rendering engine to create highly customizable UIs

(4) Performance: Flutter compiles to native ARM code, ensuring smooth animations and fast execution.

(5) Widget library: use-designed material and cupertino widgets for both android and iOS.

(6) dart language: Uses Dart, which is optimized for UI development and improves performance.

(7) Strong community support: Regular updates, growing packages, and support from google.

(b) How flutter differs from traditional approaches and its popularity.

→ Traditional mobile development requires separate codebases for android and iOS, which increases complexity and maintenance costs.

Flutter differs by:

(1) Using widgets instead of native UI components: Everything in flutter is a widget, making UI development consistent across platforms.

(2) Rendering Engine (Skia): Unlike React native, which relies on native bridges, flutter draws everything directly using Skia, boosting performance.

(3) Faster Development with Hot Reload: No need to restart the app after making changes, unlike native development.

Also, Flutter reduces development effort with a single codebase.
High performance and rich UI customisation.
Strong backing from Google and a active developer community.

Q.2) (a) Describe the concept of the widget tree in flutter. Explain how widget composition is used to build complex user interfaces.

→ In flutter, everything is a widget (buttons, text, images, layouts). Widgets are organized in a hierarchical widget tree to structure the UI.

Widget compositions:

Flutter follows a composition-based approach where UI is built by nesting widgets inside other widgets.

Instead of modifying widgets (which are immutable), new widgets are created to reflect UI changes.
Complex UIs are built using simple widgets combined together.

example:
```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("")),
        body: Center(
          child: Column(
            children: [
              Text('Hello'),
              ElevatedButton(onPressed: ())
            ],
          ),
        ),
      ),
    );
  }
}
```

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

→ Commonly used widgets are:

(i) Structural widgets:
Container, Column, Row, Stack

(2) Interactive Widgets:
Elevated Button, Text Field, Gesture Detector.

(3) Styling and Display widget:
Text, Image, Card, Icon

(4) Stateful Widgets:
Stateful Widget.

Roles in widget tree: (1) Structural widget: Define UI layout.
(2) Interactive widget: Capture user input.
(3) Styling widget: Enhance UI
(4) Stateful widget: Maintain internal state

Q3) (a) Discuss the importance of state management in Flutter application.
→ State management controls UI updates in response to changes. Without it, UI would not reflect real-time updates (e.g. updating text after a button click).

Good state management ensures
Separation of UI and logic - improves maintainability.

- efficient UI updates - Avoids unnecessary rebuilds, improving performance.

- Scalability - essential for large applications with complex state interactions.

Q.3) (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.

→ (1) setState (Built-in, local state management):
Best for simple state changes in a single widget.
for small apps with minimal state needs

(2) Provider (Recommended by Flutter, global state management).
provides efficient state handling by rebuilding only necessary parts of UI
uses ChangeNotifier to manage state updates.
Best suited for: medium sized apps needing shared state across multiple widgets.

(3) Riverpod (Advanced, more scalable approach).
Eliminates dependency injection boilerplate and Provider
Ensures better testability and separation of concerns
Best suited for - Large scale applications with complex dependencies

Comparison:

setState	Provider	Riverpod
Simple, built in	efficient UI rebuilds	scalable, testable
efficient UI rebuilds	requires additional setup	learning curve
small, local UI changes	& medium size apps	large scale apps

Q4) (a) Explain the process of integrating firebase with a flutter application. Discuss the benefits of using firebase as a backend solution.

→ Steps to integrate firebase.

(1) Create a firebase project at Firebase console.

(2) Add an iOS/Android app and download the google-services.json.

(3) Install firebase packages in pubspec.yaml

(4) Initialize firebase in main.dart

(5) Use firebase services in the app (authentication, firestore, etc.)

Benefits of firebase as a Backend:
- No need for managing backend servers.
- Real-time database synchronization
- Scalable and secure authentication.
- Integrates with other Google services.

Q.4) (b) Highlight the firebase services commonly used in flutter development and provide a brief overview of how data synchronization is achieved.

→ Common firebase services and data synchronization.

(1) firebase authentication: Manages users login (Email/Password, Google, facebook, etc.)

(2) Cloud firestore - No SQL real-time database for storing and syncing structure data.

(3) Firebase storage - stores images, videos and files

(4) Firebase messaging (FCM) - sends push notifications.

(5) Firebase analytics - tracks users interactions and app usage.

Data synchronization in firestore
firestore provides real-time updates
example.

```
StreamBuilder (
    stream: FirebaseFirestore.
    builder: (context, snapshot) {
        if( !snapshot.hasData)
        return. var messages = snapshot.Data
        return
    },
);
}
```