Navigation and Routing

Written by Thapanapong Rukkanchanunt

Outline

+Introduction

+Basic Navigation

+Advanced Navigation Techniques

+Tabs and Tab Navigation

+Deep Linking

+Web URL Strategies

Transition between Pages

- Flutter provides 2 options for transitioning between pages in your application.
- `Navigator` can be used for basic application without complex transition.
- `Router` can be used for deep linking with more complex transition.

Using Navigator

- The `Navigator` widget displays screens as a stack using the correct transition animations for the target platform.
- To navigate to a new screen, access the `Navigator` through the route's `BuildContext` and call imperative methods such as `push()` or `pop()`:

Material Page Route

- Because `Navigator` keeps a stack of `Route` objects, The push() method also takes a `Route` object.
- The `MaterialPageRoute` object is a subclass of `Route` that specifies the transition animations for Material Design.

```
onPressed: () {
  Navigator.of(context).push(
    MaterialPageRoute(
      builder: (context) => const SongScreen(song: song),
    ),
  );
},
child: Text(song.name),
```

Using Named Route

+Applications with simple navigation and deep linking requirements can use the Navigator for navigation and the MaterialApp.routes parameter for deep links:

```
@override
  Widget build(BuildContext context) {
    return MaterialApp(
      routes: {
        '/': (context) => HomeScreen(),
        '/details': (context) => DetailScreen(),
      },
    );
  }
```

Limitations

- Although named routes can handle deep links, the behavior is always the same and can’t be customized.
- When a new deep link is received by the platform, Flutter pushes a new Route onto the Navigator regardless where the user currently is.
- Flutter also doesn’t support the browser forward button for applications using named routes.
- For these reasons, we don’t recommend using named routes in most applications.

Using the Router

+ Flutter applications with advanced navigation and routing requirements (such as a web app that uses direct links to each screen, or an app with multiple `Navigator` widgets) should use a routing package such as `go_router` that can parse the route path and configure the `Navigator` whenever the app receives a new deep link.

Navigator Summary

- `Navigator` handles all builds in stack fashion
- Push will put new widget on top of the stack which is the current display
- Pop will remove widget that is on top of the stack. The next widget will become a new display.
- Your application route will be a sequence of push and pop.

Web Support

- Apps using the `Router` class integrate with the browser History API to provide a consistent experience when using the browser's back and forward buttons.
- Whenever you navigate using the `Router`, a History API entry is added to the browser's history stack. Pressing the back button uses reverse chronological navigation, meaning that the user is taken to the previously visited location that was shown using the Router.
- This means that if the user pops a page from the Navigator and then presses the browser back button the previous page is pushed back onto the stack.

BottomNavigationBar and IndexedStack

- Our app only has one page but can have multiple stack of widgets thanks to `Nagivator`.
- We can have more than one page organized by some sort of nagivation bar; `BottomNavigationBar` to be precise.
- `IndexedStack` widget will keep track of which page will be shown at the top.

