2/9/24

ASSIGNMENT 2

Unit - 3 Navigating Between Screens

Q.1 Compare and contrast Navigator 1.0 and Navigator 2.0 in Flutter. How does ~~such~~ each handle navigation and what are the key differences in their approaches?

—↳ The Navigation system underwent a significant overhaul from Navigator 1.0 to Navigator 2.0.

i) Navigator 1.0 : The primary method for navigation is through 'push' and 'pop' operations.

- 'Push' : Adds a new route to the navigation stack, displaying the corresponding screen.
- 'Pop' : Removes the ~~content~~ current route from the stack, returning to the previous screen.

ii) Navigator 2.0 : Here, the navigation is done through 'RouterDelegate' and 'RouteInformation Parser' concepts.

- 'RouterDelegate' : Defines the application's navigation structure and handles route transitions.
- 'RouteInformationParser' : Converts between the RouteInformation (eg : URL) and the corresponding 'Route' object.

eg:

```
// Navigator 1.0
Navigator.push(
```

```
    context,
    MaterialPageRoute(builder: (context) =>
    SecondScreen()),
);
```

```
// Navigator 2.0
class MyRouterDelegate extends RouterDelegate<...>
class MyRouteInformationParser extends
RouteInformationParser<...>
```

	Navigator 1.0	Navigator 2.0
>	Imperative API (older approach)	Declarative API (newer approach)
>	Uses 'push()' and 'pop()' methods for navigation	Uses 'Page' and 'Router' objects for more control.
>	Less flexible; state management is tied to the widget tree.	Decouples state management from the widget ~~its~~ tree; State can be external.
>	Stack-based; handles routes internally.	Gives developers more control over the stack and routing logic.
>	Suitable for simple apps with basic navigation.	Ideal for complex apps needing custom navigation logic.

Q.2 Explain the importance of managing app state in a Flutter application. How can improper state management affect the user experience?

—↓ App state refers to the data and configuration of an app that defines its behaviour, appearance and data flow. Proper state management is crucial for ensuring a smooth and consistent user experience.

i) Consistency.

Proper state management ensures that the app's UI consistently reflects its data. For example, when data changes (like a new message in a chat app), the UI should update immediately to reflect this.

ii) Performance.

Efficient state management reduces necessary rebuilds and re-renders of the UI, enhancing the app's performance.

iii) Maintainability

Clear state management strategies make the app code more organized, easier to understand and simpler to maintain or debug.

iv) User Experience

Proper state management ensures that

user interactions (like pressing a button or switching tabs) result in the expected behavior without glitches, delays or inconsistencies.

→ How Improper State Management Can Affect User Experience in these ways :-

i) UI Glitches : Inconsistent or incorrect state updates can lead to visual ~~graphics~~, glitches, such as flickering elements or incorrect data display.

ii) Performance Issues : Inefficient state management can cause the app to lag or become unresponsive, especially on lower-end devices.

iii) Memory leaks : Poor state management practices can lead to memory leaks, where unused memory is not released, causing the app to consume more resources than necessary and potentially crashing.

iv) Increased Development Time : Debugging and fixing issues related to improper state management can be time-consuming and ~~frustatin~~ frustrating.

Q.3 Describe the process of creating and using a router in Flutter. How does it improve the structure and navigation flow of a large application?

—> A Router is a fundamental component that manages the navigation between different screens or pages. It provides a structured and efficient way to handle screen transitions, making complex applications easier to maintain and understand.

—> Steps to create and use a router :-

1. Define the Router Delegate : Create a custom class extending 'RouterDelegate' to handle navigation logic and control the app's navigation stack.

```
class MyRouterDelegate extends RouterDelegate<RouteSettings>
with ChangeNotifier, PopNavigatorRouterDelegateMixin
<RouteSettings> {
  final GlobalKey<NavigatorState> navigatorKey
        = GlobalKey<NavigatorState>();
  List<Page> _pages = [MaterialPage(child:
                      HomeScreen())];
  @override
  Widget build(BuildContext context) {
    return Navigator {
      key : navigatorKey,
      pages : List.of(_pages),
      onPopPage : (route, result) {
```

```
        if (!route.didPop(result)) {
            return false;
        }
        _pages.removeLast();
        notifyListeners();
        return true;
      },
    );
  }
  @override
  Future<void> setNewRoutePath(RouteSettings
                configuration) async {
    // Handle route change here
  }
}
```

