ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

502071

Cross-Platform Mobile Application Development

Material Navigation Widgets

1

MaterialApp widget

- The MaterialApp widget is a top-level widget in Flutter that provides a standard design and layout for Android and iOS applications. It sets up the basic structure of a Material Design app, including the status bar, navigation bar, and other common UI elements.
  - **Theme:** The MaterialApp widget provides a default theme for the entire app, which can be customized using the theme property. The theme defines the colors, fonts, and other design elements used throughout the app.
  - **Title:** The MaterialApp widget sets the title of the app, which is displayed in the app switcher and other system UI elements.
  - **Home:** The MaterialApp widget sets the home screen of the app, which is the first screen that is displayed when the app is launched. The home screen can be any widget, such as a Scaffold or a Container.
  - **Routing:** The MaterialApp widget provides a routing mechanism for navigating between different screens or pages within the app. The routes property can be used to define a mapping of named routes to screen widgets

MaterialApp widget

- **Localizations**: The MaterialApp widget provides support for localizing the app, which allows the app to display text and other content in different languages or regions. The localizationsDelegates and supportedLocales properties can be used to specify the set of supported languages and the widgets that provide the translations.
- **Debug Banner**: The MaterialApp widget displays a debug banner in the top right corner of the screen by default, which can be useful during development. This banner can be disabled using the debugShowCheckedModeBanner property.
- **Other Properties**: There are many other properties of the MaterialApp widget, including navigatorKey for managing the Navigator widget, onGenerateRoute for handling custom routing logic, and builder for wrapping the app in a higher-order widget.

```
void main() {
    runApp(const MaterialApp(home: MyApp(),
        debugShowCheckedModeBanner: false
    ));
}
```

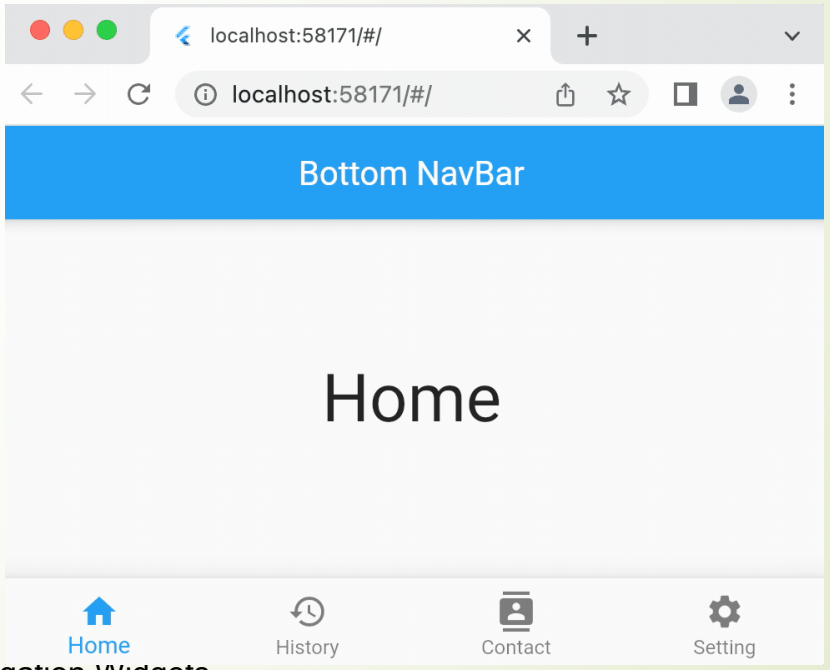
Scaffold widget

- The Scaffold widget is a pre-designed screen structure that provides a standard layout for an app screen. It serves as a basic framework for building material design apps and allows developers to quickly create a UI with commonly used elements like app bars, drawers, bottom navigation and floating action buttons.
- The Scaffold widget is commonly used as the root widget of an app screen and can be nested within other widgets to create more complex layouts. It provides a streamlined approach to app development and helps maintain consistency across different screens and sections of an app.

