EXPERIMENT 5

Aim: To apply navigation,routing and gestures in Flutter

Theory:
In Flutter, navigation and routing are crucial aspects for building multi-screen applications. They allow users to move between different parts of the app smoothly. Here's a breakdown of the key concepts:

Navigation
Navigation refers to the process of moving between different screens (or routes) within an app. It involves transitioning from one visual state to another. In Flutter, navigation is managed by the `Navigator` class, which maintains a stack of routes.

Routing
Routing, on the other hand, is the mechanism by which an app determines which screen (or route) to display for a given user action. It involves defining routes, mapping them to specific screens, and handling navigation events.

Key Concepts:

1. Routes: In Flutter, a route represents a screen or page in the app. Each route is typically associated with a widget that defines the UI for that screen.

2. Navigator: The `Navigator` class manages a stack of routes. It provides methods for pushing, popping, and replacing routes, allowing for navigation between screens.

3. Named Routes: Named routes provide a way to navigate to a specific screen using a unique identifier (route name) instead of directly referencing the widget class. This approach makes it easier to manage navigation and decouples the UI from the navigation logic.

4. Route Parameters: Routes can accept parameters, which allow passing data from one screen to another. This is useful for dynamically updating the UI based on user interactions or external data.

5. Navigation Methods: Flutter provides several methods for navigation, including `Navigator.push()`, `Navigator.pop()`, `Navigator.pushNamed()`, and `Navigator.popAndPushNamed()`. These methods are used to navigate forward, backward, or replace routes on the stack.

Workflow:

1. Define Routes: Begin by defining routes for each screen in your app. This involves creating a mapping between route names and corresponding widget classes.

2. Navigate Between Screens: Use navigation methods to move between screens based on user interactions, such as button clicks or gestures. You can push new routes onto the stack, pop routes off the stack, or replace routes as needed.

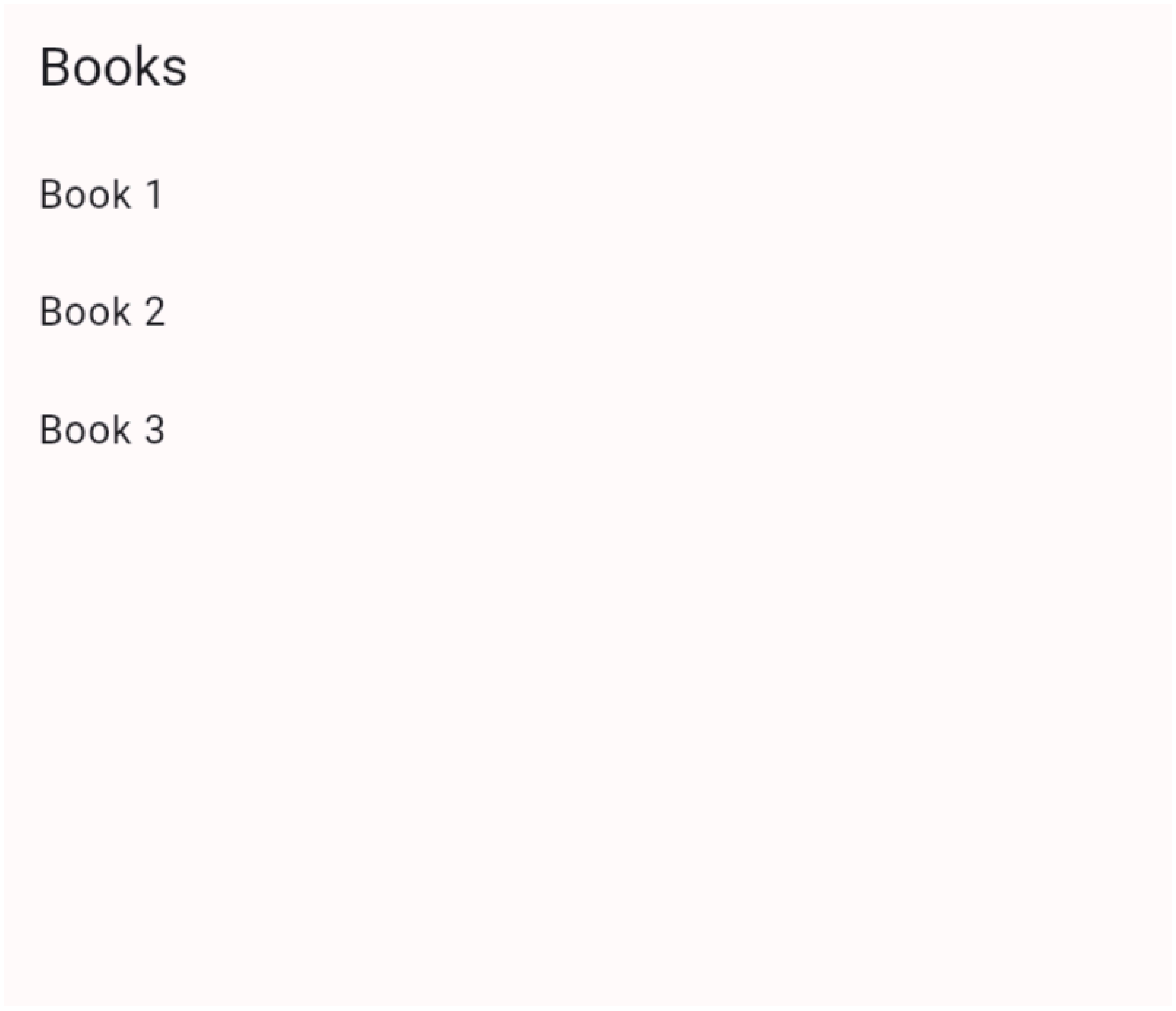
3. Pass Data Between Screens: Utilize route parameters to pass data between screens when navigating. This allows screens to communicate with each other and update their UI accordingly.