2. Define the Route Information Parser: Create a class extending 'RouteInformationParser' to parse and convert route information (URLs) into a data model that the router delegate can use.

```
class MyRouteInformationParser extends Route
  InformationParser<RouteSettings> {
  @override
  Future<RouteSettings> parseRouteInformation
          (RouteInformation routeInfo) async {
    final uri = Uri.parse(routeInfo.location!);
    if (uri.pathSegments.isEmpty) {
      return RouteSettings(name: '/${uri.pathSegments
                                    .first}');
    }
  }
}
```

3. Set up the Router in the 'MaterialApp': Use the 'MaterialApp.router' widget to initialize the router, passing in the custom 'RouterDelegate' and 'RouteInformationParser'.

```
void main() {
  runApp(MaterialApp.router(
    routerDelegate: MyRouterDelegate(),
    routeInformationParser: MyRouteInformation
                            Parser(),
  ));
}
```

—4 ~~How~~ It improves the Structure and Navigation Flow of a Large Application in these ways:~

i) Declarative Approach: Provides a more predictable and maintainable Navigation model by clearly defining routes and state changes.

ii) Deep Linking Support: Built-in support for deep linking and URL synchronization, allowing direct navigation to specific parts of the app.

iii) Flexibility and Control: Offers granular control over the navigation stack, making it easier to manage complex navigation flows.

iv) Improved State Management: Can integrate

with various state management solutions (like 'Provider' or 'BLOC') to maintain a clear app state.

Q.4 Outline the steps involved in implementing a splash screen in a Flutter App. Why is it important to include a splash screen in some applications?

→ A splash screen is a brief introductory screen that appears before the main app's UI. It can be used to display a logo, loading indicator or other visual elements while the app is initializing.

→ Steps to Implement a splash screen in a Flutter App :-

1. Create a splash screen Widget : Build a Flutter widget that represents your splash screen with a logo, animation or image.

```
class SplashScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body : Center(
        child : Image.asset('assets/logo.png'),
      ),
    );
  }
}
```

2. Configure Splash Screen in Main Entry Point: Set the splash screen widget as the initial route or display it using a stateful widget that navigates to the main screen after a delay.

```
void main() {
  runApp(MaterialApp(
    home: SplashScreen(),
  ));
}
```

3. Add Timer for Navigation: Use a 'Future.delayed' method to navigate to the main screen after a set duration.

```
class SplashScreen extends StatefulWidget {
  @override
  _SplashScreenState createState() => _SplashScreenState();
}
class _SplashScreenState extends State<SplashScreen> {
  @override
  void initState() {
    super.initState();
    Future.delayed(Duration(seconds: 3), () {
      Navigator.pushReplacement(context,
        MaterialPageRoute(builder: (context) =>
        HomeScreen()));
    });
  }
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Image.asset('assets/logo.png'),
    ),
  );
}
}
```

4. Customize Native Splash Screen:
   - Android: Edit the 'styles.xml' file to customize the native splash screen.
   - iOS: Modify the 'LaunchScreen.Storyboard' file in Xcode.

5. Use 'flutter_native_splash' Package (Optional):
   - Install the 'flutter_native_splash' package to create a native splash screen automatically.
   - Configure it in 'pubspec.yaml' and run 'flutter pub run', 'flutter_native_splash:create' to apply changes.

→ Importance of Including a Splash Screen:-

i) Branding: Provides an opportunity to showcase the app's logo or brand identity.

ii) Loading Time Masking: Gives the app time to load necessary resources or perform

initializations without displaying a blank screen.

iii) Smooth Transition: Creates a smoother transition to the main content, enhancing the perceived performance and user experience.

iv) User Engagement: Engages users visually while the app prepares to present its main content.

Q.5 Discuss the considerations and best practices for designing a login screen in Flutter. How can you ensure both security and a smooth user experience?

