Experiment No: 5

Experiment No 5 **Aim: To apply navigation, routing and gestures in Flutter App**	
ROLL NO	**30**
NAME	**Kaushik Kotian**
CLASS	**D15-B**
SUBJECT	**MAD & PWA Lab**
LO-MAPPED	

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

Theory:

Navigation and Routing
Flutter uses a Navigator widget to manage a stack of screens (or routes). This stack allows users to navigate between screens by pushing new routes onto the stack (navigating forward) or popping routes from the stack (navigating back). The MaterialApp widget, which wraps the entire app, holds the configuration for routes, making it possible to navigate using named routes.

Named Routes: Instead of directly creating and navigating to a new screen, named routes allow developers to define a map of route names and corresponding widgets. This approach centralizes navigation logic, making it easier to manage.
Argument Passing
Passing arguments between routes is a common requirement. Flutter supports passing arbitrary objects as arguments during navigation. In the provided example, a custom class ScreenArguments is used to pass a title and message from the HomePage to the SecondPage.

Passing and Extracting Arguments: When navigating to a route, you can pass arguments using the arguments parameter of Navigator.pushNamed(). The receiving route extracts these arguments, typically in its build method, using ModalRoute.of(context)?.settings.arguments.
Gestures

Flutter's GestureDetector widget allows you to detect and respond to gestures, such as taps, double taps, drags, and more. In the example, GestureDetector is used to detect a tap on a container, triggering navigation to the second screen.

Gesture Detection: Gesture Detection
In this updated example, gesture detection is implicitly used through the ElevatedButton widget. When the user taps (a gesture) on the ElevatedButton, the onPressed callback is triggered. This is a form of gesture detection, although it's handled by the button widget itself rather than a direct use of the GestureDetector widget.

In Flutter, many widgets like ElevatedButton, FlatButton, IconButton, and others internally use gesture detection to respond to user interactions such as taps. The GestureDetector widget provides a more granular and lower-level way to handle various user gestures (like taps, double taps, long presses, swipes, etc.) directly on any part of the widget tree, but in this case, the simple action of tapping a button to navigate to another screen does not require direct use of GestureDetector.

Code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Navigation Demo',
      home: HomePage(),
      routes: {
        '/second': (context) => SecondPage(),
      },
    );
  }
}