4. Handle Navigation Events: Implement logic to handle navigation events and respond to user actions appropriately. This may involve showing loading indicators, validating inputs, or performing other tasks before navigating to a new screen.

5. Handle Navigation Back: Consider how users will navigate back to previous screens using the system back button or gestures. Ensure that your app handles navigation back correctly and maintains a consistent user experience.

Implementation:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Book Reading App',
      initialRoute: '/',
      routes: {
        '/': (context) => HomePage(),
        '/bookDetail': (context) => BookDetailPage(),
      },
    );
  }
}

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Books'),
      ),
      body: ListView(
        children: [
          ListTile(
            title: Text('Book 1'),
            onTap: () {
              Navigator.pushNamed(context, '/bookDetail');
            },
          ),
          ListTile(
            title: Text('Book 2'),
            onTap: () {
              Navigator.pushNamed(context, '/bookDetail');
            },
          ),
          ListTile(
            title: Text('Book 3'),
            onTap: () {
              Navigator.pushNamed(context, '/bookDetail');
            },
          ),
        ],
      ),
    );
  }
}

class BookDetailPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Book Detail'),
      ),
      body: Center(
        child: Text('Book Details'),
      ),
    );
  }
}
```

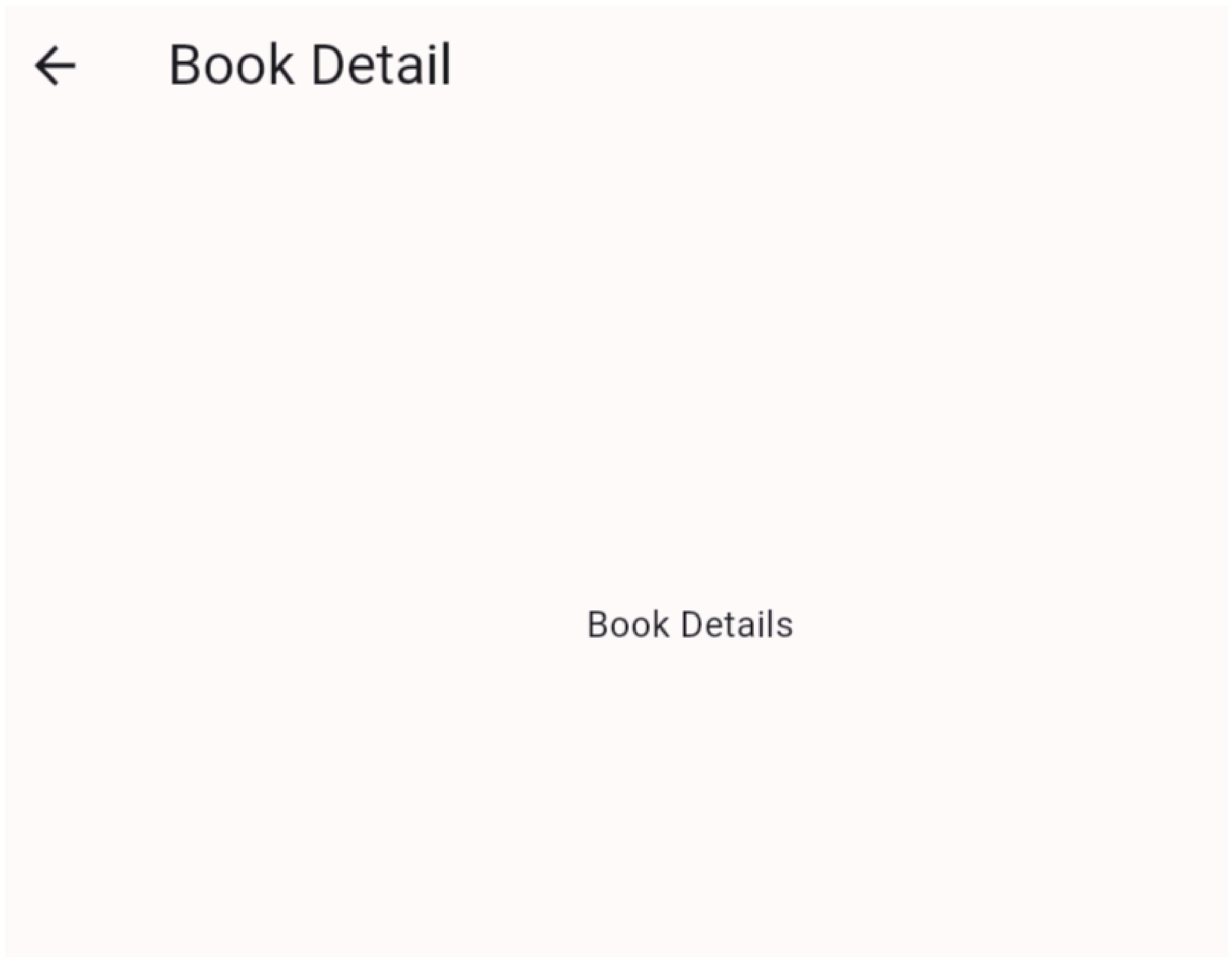


Books

Book 1

Book 2

Book 3



Gestures:

Gestures in Flutter refer to user interactions such as taps, drags, swipes, pinches, and more. Flutter provides a rich set of built-in widgets and APIs to handle various types of gestures efficiently.

Key Concepts:

1. GestureDetector Widget: The `GestureDetector` widget is the primary tool for detecting gestures in Flutter. It wraps its child widget and provides callbacks for various types of gestures.

2. Gesture Recognizers: Flutter uses gesture recognizers to interpret raw pointer events into high-level gestures. Gesture recognizers translate low-level touch events (like down, move, and up) into semantic gestures (like tap, double tap, drag, etc.).

3. Callbacks: The `GestureDetector` widget offers several callback properties to handle different types of gestures, such as `onTap`, `onDoubleTap`, `onLongPress`, `onPanUpdate`, `onVerticalDragStart`, `onScaleStart`, and many more.

4. Gesture Detection Process: When a user interacts with the screen, Flutter's gesture detection system captures the raw pointer events and dispatches them to the appropriate gesture recognizer. If the recognizer identifies a recognized gesture, it triggers the corresponding callback.

5. Gesture Behavior: Flutter provides flexibility in defining how gestures behave within widgets. For example, you can customize the area where gestures are detected using the `behavior` property of the `GestureDetector`, or you can specify the direction in which a drag gesture can occur using the `dragStartBehavior` property.

Workflow:

1. Add GestureDetector: Wrap the widget(s) that you want to make interactive with a `GestureDetector`.

2. Define Callbacks: Specify the appropriate callback properties of the `GestureDetector` to handle the desired gestures. For example, use `onTap` for single taps, `onDoubleTap` for double taps, `onLongPress` for long presses, etc.

3. Implement Gesture Handlers: Write the corresponding callback functions to respond to the detected gestures. These functions will contain the logic to execute when the user performs the specified gesture.

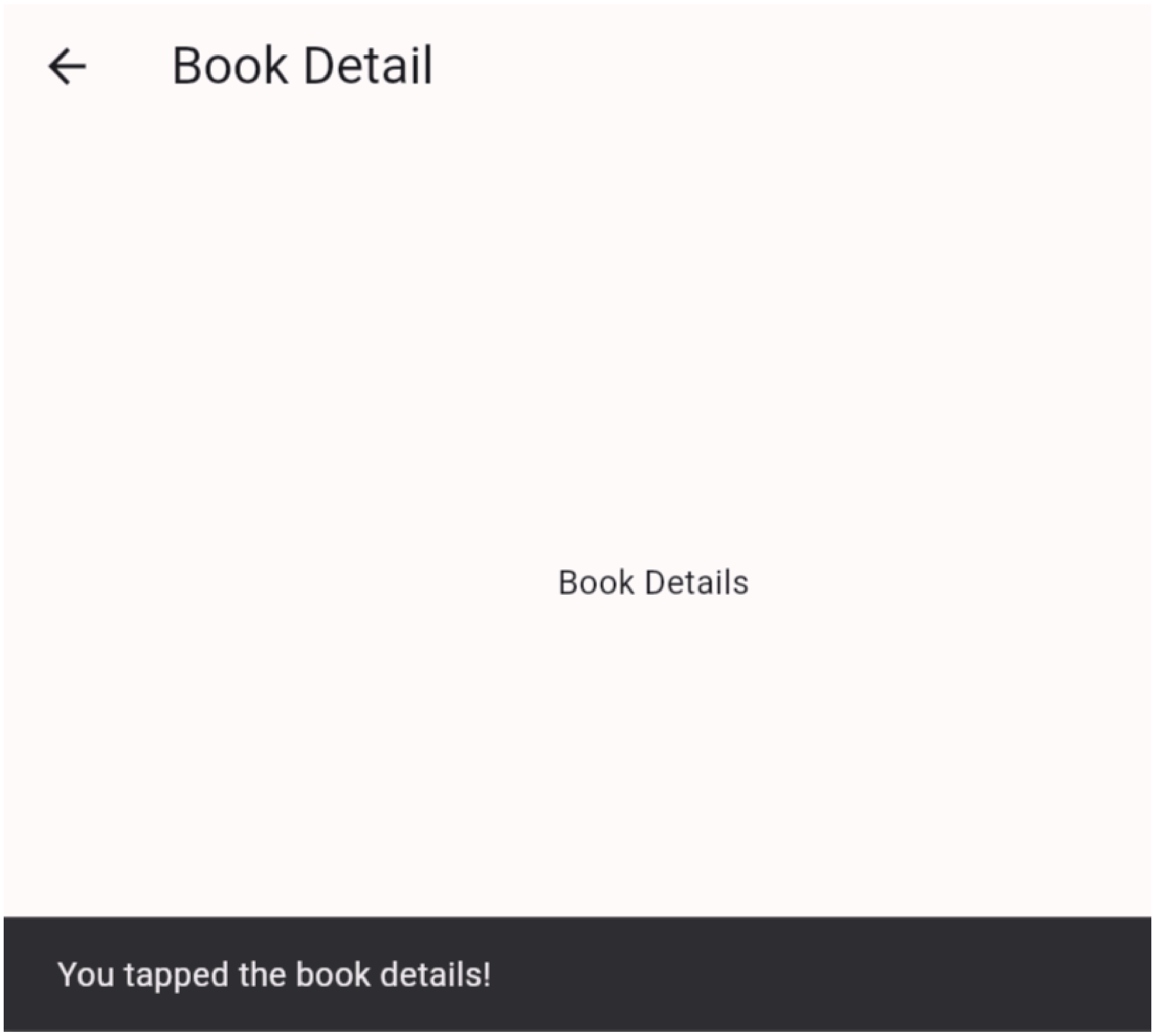
4. Gesture Customization: Optionally, customize the gesture detection behavior by adjusting properties like `behavior`, `dragStartBehavior`, or by combining multiple gesture recognizers.