—4 Considerations and Best Practices for designing a login screen :-

1. User Experience

A well-designed login screen should have a simple and clean layout with clear input fields for username/email and password. Additionally, providing "Forgot Password" and "Sign Up" options, along with social login buttons for convenience, can enhance the user experience.

2. Accessibility

To ensure accessibility, use large,

readable fonts and high-contrast colors. Make the screen navigable via keyboard or screen reader and label input fields properly to assist users with disabilities.

3. Responsiveness.

Design your login screen to be responsive to different screen sizes using MediaQuery or LayoutBuilder and use adaptive widgets to adjust UI components accordingly.

4. Validation

Validate input fields for correct formats and provide instant feedback on errors to guide users. This helps prevent mistakes and improves the user experience.

eg: Use TextFormField for input fields with validation, ElevatedButton for the login button and Scaffold to structure the screen.

→ Ensuring Both security and a smooth user Experience :-

i) Balance Security and Usability : Implement strong security measures (like HTTPS, token-based authentication, & secure storage) while ensuring that the login process is simple, fast and user-friendly.

ii) Provide Alternatives : Offer alternative login methods (social login, biometric) to give users more options and enhance security.

iii) Maintain Clarity and Simplicity : Use clear labels, concise instructions and a clean design to guide users through the login process without confusion.

iv) Handle Errors Gracefully : Inform users of errors clearly and promptly, with actionable messages that help them correct the issue without frustration.

Q.6 Explain the process of transitioning from a login screen to an onboarding screen in Flutter. What factors should be considered to ensure a seamless transition?

→ Steps to Transition from login to Onboarding Screen :-

1. Checking user login Status.
   Determine if the user is logging in for the first time or if they have already completed onboarding. This can be done using persistent storage, such as 'SharedPreferences' or secure storage.

2. Navigate Based on the User Status.
   If the user is new, navigate to the

onboarding screen. If onboarding is already completed, navigate directly to the main home screen.

3. Implement Navigation Logic.
Use Flutter's navigation methods like 'Navigator.pushReplacement', to replace the login screen with appropriate next screen.

4. Animate the Transition.
Apply smooth animation between the logic and onboarding screens to enhance the user experience, using Flutter's built-in transition or custom 'PageRoute' transitions.

→ Factors to Ensure a Seamless Transition :-

i) User status check : Ensure the user status is checked promptly without delays to decide the correct next screen.

ii) Smooth Navigation : use 'Navigator.pushReplacement' to avoid stacking screens and ensure smooth transitions between screens.

iii) Consistent animation : Apply consistent animations to maintain visual coherence, using Flutter's ~~PageBt~~ PageRouteBuilder or built-in Transition animations.

10) Minimal Delay : Minimize loading times between screens to keep the user engaged. Preload any data required for the onboarding or home screen.

Q.7 What are the key elements to consider when transitioning from an onboarding screen in a Flutter app? How can you maintain user engagement during this process?

→ Key Elements for transitioning from Onboarding to Home screen :-

1. Navigation Logic
   Store user onboarding status in persistent storage. Use 'Navigator.pushReplacement' for smooth navigation to home screen.

2. Animation and Transition Effects
   Apply consistent animations (slide, fade) for seamless experience. Use 'PageRouteBuilder' for custom animations aligned with the app's branding.

3. Loading and Preloading Content.
   Preload necessary data during onboarding to minimize delays. Show loading indicators to show that the app is preparing content if data fetching is needed.

Q.

4. Retention Mechanisms.

Highlight key app features on the last onboarding screen. Include a strong call to action (CTA) to encourage users to proceed to the home screen.

→ Maintaining User Engagement during the Transition :-

i) Smooth Transitions : use consistent and smooth animations to provide a natural and engaging flow between onboarding and home.

ii) Positive Feedback : Display messages / animations that congratulate or welcome users, making them feel valued.

iii) Quick loading : Minimize loading times with preloaded content or background data fetching to keep users engaged.

iv) Personalization : If possible, personalize the home screen based on user preferences gathered during onboarding.

Q.8 How can you manage tab selection and state retention in a Flutter application? Provide an example to illustrate your approach.

→ When working with a tab bar, it's crucial to manage the selected tab and ensure that the state of each tab is preserved when the user switches between them, which prevents data loss and maintains a consistent user experience.

