

Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 2
Slivers

Lecture 5



Chapter

05

Table Of Content

- Navigator
- Slivers



In Flutter, navigation allows you to move between different screens or routes in your app. Understanding how navigation works is fundamental for building apps with multiple screens or dynamic content.



Introduction to Flutter Navigation

Flutter uses a **Navigator** widget to manage routing and navigation. This widget is responsible for managing a stack of pages (or routes) that the user can navigate between.

A route is typically a screen or page in your app. The most basic form of navigation in Flutter involves pushing a new route and popping the current one, off the stack.



Types of Navigation

In Flutter, there are two main types of navigation



Imperative Navigation

Imperative navigation means you push and pop routes programmatically. You decide when and which route to navigate to based on user actions or other events.

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



Declarative Navigation

Declarative navigation allows Flutter to manage the navigation state by defining routes declaratively using `Navigator 2.0`. It is used for more complex navigation scenarios, such as apps with deep linking.



Navigating Between Screens (Routes)

In Flutter, routes represent pages in your app. The simplest way to navigate between routes is by using the `Navigator` widget. There are two primary ways to define and manage routes:

- **Anonymous Routes**
- **Named Routes**



Anonymous Routes

An anonymous route is one defined inline when calling `Navigator.push()`. It's a simple and quick way to navigate to a new screen.

```
Navigator.push( context,
```

```
MaterialPageRoute(builder: (context) => SecondScreen()),);
```



***MaterialPageRoute** is a built-in Flutter class that provides a standard page transition for Material Design apps. It is used with `Navigator.push()` to push a new screen (or route) onto the navigation stack. The transition for this route is typically a slide from right to left on mobile devices.*

*The **MaterialPageRoute** constructor takes a **builder** function, which returns the screen (widget) to navigate to.*



Chapter 05

Navigator

Named Routes

Named routes are defined in your app's `MaterialApp` widget. You can navigate to named routes using the

`Navigator.pushNamed()` method.

```
// Define named routes in MaterialApp
MaterialApp(
  initialRoute: '/',
  routes: {
    '/': (context) => FirstScreen(),
    '/second': (context) => SecondScreen(),
  },
);
```

```
// Navigate to a named route
```

```
Navigator.pushNamed(context, '/second');
```



MaterialApp should be placed at the root of your widget tree, typically in the `main()` function. It acts as the entry point to your app and manages routing and theme.



Passing Data Between Screens

Passing data while navigating is a common requirement. Flutter provides two ways to pass data:

- **Via constructor** when navigating to a screen.
- Using `Navigator's arguments parameter` for named routes.



Passing Data via Constructor

You can pass data directly when you push a new route:

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => SecondScreen(data: 'Hello from First Screen'),  
  ),  
  
);
```

Note — With **named routes**, You cannot pass data directly via constructors. This is only for **Anonymous**

route(MaterialPageRoute).



```
class SecondScreen extends StatelessWidget {  
  final String data;
```

```
  SecondScreen({required this.data});
```

```
  @override
```

```
  Widget build(BuildContext context) {
```

```
    return Scaffold(
```

```
      body: Center(child: Text(data)),
```

```
    );
```

```
  }
```

```
}
```



Passing Data Using arguments

With named routes, you can use the arguments parameter to pass data:

```
Navigator.pushNamed(context, '/second', arguments: 'Hello from First Screen',);
```



Chapter 05

Navigator

In the receiving screen, use `ModalRoute.of(context)?.settings.arguments` to access the passed data:

```
class SecondScreen extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    final String data = ModalRoute.of(context)?.settings.arguments as String;  
  
    return Scaffold(  
      body: Center(child: Text(data)),  
    );  
  }  
}
```



Chapter 05

Navigator

Handling Route Parameters

You might need to pass dynamic parameters (e.g., IDs or search queries). For this, you can define routes with parameters and extract the parameters from the route settings. We use `onGenerateRoute` in place of `route`.

`onGenerateRoute` is used when you need **dynamic route handling**, such as:

- Passing arguments to routes.
- Handling routes that might change based on conditions (e.g., query parameters, user-specific data).
- Catching unrecognized routes to provide custom handling or error screens.



Advanced Navigation: Navigator 2.0

Navigator 2.0 allows you to manage the navigation stack more declaratively. This is helpful for complex scenarios like deep linking, custom back navigation, or nested navigation.



Chapter 05

Navigator

Best Practices

Here are some best practices to follow when working with navigation in Flutter:

- **Keep your routes organised:** Use a centralised route management system (e.g., using a class or a package like `app_router`) to keep routes in one place.
- **Avoid deep nesting of navigation:** If your app has complex screens, try to use nested navigators instead of deeply nested routes.
- **Use named routes for consistency:** Named routes provide better code clarity, especially in larger apps.
- **Handle back navigation properly:** When using Navigator 2.0 or deep navigation, ensure that the back navigation behavior is intuitive.
- **Use Route Guards:** For handling authorization and authentication, ensure that you block routes that require a certain state.



Slivers

What is SliverList Class in Flutter App Development?

Slivers offer you the extraordinary view of lists when they are scrolling.

It can also let you impact the grids and lists scrolling experience. The features of `sliver_tools` offers are:

- SliverStack
- MultiSliver
- SliverAnimatedPaintExtent
- SliverClip



The slivers must be wrapped entirely around CustomScrollView to offer you better scrolling effects to grids or lists. CustomScrollView wraps Slivers widget can include all slivers that you are creating.

1. MultiSliver

MultiSliver() widget lets you wrap multiple slivers together effectively.