5. Testing and Refinement: Test the gestures on different devices and screen sizes to ensure a consistent and intuitive user experience. Refine the gesture handling logic as needed to improve usability and responsiveness.

Implementation:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Book Reading App',
      initialRoute: '/',
      routes: {
        '/': (context) => HomePage(),
        '/bookDetail': (context) => BookDetailPage(),
      },
    );
  }
}

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Books'),
      ),
      body: ListView(
        children: [
          ListTile(
            title: Text('Book 1'),
            onTap: () {
              Navigator.pushNamed(context, '/bookDetail');
            },
          ),
          ListTile(
            title: Text('Book 2'),
            onTap: () {
              Navigator.pushNamed(context, '/bookDetail');
            },
          ),
          ListTile(
            title: Text('Book 3'),
            onTap: () {
              Navigator.pushNamed(context, '/bookDetail');
            },
          ),
        ],
      ),
    );
  }
}

class BookDetailPage extends StatelessWidget {
  @override
```

```
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('Book Detail'),
     ),
     body: Center(
       child: GestureDetector(
         onTap: () {
           ScaffoldMessenger.of(context).showSnackBar(
             SnackBar(
               content: Text('You tapped the book details!'),
             ),
           );
         },
         child: Text('Book Details'),
       ),
     ),
   );
 }
}
```



Book Details

You tapped the book details!

Conclusion:
We implemented navigation,routing and gestures in Flutter.
We navigated from Book List page to Book Details page.
Then using gestures we used a snackbar to show that the Book Details text has been clicked on.