→ Key Approaches to Manage Tab Selection and State Retention :-

1. Using 'TabBarController'.
   Manages the state and synchronization of the tabs and their associated views. This allows for programmatic control over the tabs.

2. Using 'AutomaticKeepAliveClientMixin'.
   This mixin is used to preserve the state of each state so that when a user switches between tabs, the content does not reset or reload. By overriding 'wantKeepAlive' to return 'true', the state of each tab is kept alive.

eg :-

```
import 'package:flutter/material.dart';
void main() => runApp(MaterialApp(home:
                TabExample()));
class TabExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
        return DefaultTabController(
          length: 3,
          child: Scaffold(
            appBar: AppBar(
              bottom: TabBar(tabs: [
                Tab(text: 'Home'),
                Tab(text: 'Search'),
                Tab(text: 'Settings')
              ]),
            ),
            body: TabBarView(children: [
              HomeTab(),
              Center(child: Text('Search')),
              Center(child: Text('Settings')),
            ]),
          );
        );
      }
}
class HomeTab extends StatefulWidget {
  @override
  _HomeTabState createState() => _HomeTabState();
}
class _HomeTabState extends State<HomeTab> with
                AutomaticKeepAliveClientMixin {
  int _counter = 0;
  @override
  Widget build(BuildContext context) {
    super.build(context);
    return Center(
```

```
        child : ElevatedButton (
          onPressed : () => setState (() => counter++),
          child : Text ('Counter : $ - counter'),
        ),
      );
    }
    @override
    bool get wantKeepAlive => True; // Retain State
}
```

Q.9 What are deep links and how are they used in Flutter applications? Provide an example of implementing deep links to navigate to specific screens.

—4 Deep links are URLs that take users directly to a specific content or page within a mobile app. When a user clicks on a deep link (from an email, a social media post or a web page), the app is launched and the user is taken to a particular screen, bypassing the default home or splash screen. Deep linking enhances user engagement by simplifying navigation, improving user experience and increasing conversion rates by directing users to specific, relevant content immeadiately.

—4 Types of Deep links :-

(1) Basic Deep Links : Regular URLs (like myapp://screen1)

that open the app to a specific screen if the app is already installed.

(2.) Defferred Deep Links : ~~URLs~~ These work even if the app isn't installed. Users are redirected to the app store and upon installation, the app opens on the desired screen.

(3.) Universal (iOS) / App (Android) links : Web URLs that can open the app if it's installed or fallback ~~at~~ to website if the app is not installed.

→ Implementation Deep links :~

1. Add Dependencies : Add the 'uni_links' package to your 'pubspec.yaml'.

```
dependencies :
  flutter :
    sdk : flutter
  uni_links : ^0.5.1
```

2. Configure Platform-Specific Files :
   - Android : Modify 'AndroidManifest.xml' to define an intent filter.
   - iOS : Configure URL Schemes in the 'Info.plist'.

3. Listen for Deep Links : Use the 'uni_links' package in your main Flutter code to listen for incoming deep links.

eg:
```
import 'package:flutter/material.dart';
import 'package:uni_links/uni_links.dart';
import 'dart:async';
void main() => runApp(MaterialApp(home:
                DeepLinkHandler()));
class DeepLinkHandler extends StatefulWidget {
  @override
  _DeepLinkHandlerState createState() => _DeepLink
                HandlerState();
}
class _DeepLinkHandlerState extends State<Deep
                LinkHandler> {
  late final StreamSubscription _sub = uriLinkStream.
                listen(url) {
    if (uri != null) Navigator.pushNamed(context,
                uri.path);
  });
  @override
  void dispose() {
    _sub.cancel();
    super.dispose();
  }
  @override
  Widget Build(buildContext context) {
    return MaterialApp(
      routes: {
        '/home': (_) => Screen('Home Screen'),
        '/profile': (_) => Screen('Profile Screen'),
      },
      home: Scaffold(
```

```
        appBar: AppBar(title: Text('Deep Link
                                    Example')),
        body: Center(child: Text('Welcome!'));
      ));
    );
  }
}
class Screen extends StatelessWidget {
  final String title;
  Screen(this.title);
  @override
  Widget Build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(title));
      body: Center(child: Text('This is the
                                $title')),
    );
  }
}
```

Q.10 Discuss the role of a navigation state object in Flutter. How does it interact with other components of the app's navigation system?

