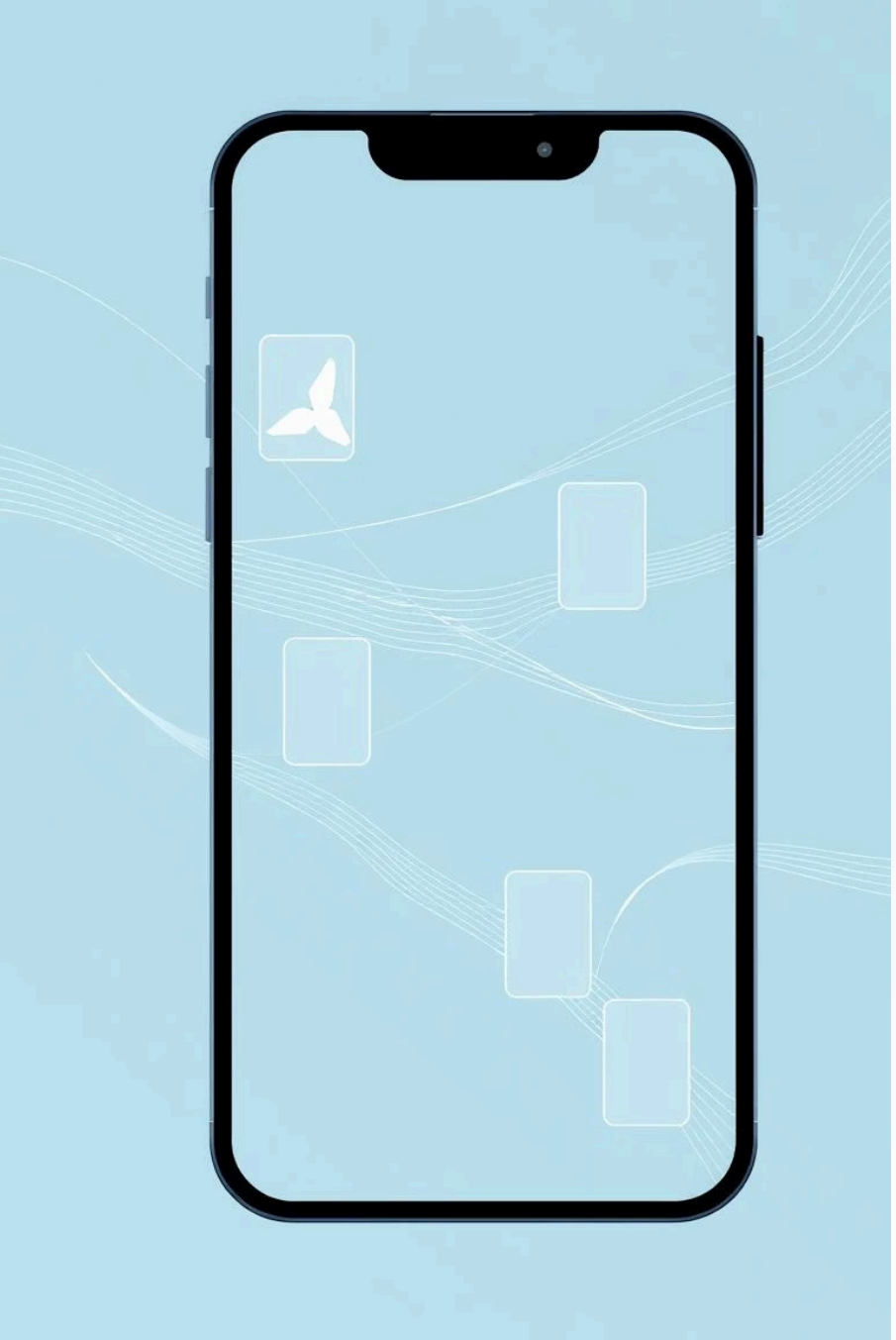


Navigation in Flutter:

what & Why Navigation Matters in Flutter Apps

Navigation is crucial for a smooth and intuitive user experience in any Flutter application. It dictates how users interact with your app's various screens and content.

Seamless User Flow

Navigation enables seamless movement between screens, shaping user experience.

Structured App Flow

Routing organizes app flow, manages screen hierarchy, and supports deep linking.

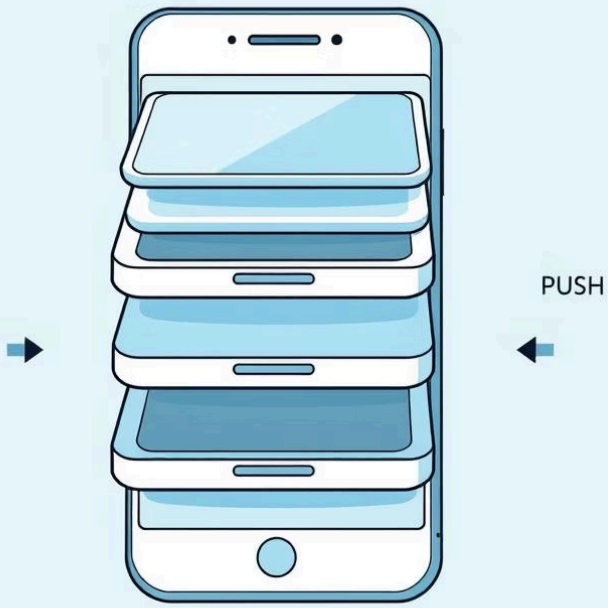
Improved Codebase

Proper navigation architecture improves code modularity, maintainability, and scalability.