Bottom Navigation

Bottom Navigation Bar

- The BottomNavigationBar widget is a built-in widget in Flutter that provides a navigation menu at the bottom of the screen. It is commonly used in mobile applications to allow users to switch between different views or screens.
- A bottom navigation bar is usually used in conjunction with a Scaffold, where it is provided as the Scaffold.bottomNavigationBar argument.



Bottom Navigation Bar

- To use the BottomNavigationBar widget in your Flutter app, you need to follow these steps:
  - Create a list of BottomNavigationBarItem widgets, each representing a navigation item:

```
List<BottomNavigationBarItem> _items = [
    BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home',),
    BottomNavigationBarItem(icon: Icon(Icons.search), label: 'Search',),
    BottomNavigationBarItem(icon: Icon(Icons.person), label: 'Profile',),
];
```

  - Create a BottomNavigationBar widget and pass in the list of navigation items:

```
Scaffold(
    bottomNavigationBar: BottomNavigationBar(items: _items, onTap: _onTap),
    appBar: AppBar(title: const Text('Basic Form')),
    body: Container(…),
)
```

Bottom Navigation Bar

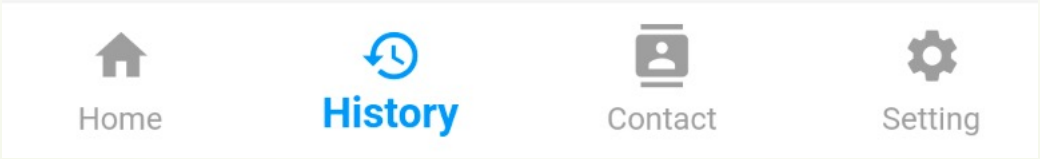
- Flutter's BottomNavigationBar widget is highly customizable, and there are various ways to customize it to match the design of your app. Here are some common ways to customize the BottomNavigationBar:
  - **Change the colors**: You can change the background color, the color of the selected item, and the color of the unselected items

```
BottomNavigationBar(
    backgroundColor: Colors.white,
    selectedItemColor: Colors.blue,
    unselectedItemColor: Colors.grey,
)
```

Bottom Navigation Bar

- Change the font style: You can change the font size, color, and weight of the labels.

```
BottomNavigationBar(
    backgroundColor: Colors.white,
    selectedItemColor: Colors.blue,
    unselectedItemColor: Colors.grey,
    selectedLabelStyle: TextStyle(
        fontSize: 16,
        color: Colors.black,
        fontWeight: FontWeight.bold,
    )
)
```