—4 A Navigation State Object is essential for managing and controlling the app's navigation stack. It tracks the current navigation state, including active screens and their order and provide methods to manipulate the stack (eg: pushing or popping routes). This ensures a smooth and consistent user experience.

— Key Roles of the Navigation State Object :-

1. Manages Navigation Stack.
   Keeps track of all active routes (screens) and their order. Handles adding, removing or replacing screens in the stack.

2. Provides Navigation Methods.
   Exposes methods like push(), pop(), pushReplacement(), etc. to navigate between different screens. It centralizes all navigation actions, making it easier to manage complex navigation flows.

3. Interacts with Routing Mechanisms.
   Works closely with the 'Router' and 'Route' objects to determine which screen to show and how to transition between screens. It supports both imperative and declarative navigation approach.

4. State Synchronization
   Ensures that the navigation state is synchronized with app's state management solution and also maintains consistent state across deep links, back button handling and screen transitions.

— Interactions with Other Components :-

i) Navigator : Uses the state object to manipulate the navigation stack, adding or removing routes

ii) Router : Works with the state object to determine the route to be displayed based on app state.

iii) State Management : Syncs navigation actions with app state to maintain a coherent user experience.

iv) UI Widgets : Triggers navigation events (eg: button clicks) that interact with the stage object.

Q.11 What is a route information parser in Flutter and how does it work? Explain its significance in the context of handling deep links.

-> A Route Information Parser is a core component of the declarative navigation system and it is responsible for converting a platform-specific route information (like a URL or ~~dep~~ deep link) into a type-safe data structure that the app can use to ~~whi~~ determine which screen or route to display.

-> The Route Information works like :

1. Receives Route Information (Input)

When a user navigates using a deep link or types a URL directly, the system provides the route information (eg: /home).

2. Parses the Route
The 'RouteInformationParser' converts this raw route information into a data model that represents the app's internal navigation state.

3. Generate Route State (Output)
The parsed route state is then used by a 'RouterDelegate' to build and manage the navigation stack. The 'RouterDelegate' decides which page or widget should be displayed based on parsed information.

→ Significance of Route Information Parser in Deep linking are :-

i) Deep link Support : Interprets URLs from external sources (emails) to navigate directly to specific app states.

ii) Dynamic Routing : Enables complex navigation flows by loading specific pages or content directly from a URL.

iii) Flexibility : Offers greater control over navigation by decoupling route parsing from page

rendering, unlike traditional Navigator 1.0 API.

Q.12 Describe the steps to connect a route information parser to an app router in Flutter. How does this connection facilitate navigation?

→ Steps to connect a Route Information Parser to an App Router :-

Step-1: Create Route Information Parser:

```
class MyRouteInformationParser extends RouteInformationParser<String> {
  @override
  Future<String> parseRouteInformation(RouteInformation routeInfo) async {
    return Uri.parse(routeInfo.location ?? '/').path;
  }
  @override
  RouteInformation restoreRouteInformation(String path) {
    return RouteInformation(location: path);
  }
}
```

Step-2: Define Router Delegate:

```
class MyRouterDelegate extends RouterDelegate<String>
                       with ChangeNotifier {
  String _currentPath = '/';
  @override
  Widget Build(BuildContext context) {
```

```
return Navigator(
  pages: [
    if (_currentPath == '/') MaterialPage(child: HomePage()),
    if (_currentPath == '/profile') MaterialPage
                                    (child: ProfilePage()),
  ],
  onPopPage: (route, result) {
    _currentPath = '/';
    notifyListeners();
    return route.didPop(result);
  },
);
}
@override
Future<void> setNewRoutePath(String path) async {
  _currentPath = path;
  notifyListeners();
}
}
```

Step-3: Configure MaterialApp.router:

```
void main() {
  runApp(MaterialApp.router(
    routeInformationParser: MyRouteInformationParser(),
    routerDelegate: MyRouterDelegate(),
  ));
}
```

→ This Connection Facilitate Navigation as :-

i) Converts URLs to App State : The 'RouteInformation

Parser' translates URLs into a structured app state, determining which screen to display.

ii) Updates Navigation Stack: The 'RouterDelegate' uses this state to adjust the navigation stack, managing which pages to show or hide.

iii) Synchronizes URL and State: This approach keeps the browser's URL and app state in sync, ensuring a consistent user experience across different platforms.