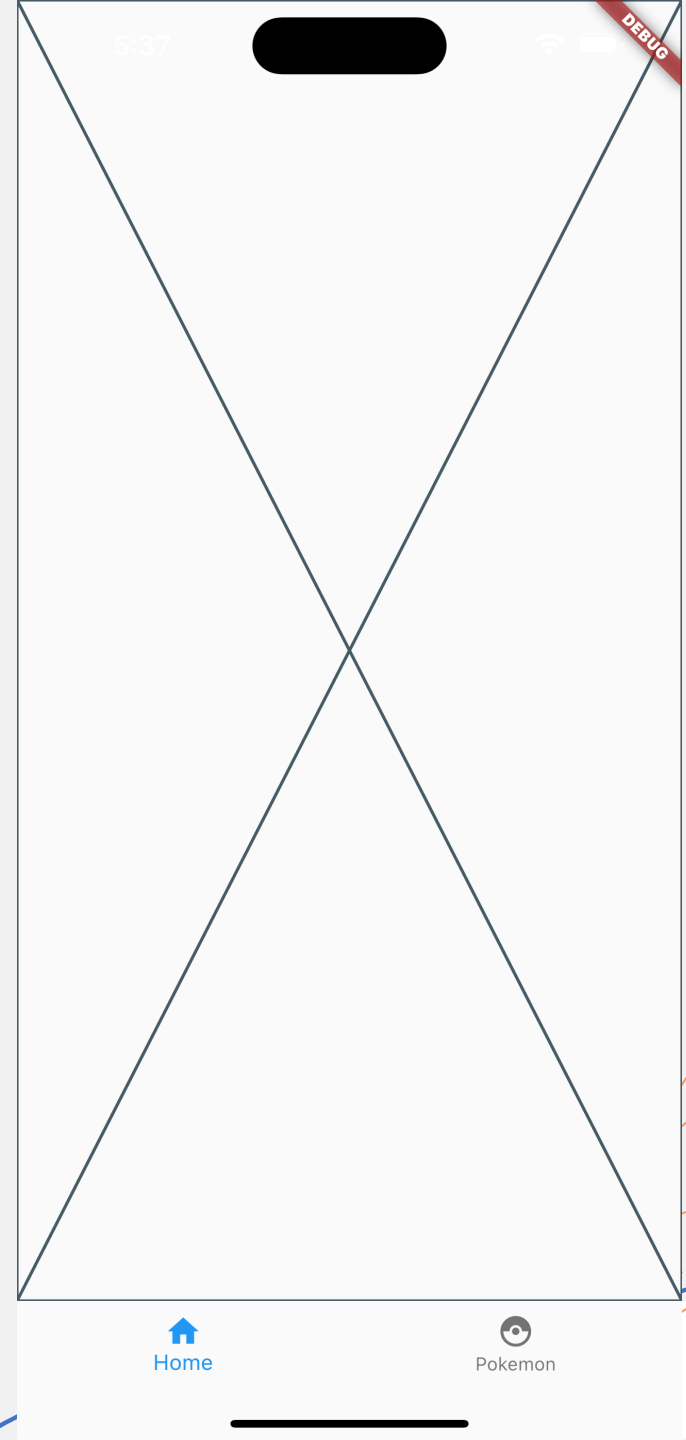
Creating Buttom Navigation Bar

+ `BottomNavigationBar` needs to keep track of which page is being selected so we will use `StatefulWidget`
+ Create a new file called `bottom_navigation.dart` and write a `MyBottomNavigation` class. Notice that the new file is automatically imported in `main.dart`.
+ Also change the home screen in `main.dart` to this class.

```
+home: const MyBottomNavigation()
```

MyBottomNavigation

```
Widget build(BuildContext context) {
  return Scaffold(
    body: const Placeholder(),
    bottomNavigationBar: BottomNavigationBar(
      items: const [
        BottomNavigationBarItem(
            icon: Icon(Icons.home),
            label: 'Home'
        ),
        BottomNavigationBarItem(
            icon: Icon(Icons.catching_pokemon),
            label: 'Pokemon'
        ),
      ],
    ),
  );
}
```