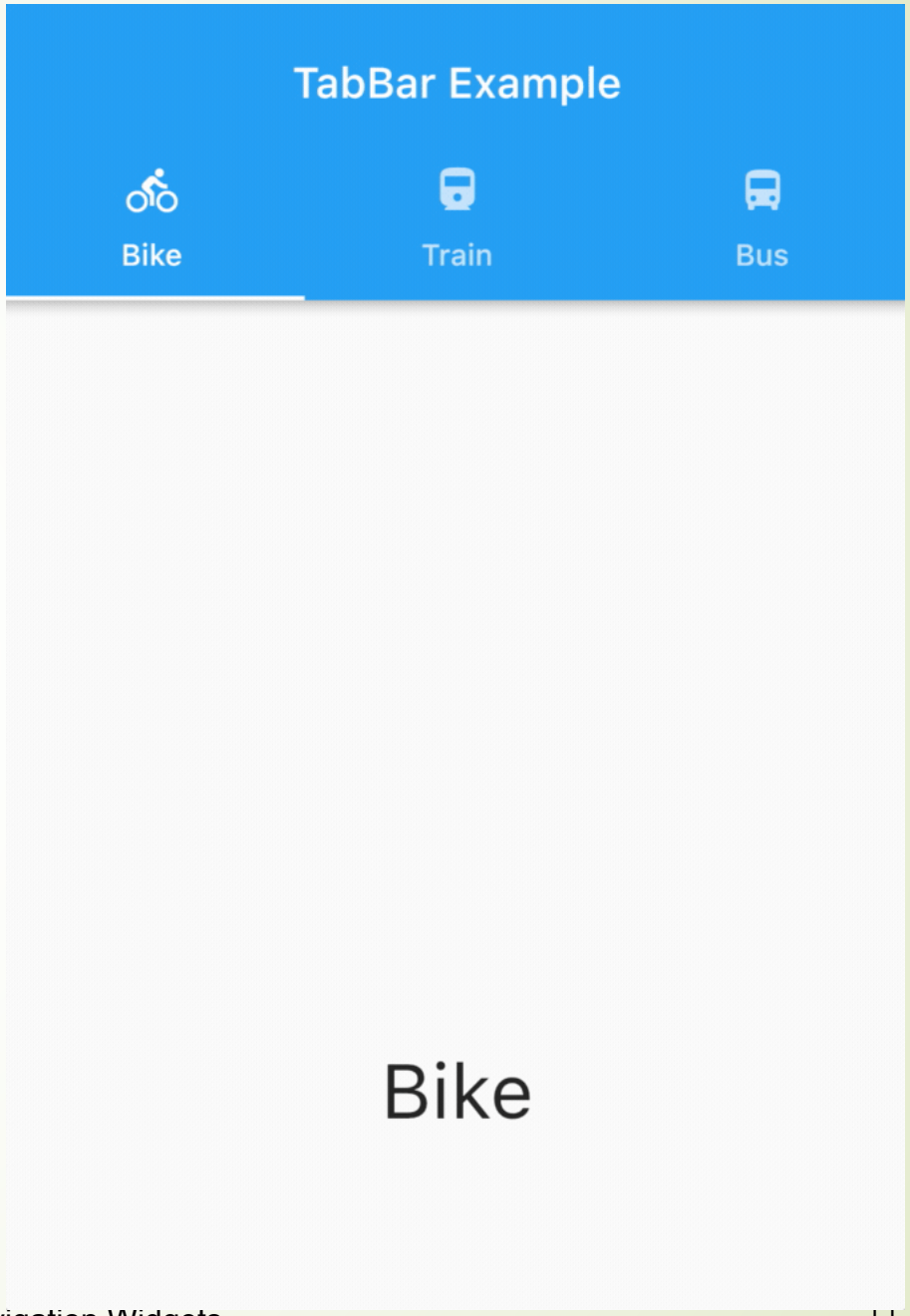


TabBar

TabBar widget

- The TabBar widget in Flutter is used to create a set of tabs that can be switched by the user. It can be used in combination with the TabBarView widget to display different content for each tab.

- Working with tabs is a common pattern in apps that follow the Material Design guidelines. Flutter includes a convenient way to create tab layouts as part of the material library.



TabBar widget

- Designing a TabController
  - The TabController as the name suggests controls the functioning of each tab by syncing the tabs and the contents with each other. The DefaultTabController widget is one of the simplest ways to create tabs in flutter.

```
DefaultTabController(
    length: 3,
    child: Scaffold(
        appBar: AppBar(
            title: const Text('TabBar Example'),
            bottom: const TabBar(tabs: […]),
        )
        body: TabBarView(children: [
            _createTabbarView('Bike'),
            _createTabbarView('Train'),
            _createTabbarView('Bus')
    ],)
```

