Experiment No.5
MAD & PWA LAB

I. Aim: To apply navigation, routing and gestures in Flutter App.

II. Theory:

Navigation and **Routing** are essential for mobile applications, allowing users to move between screens. In Flutter, navigation can be done imperatively using the Navigator widget or declaratively using the Router widget. The MaterialApp.routes property is often sufficient for small apps.

There are two main approaches to navigation in Flutter: imperative and declarative. The imperative approach uses the Navigator widget to directly manipulate the navigation stack. This involves pushing new routes onto the stack to navigate to different screens and popping routes off the stack to go back. The **Navigator.push()** method is used to push a new route onto the stack, while **Navigator.pop()** is used to pop the current route off the stack.

The declarative approach, on the other hand, uses the Router widget to define the app's routes and how they should be displayed. This is similar to how widgets are built using the build() method. The MaterialApp widget's routes property can be used to define the app's routes in a declarative way.

In a simple Flutter app, you can often use the MaterialApp.routes property to define your routes. For example, you might have a login screen route and a home screen route, each with a button that uses Navigator.push() to navigate to the other route. Overall, understanding how to navigate between screens is essential for building a successful Flutter app, and both the imperative and declarative approaches offer flexibility and power in managing your app's navigation flow.

To navigate in Flutter:

1. Create two routes: **SignIn Screen** and Home Screen, each with a button.
2. Use **Navigator.push()** to move from Signup to Home, adding the new route to the stack.
3. Use **Navigator.pop()** to return from Home to SignUp, removing the top route from the stack.

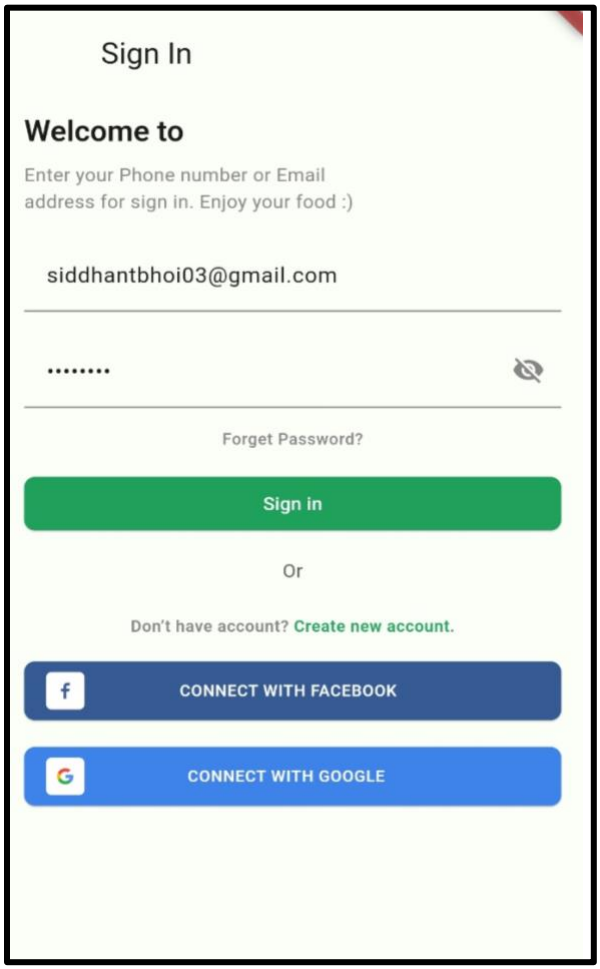
Navigator.push() manages routes and uses MaterialPageRoute for platform-specific animations.