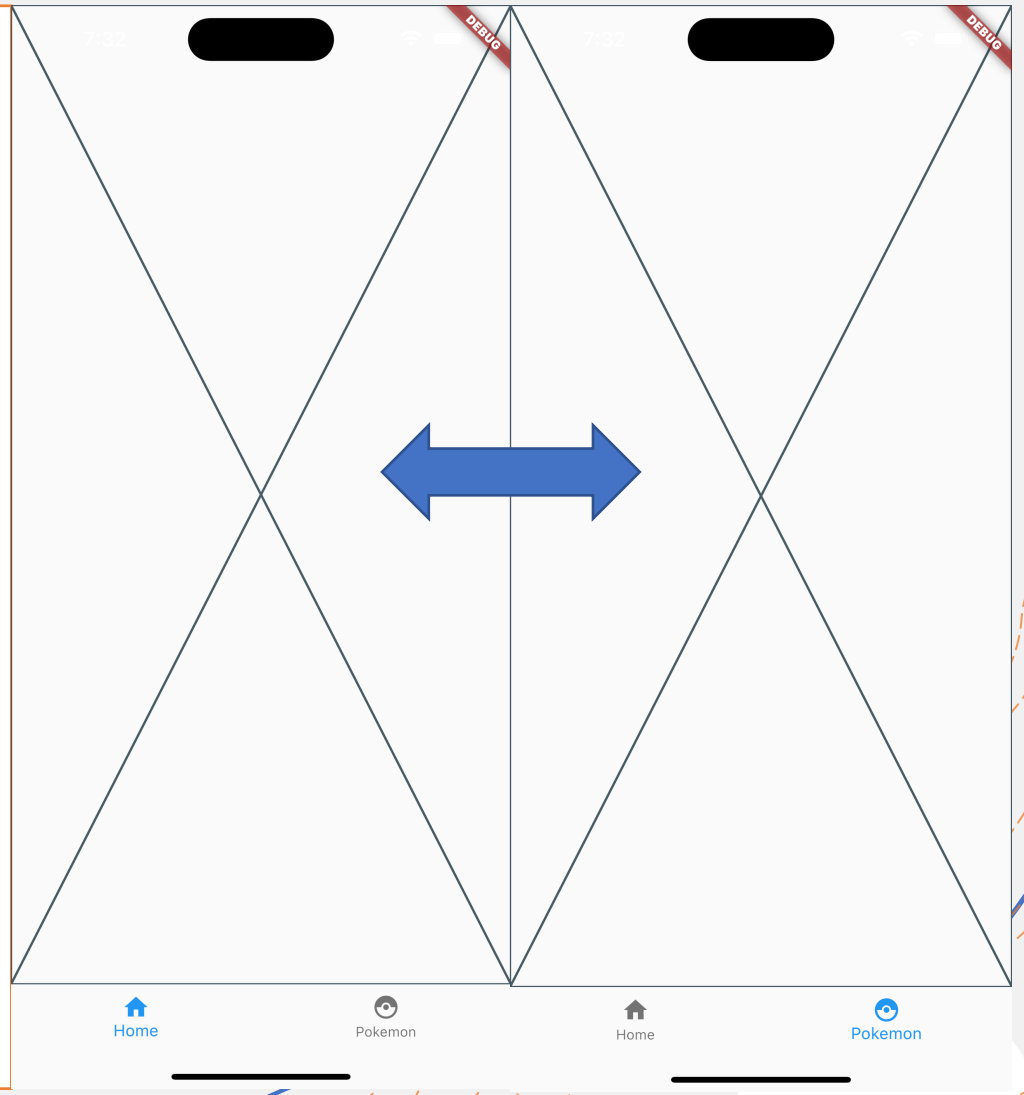


Keeping State Variables

- We introduct the private variable `_currentIndex` to keep track of what page we are current on.
- We then use `currentIndex` and `onTap` parameter inside `BottomNavigationBar` widget.
  - `currentIndex` parameter tells the app which page you are on so that the corresponding icon is highlighted.
  - `onTap` parameters will take a function whose parameter is the index of icon the user is tapping.
  - Remember that `index` always starts from 0.

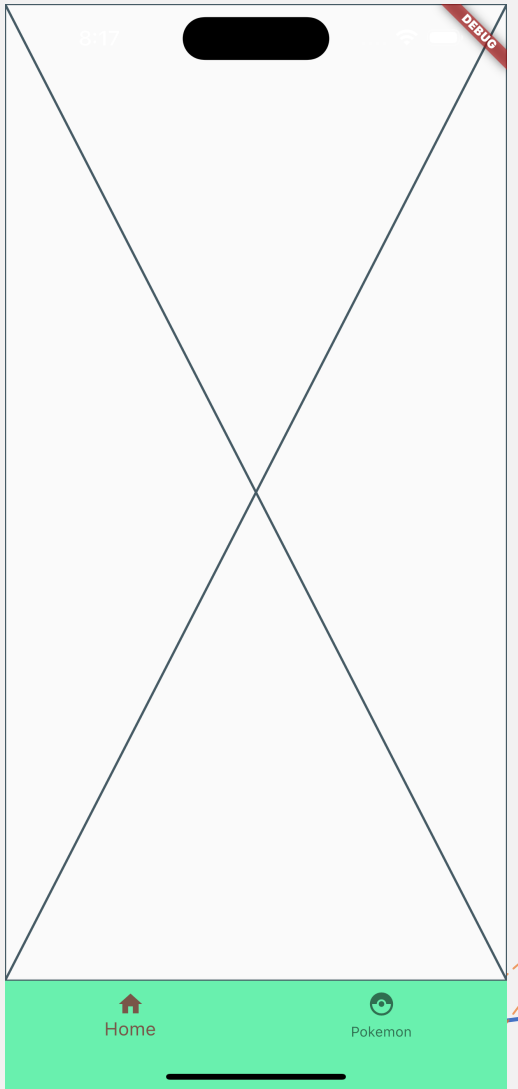
Current Index Setup

```
int _currentIndex = 0;
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: const Placeholder(),
    bottomNavigationBar: BottomNavigationBar(
      currentIndex: _currentIndex,
      onTap: (index) {
        setState(() {
          _currentIndex = index;
        });
      },
      items: const [...],
    ),
  );
}
```



BottomNavigationBar Customization

+ backgroundColor
+ selectedItemColor
+ selectedIconTheme
+ selectedFontSize
+ And many more...



IndexedStack Summary

- Notice that `children` is `const`. This will let flutter know that even if we change state (change page), the inactive pages will not be re-rendered, keeping their current state.
- Together with `Navigator`, we should be able to navigate through your app as you desire.