TabBar widget

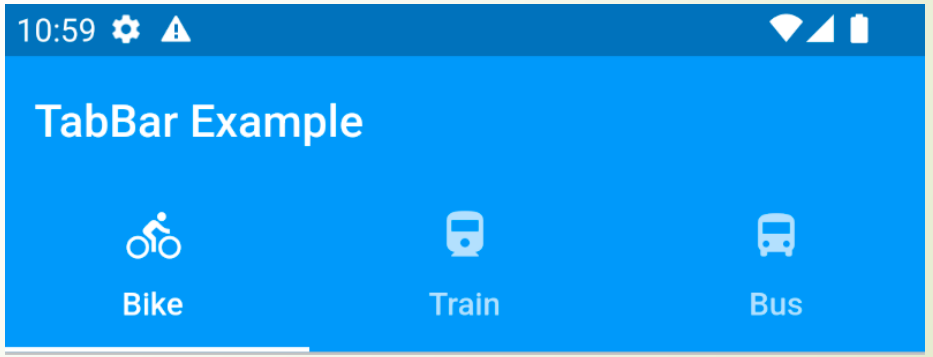
- In this example, the DefaultTabController widget is wrapping the Scaffold widget. It's also setting the length parameter to 3, which tells the TabBar and TabBarView widgets how many tabs to create.
- By using a DefaultTabController, you don't have to create and manage the TabController yourself. The DefaultTabController automatically creates a TabController and passes it down to its descendants via the InheritedWidget mechanism. This makes it easier to manage the state of the TabBar and TabBarView and eliminates the need for boilerplate code.

```
DefaultTabController(
    length: 3,
    child: Scaffold()
)
```

TabBar widget

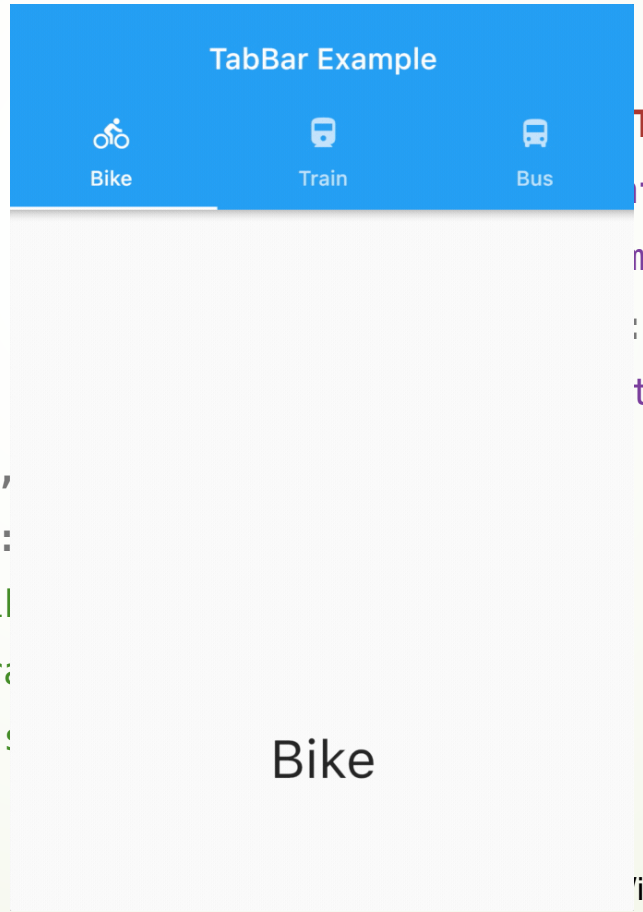
- Adding tabs: A tab in Flutter can be created using a TabBar widget as shown below

```
Scaffold(
    appBar: AppBar(
        title: const Text('TabBar Example'),
        bottom: const TabBar(tabs: [
            Tab(text: 'Bike',icon: Icon(Icons.directions_bike),),
            Tab(text: 'Train',icon: Icon(Icons.directions_train),),
            Tab(text: 'Bus',icon: Icon(Icons.directions_bus),)
        ]),
    )
    body: TabBarView(…)
)
```



TabBar widget

- Adding content to tabs: The TabBarView widget can be used to specify the contents of each tab. For the sake of simplicity we will display the icons in the tab as the content of the tab as shown below.




```
AppBar(
    title: const Text('TabBar
    bottom: const TabBar(...),
    body: TabBarView(children:
        _createTabbarView('Bi
        _createTabbarView('Tr
        _createTabbarView('Bu
    ]),
)
```

```
TabbarView(String label) {
tainer(
ment: Alignment.center,
: Text(label,
tyle: const TextStyle(fontSize: 36),
```