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home Page'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pushNamed(
              context,
              '/second',
              arguments: ScreenArguments(
                'Hello from Home Page',
                'This message was passed as an argument!',
              ),
            );
          },
          style: ButtonStyle(
```

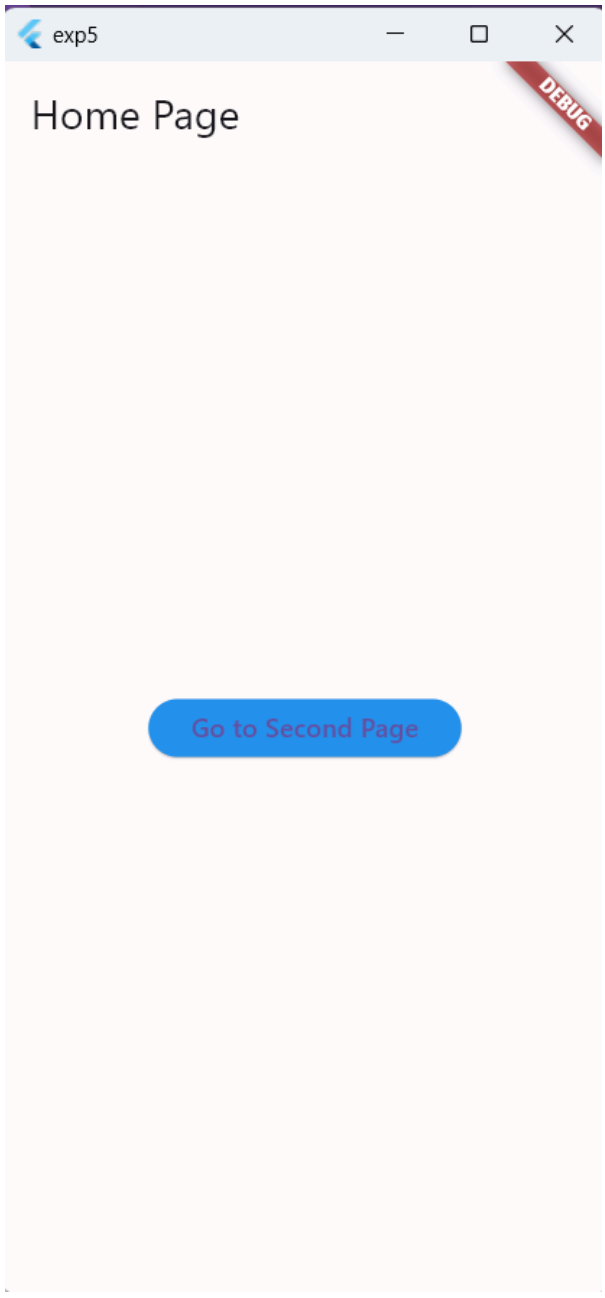
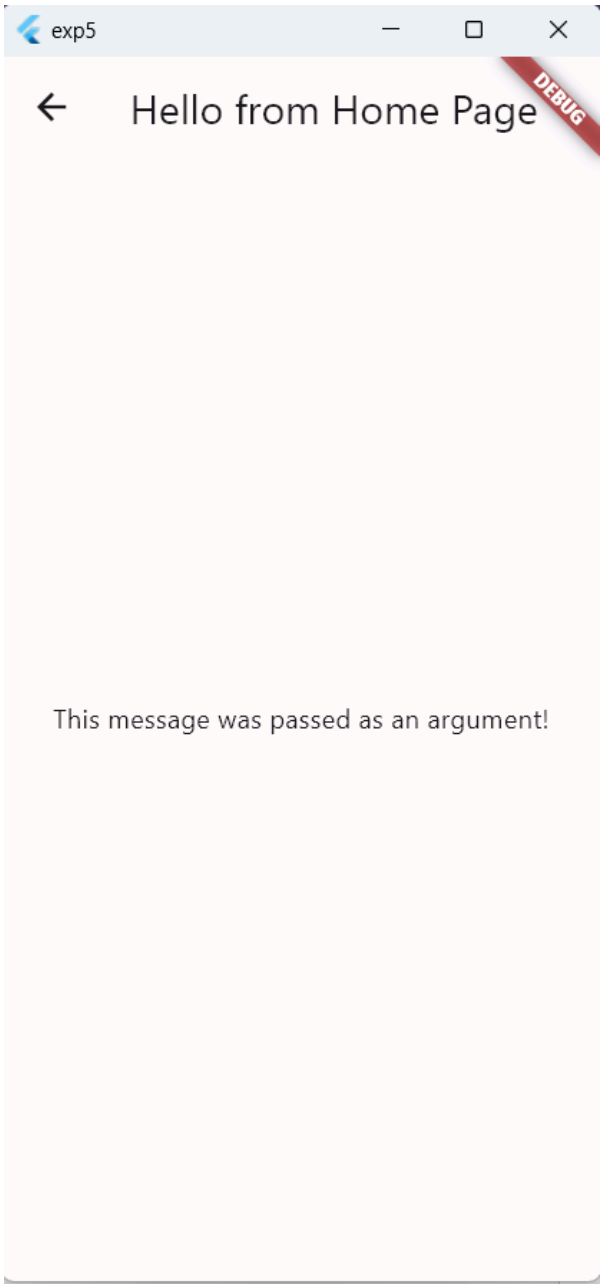
```
            backgroundColor: MaterialStateProperty.all<Color>(Colors.blue),
          ),
          child: Text('Go to Second Page'),
        ),
      ),
    );
  }
}

class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final ScreenArguments args =
        ModalRoute.of(context)?.settings.arguments as ScreenArguments;

    return Scaffold(
      appBar: AppBar(
        title: Text(args.title),
      ),
      body: Center(
        child: Text(args.message),
      ),
    );
  }
}

class ScreenArguments {
  final String title;
  final String message;

  ScreenArguments(this.title, this.message);
}
```

First Page	Second page
	

Conclusion:
The provided Flutter application exemplifies how to implement navigation between screens using named routes, pass arguments between these screens, and handle user gestures for interaction. This example serves as a foundational piece for understanding how to create multi-screen applications with Flutter that are interactive and capable of passing data seamlessly between screens.