Q.13 Explain the process of converting a URL into an app state in Flutter. Why is this process crucial for handling deep links?

→ Converting a URL into App State :-

1. Receive URL
When a URL is accessed (eg: through deep links or browser navigation), Flutter captures this URL.

2. Parse the URL
The 'RouteInformationParser' converts the raw URL into a structured data model representing the app state.

☆ Same code as Step-1 in Question-12. (not the 2nd override)

3. ~~Regenarate~~ Regnerate Navigation ~~Stack~~ State.

The parsed data model is used by 'RouterDelegate' to build the navigation stack and display the appropriate screen.

* Same code as step-2 in Question-12. Only add this in 'onPopPage'
  - if(!route.didPop(result)) return false;

—↳ Importance for Handling Deep links :-

i) Direct Navigation : Translates URLs into app state, allowing users to navigate to specific parts of the app without manually entering the app's main screen.

ii) Consistency : Keeps the app state in sync with the URL, ensuring a smooth and consistent user experience.

iii) Web Integration : Supports web apps by aligning URLs with app states, facilitating seamless transitions and navigation.

Q.14 How can an app state can be converted back to a URL in Flutter? Discuss the importance of this process in maintaining the app's navigation state.

—↳ Converting App State to a URL :-

1. Define App State
   Represent the app state as a data

model that describes the current navigation state.

```
class RoutePath {
  final String path;
  final Map<String, String> queryParameters;
  RoutePath (this.path, this.queryParameters);
  bool get isHomePage => path == '/';
  bool get isProfilePage => path == '/profile';
}
```

2. Use Route Information Restorer.

Implement 'RouteInformationParser' to convert the app state back into a URL that reflects the current navigation state.

* Same code as Step-1 in Question-12

-4 Importance of This Process :-

i) Deep linking support : Converts app state into URLs, enabling users to bookmark, share or revisit specific app screens through direct links.

ii) Consistency : Ensures that URLs accurately represent the current app state, allowing users to return to the same place in the app as before.

iii) Browser Navigation : Facilitates web navigation and history management, ensuring a

seamless experience across browser sessions and app states.

Q.15 Outline the methods used to test deep links in Flutter application. what challenges might arise and how can they be mitigated?

-> Testing deep links is crucial to ensure that your app handles URL-based navigation correctly. The Methods for testing deep links are :-

1. Manual Testing.
   - Direct URL Navigation : Manually type deep links into a web browser to see if they open the correct screen.
   - Custom URL schemes : On iOS, use the 'xcrun simctl openurl booted "myapp://profile" command and on Android, use 'adb shell am start -d "myapp://profile".

2. Integration Testing (Automated Testing)
   - Use Integration Test Framework : Write integration tests to stimulate navigation and verify that deep links are handled correctly across the app.

3. Unit Testing
   - Mock Deep Links : Use testing libraries like 'flutter_test' to stimulate deep link navigation and verify that the app handles them correctly.

- Challenges and Mitigation Strategies :-

i) Platform-Specific Behaviour.
- Challenge : Deep link handling may differ between iOS and Android.
- Mitigation : Test deep links on both platforms to ensure consistent behaviour.

ii) Complex Navigation Flows.
- Challenge : They may involve complex navigation, especially with nested routes.
- Mitigation : Use thorough integration testing to cover various scenarios and ensure accurate routing.

iii) Handling Edge Cases.
- Challenge : Edge cases like malformed URLs or invalid routes can ~~access~~ cause crashes or unexpected behaviour.
- Mitigation : Implement robust error handling and validation in your deep link parsing logic.

iv) Environment Differences
- Challenge : Deep links might behave differently in development vs. production environments.
- Mitigation : Test deep links in a staging environments that closely mirror production.

~~~~ x ~~~~ x ~~~~
~~~~