TabController

- In Flutter, there are two options to control a TabBar and a TabView: the DefaultTabController widget and the TabController object.
  - DefaultTabController: This is a convenient way to link a TabBar and a TabView together without creating a TabController object manually.
  - TabController: If you need more control over the TabBar and the TabView, you can create a TabController object manually. This allows you to customize the behavior of the TabController, such as adding a listener to listen for tab changes. With the TabController, you need to set the controller property of the TabBar and TabView widgets to the TabController object.

TabController

```
DefaultTabController(
    length: 3,
    child: Scaffold(..)
)
```

```
late TabController _controller =
                        TabController(length: 2, vsync: this);

Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: const Text('TabBar Example'),
            bottom: TabBar(controller: _controller, tabs: [..]),
        ),
        body: TabBarView(
            controller: _controller,
            children: [..]),
    );
}
```

TabController

- When initializing a TabController, you need to specify the number of tabs using the length parameter. For example, TabController(length: 3) creates a TabController object with three tabs.
- The vsync parameter specifies the object that synchronizes the animation of the TabBar and the TabView. Typically, you would pass this if your widget implements the TickerProviderStateMixin.

```
class _MyAppState extends State<MyApp> with SingleTickerProviderStateMixin {
    late TabController _controller = TabController(length: 2, vsync: this);
    void dispose() {
        _controller.dispose();
        super.dispose();
    }
    Widget build(BuildContext context) {return Scaffold(…);}
}
```

TickerProviderStateMixin

- TickerProviderStateMixin is a mixin that provides a TickerProvider to the state of a widget. A TickerProvider is an object that provides a way to create an animation Ticker that can be used to animate a widget.
- A Ticker is a class that manages a sequence of animation frames, called ticks, and notifies listeners each time a new frame is available. Animations created with Ticker can be used to create smooth and responsive animations, and they are commonly used to animate the state of a widget over time.
- To use Ticker and create animations in Flutter, you need to pass a TickerProvider to the animation controller. The TickerProviderStateMixin is a convenient way to provide a TickerProvider to the state of a widget.