The Navigator Widget: Flutter’s Core Navigation Engine

The `Navigator` widget is Flutter's fundamental tool for managing screens. It operates like a stack, where each screen is a "Route" object.

- `push()` adds a new screen on top of the stack.
- `pop()` removes the current screen, revealing the one below it.
- Supports platform-specific transitions: `MaterialPageRoute` (Android) and `CupertinoPageRoute` (iOS).



Example: Navigating to a New Screen

```
Navigator.push(
 context,
 MaterialPageRoute(builder: (_) =>
SecondScreen()),
);
```

Basic Navigation Example: Two Screens and Back

Let's look at a simple scenario: navigating from a first screen to a second, and then returning. This demonstrates the basic `push` and `pop` operations in action.

First Screen (`HomeScreen`)

```
// In HomeScreen widget
ElevatedButton(
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) =>
SecondScreen()),
    );
  },
  child: Text('Go to Second Screen'),
)
```

Second Screen (`SecondScreen`)

```
// In SecondScreen widget
ElevatedButton(
 onPressed: () {
 Navigator.pop(context);
 },
 child: Text('Go Back'),
)
```

task

pushReplacement & pushAndRemoveUntil: Advanced Navigation

Beyond basic navigation, Flutter provides more powerful methods for precise stack management, crucial for complex app flows.

pushReplacement

Replaces the current route with a new one, preventing the user from navigating back to the previous screen. Ideal for login or splash screens.

```
Navigator.pushReplacement(
  context,
  MaterialPageRoute(builder: (_) =>
DashboardScreen()),
);
```

pushAndRemoveUntil

Pushes a new route and removes all existing routes until a specified condition is met. Useful for clearing the entire stack or navigating to a root screen.

```
Navigator.pushAndRemoveUntil(
  context,
  MaterialPageRoute(builder: (context) =>
HomeScreen()),
  (route) => false,
// Removes all previous routes
);
```

task

You will create **3 screens**: (`HomeScreen` (main screen) _ProfileScreen_ `DashboardScreen`)

1) from home sceen navigate to `ProfileScreen` **and replace** the current `HomeScreen`.

2)**From ProfileScreen**: When the user presses the button (Go to Dashboard and remove all previous),

Named Routes

Named routes provide a clear, string-based way to define and manage your app's navigation paths, making code more readable and enabling easier deep linking.

Defining Named Routes

Register routes in your `MaterialApp`'s `routes` map. Each entry maps a string name to a builder function that creates the target screen.

```
MaterialApp(
  routes: {
    '/': (context) => HomeScreen(),
    '/details': (context) => DetailScreen(),
    '/settings': (context) => SettingsScreen(),
  },
)
```

Using Named Routes

Navigate to a screen by its registered name using `Navigator.pushNamed()`. This simplifies navigation calls, especially for frequently accessed screens.

```
// From HomeScreen
ElevatedButton(
  onPressed: () {
    Navigator.pushNamed(context, '/details');
  },
  child: Text('View Details'),
)
```

task

Passing Data Between Screens

Often, you need to send information from one screen to another when navigating. Flutter provides several ways to achieve this, primarily through route arguments.

using Constructor

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

class ScreenOne extends StatelessWidget {
  const ScreenOne({super.key});

  @override
  Widget build(BuildContext context) {
;

    return Scaffold(
      appBar: AppBar(title: const Text('Screen
One')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (context) => ScreenTwo(name:
'Kareem'),
              ),
            );
          },
          child: const Text('Go to Screen Two'),
        ),
      ),
    );
  }
}
```

using Constructor

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

class ScreenTwo extends StatelessWidget {
  final String name;

  const ScreenTwo({super.key, required
this.name});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Screen
Two')),
      body: Center(
        child: Text('Hello, $name!'),
      ),
    );
  }
}
```

task

Configuring Named Routes in MaterialApp

While direct argument passing through the `routes` map is not common, configuring your `MaterialApp` with named routes is a fundamental step towards structured navigation. For passing dynamic data with named routes, you typically use `onGenerateRoute`.

Basic MaterialApp Setup for Named Routes

```
ElevatedButton(
        onPressed: () {
          Navigator.pushNamed(
            context,
            '/second',
            arguments: "karim",
          );
ModalRoute.of(context)!.settings.arguments as String;
```

task

generateRoute()

For more detailed implementation and an example, refer to this GitHub repository:

[Expense App: routes.dart example]

✅ مميزات الطريقة دي:

ميزة	شرح
أكثر مرونة	تقدر تمرر `arguments` بسهولة
مناسبة للمشاريع الكبيرة	بدل ما تحط كل المسارات في `routes: {}`, تكتب logic خاص بيك
تقدر تضيف شروط	زي check لو المستخدم مسجل دخول ولا لأ
تقدر تتعامل مع DI	تمرر Services، ViewModels إلخ