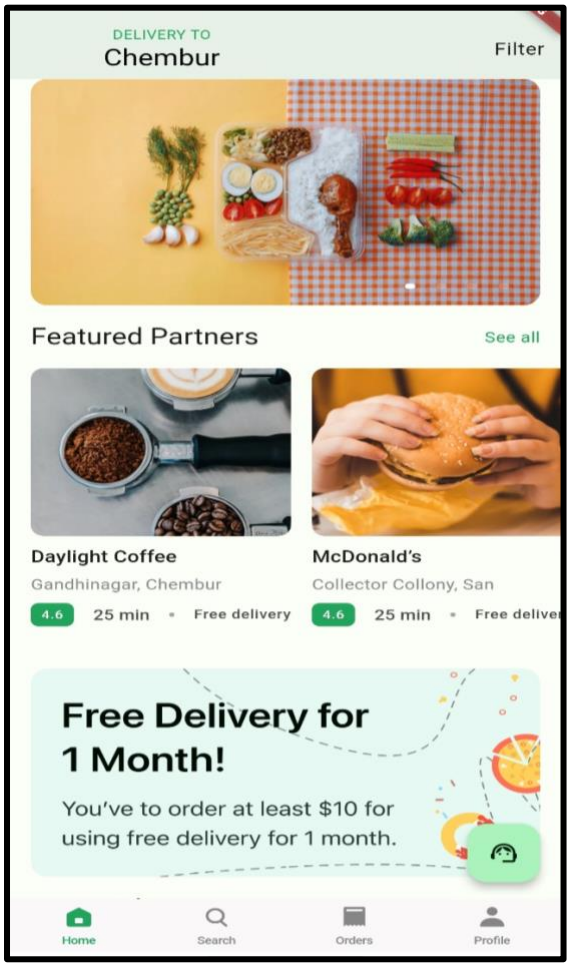
```
recognizer: TapGestureRecognizer()
      ..onTap = () => Navigator.push(
                  context,
                  MaterialPageRoute(
                  builder: (context) => const SignUpScreen(),
                  ), ),
```

Now, we need to use the **Navigator.pop()** method to close the second route and return to the first route. The pop() method allows us to remove the current route from the stack, which is managed by the Navigator.

```
recognizer: TapGestureRecognizer()
      ..onTap = () => Navigator.pop(
                  context,
                  MaterialPageRoute(
                  builder: (context) => const HomeScreen(),
                  ),),
```

So, on clicking the **Sign In** button in the SignIn Screen, we redirect to the Home Screen. And after clicking on the **Sign Out** button in the Home Screen, we go back to the SignIn Screen.





Gesture Detector:

Gestures are an interesting feature in Flutter that allows us to interact with the mobile app (or anytouch-based device). Generally, gestures define any physical action or movement of a user in the intention of specific control of the mobile device. Some of the examples of gestures are:
1. When the mobile screen is locked, you slide your finger across the screen to unlock it
2. Tapping a button on your mobile screen
3. Tapping and holding an app icon on a touch-based device to drag it across screens.

Flutter provides a widget that gives excellent support for all types of gestures by using the **GestureDetector** widget.The GestureWidget is non-visual widgets, which is primarily used for detecting the user's gesture. The basic idea of the gesture detector is a stateless widget that contains parameters in its constructor for different touch events.

Signup:

```
Center(
  child: Text.rich(
    TextSpan(
      style: Theme.of(context)
          .textTheme
          .bodySmall!
          .copyWith(fontWeight: FontWeight.w600),
      text: "Don’t have account? ",
      children: <TextSpan>[
        TextSpan(
          text: "Create new account.",
          style: const TextStyle(color: primaryColor),
          recognizer: TapGestureRecognizer()
            ..onTap = () => Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (context) => const SignUpScreen(),
                  ),
                ),
        )
      ],
    ),
  ),
),
```

Home:

```
floatingActionButton: FloatingActionButton(
        onPressed: () {
          Navigator.push(
            context,
            CupertinoPageRoute(builder: (context) => const SupportPage()),
          );
        },
        child: Icon(Icons.support_agent), // Icon for support agent
      ),
```

```
SectionTitle(
                title: "Featured Partners",
                press: () => Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (context) => const FeaturedScreen(),
                  ),
                ),
              ),
```

```
 onPressed: () {
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) => const FilterScreen(),
                ),
              );
            },
```

III. Conclusion:

Since, navigating and routing are crucial for Flutter app development, allowing seamless screen transitions. During implementation, managing the navigation stack and preserving app state posed challenges. To address this, utilizing state management solutions like Provider or Riverpod proved effective. By implementing these techniques, the app's state was preserved correctly, enhancing the overall user experience.