TabController

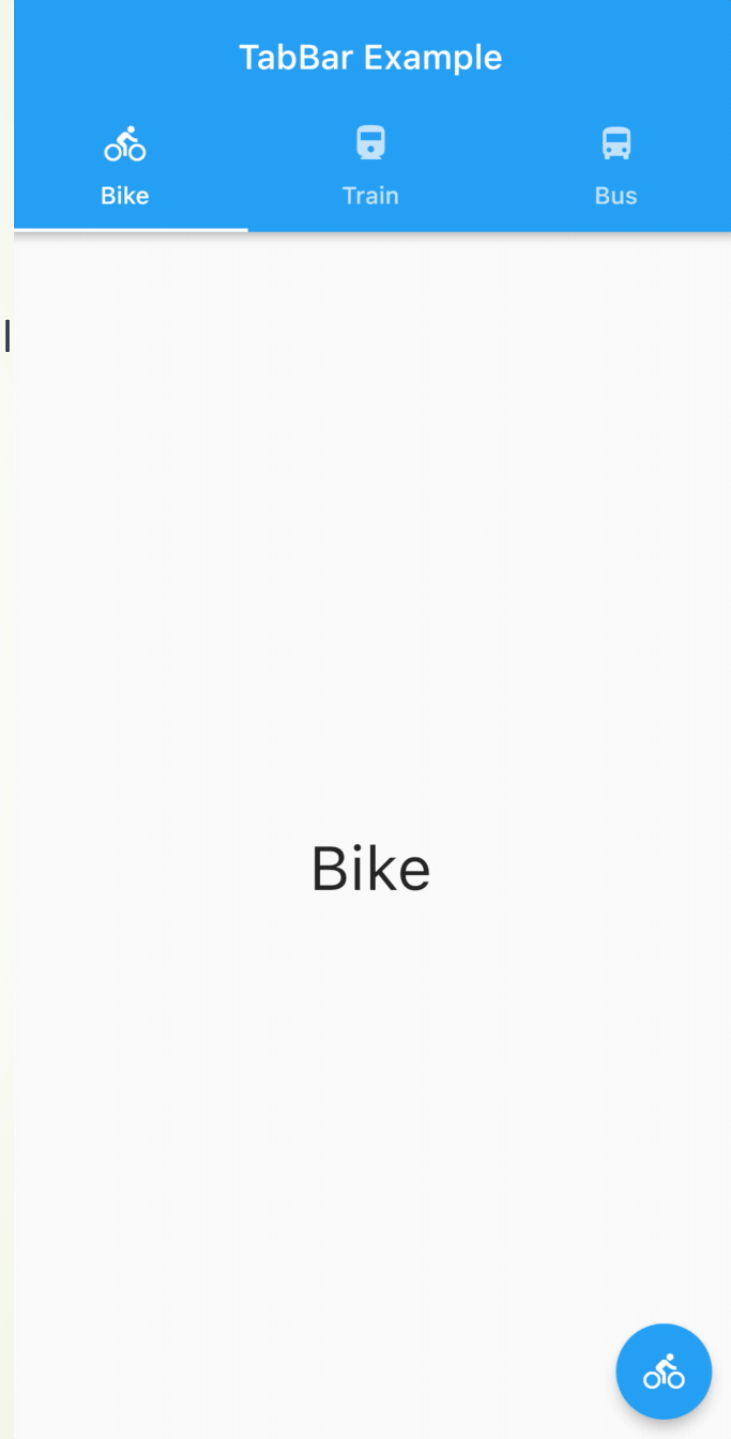
- We also call _tabController.dispose() in the dispose method to clean up the resources used by the TabController object when the widget is removed from the widget tree.
- It's important to call dispose() on the TabController when it's no longer needed to avoid memory leaks and to stop any running animations that may cause performance issues.

```
class _MyAppState extends State<MyApp> with SingleTickerProviderStateMixin {
    late TabController _controller = TabController(length: 2, vsync: this);
    void dispose() {
        _controller.dispose();
        super.dispose();
    }
    Widget build(BuildContext context) {return Scaffold(…);}
}
```

TabController

- The TabController class provides a method called animateTo, which all tab by specifying its index.

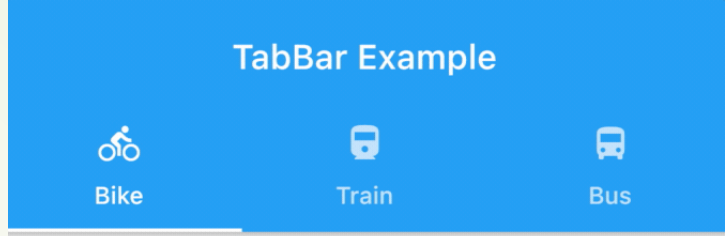
```
void _changeTab(int index) {
    _tabController.animateTo(index);
}

floatingActionButton: FloatingActionButton(
    onPressed: () => _changeTab(1),
    child: Icon(Icons.add),
),
```



TabController

- One of the useful methods of the TabController class is addListener, v
  listener that will be called whenever the value of the TabController cha
  a listener to a TabController is as follows

```
_controller.addListener(() {
    // do something when the tab changes
    if (!_controller.indexIsChanging) {
        print(_controller.index);
    }
});
```