```
Scaffold(  
  appBar: AppBar(title: const Text("Flutter Silver Class Sample")),  
  body: CustomScrollView(  
    slivers: [  
      MultiSliver(  
        pushPinnedChildren: true,  
        children: [  
          SliverPinnedHeader(  
            child: Container(  
              color: Colors.yellow,  
              height: 100,  
              child: const Text(  
                "I am a Pinned Header",  
                style: TextStyle(fontSize: 30),  
              )),  
        ],  
      ),  
    ],  
  ),  
)
```




2. SliverGrid

SliverGrid() can effectively put such items according to the 2D system. Here it needs gridDelegate that can structure the delegate and grid layout that passes items on screen for scrolling purposes. Here the parameter childCount is considered to be the total number of things that are required to be displayed.



```
Scaffold(  
  appBar: AppBar(title: const Text("Flutter Silver Class Sample")),  
  body: CustomScrollView(  
    slivers: [  
      SliverGrid(  
        gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(  
          crossAxisCount: 3,  
          mainAxisSpacing: 10,  
          crossAxisSpacing: 10,  
          childAspectRatio: 2.0,  
        ),  
        delegate: SliverChildBuilderDelegate(  
          (context, index) {  
            return Container(  
              height: 20,  
              color: Colors.amber[100],  
              alignment: Alignment.center,  
              child: Text(index.toString()),  
            );  
          },  
          childCount: 41,  
        ),  
      ),  
    ],  
  ),  
),
```

Flutter Silver Class Sample

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	16	17
18	19	20
21	22	23
24	25	26
27	28	29
30	31	32

SliverList Class used in Flutter app development

The SliverList Class variables are entities for the data, and it is the name used to store the data in the system. The server-side flutter app development languages are beneficial to script the code with the web pages more effectively.

The Flutter app development uses the various regulations used by many Flutter app development company like us. Among the most effective used in the flutter platform is using the SliverList Class variables used codings.



For utilizing the server script programming codes in the flutter programming platforms, the SliverList Class is a syntax used to embed the server code for the web page applications effectively. The dynamic web contents in the programming languages are created using the server scripting codes, where the web page is written to the browser coding.



Impact of effective SliverList Class server codes:

The flutter app development is used to make the effective development of the web page applications on the flutter programming platforms, where they are using the control similar to the applications that are used with the use of the Flutter app development.

To use the dynamic web page with the flutter app development, you must learn the use of the SliverList Class syntax in the server script codes. There are some rules that need to be followed for using the SliverList Class variables syntax in the programming codes. They are listed below.



Variables in SliverList Class languages:

The SliverList Class variables in the flutter are the specific SliverList Class used to store and retrieve the data using the programming codes.

The access capability to use any characters or the sentences as the variables by the program developers, but the variables should get initiated with the use of any alphanumeric characters, and they do not include the usage of any reserved or whitespace characters in it.

The programmer should need to vary based on declaring what type of data it is, where based on the usage of the SliverList Class, only the data get stored in the program. Based on the SliverList Class, the compiler only evaluates the data partitions to store the data for executing the programs.



Most compelling features:

For example, if you want to keep the data as the integer and use the variable as the student, then it should be declared as the int student. Even with the use of the integer set of the variables, you can use the date or some other whole number elements more effectively.

The most compelling features of using programming with the SliverList Class are they easily conclude what type of data is used to process the variable used for the applications. Sometimes, you need to specify the type of data in the SliverList Class, where it is occasional.



For example, suppose you are pasting the keyword in the programming code. You don't need to use the SliverList Class to use the program's data in the flutter SliverList Class variables programming platforms. Some other forms of using the variables in the flutter framework are presented here.

Most of the time, the inputs are received in the form of the string SliverList Class; even the data is used in the entered number as the input. So the converting SliverList Class is getting used for the string SliverList Class to some other data.



Live Coding

What is Next ?



Thank You