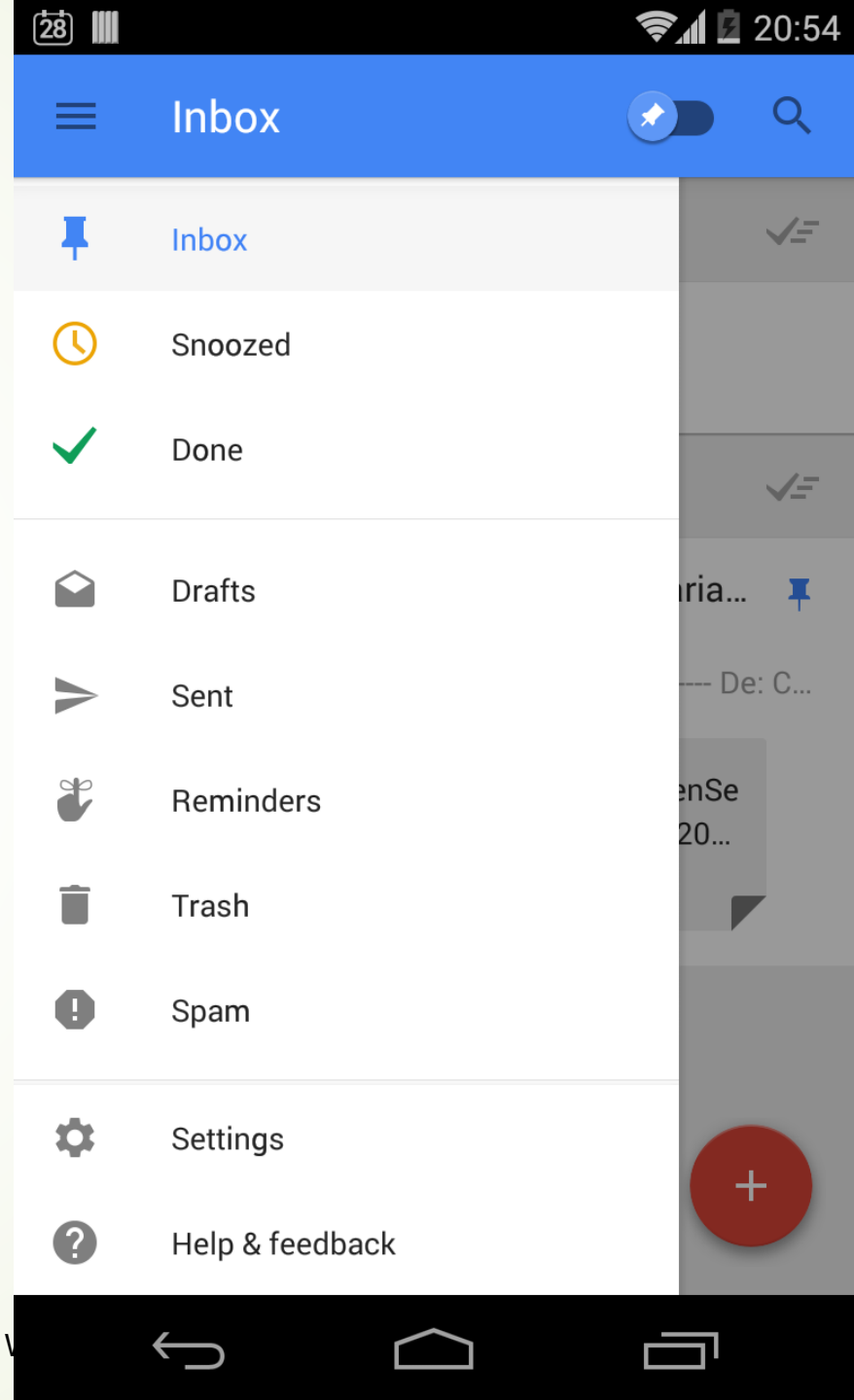


Bike

Drawer

Drawer widget

- a Drawer widget is a panel that slides in from the edge of the screen, usually from the left or right, and provides a space for additional functionality such as navigation menus or settings.
- The Drawer is commonly used in mobile applications to provide a simple and convenient way for users to access app features and settings.



Drawer widget

- To use a Drawer widget in your Flutter app, you need to add a Drawer widget as a child of the Scaffold and specify the content that will be displayed inside the Drawer.

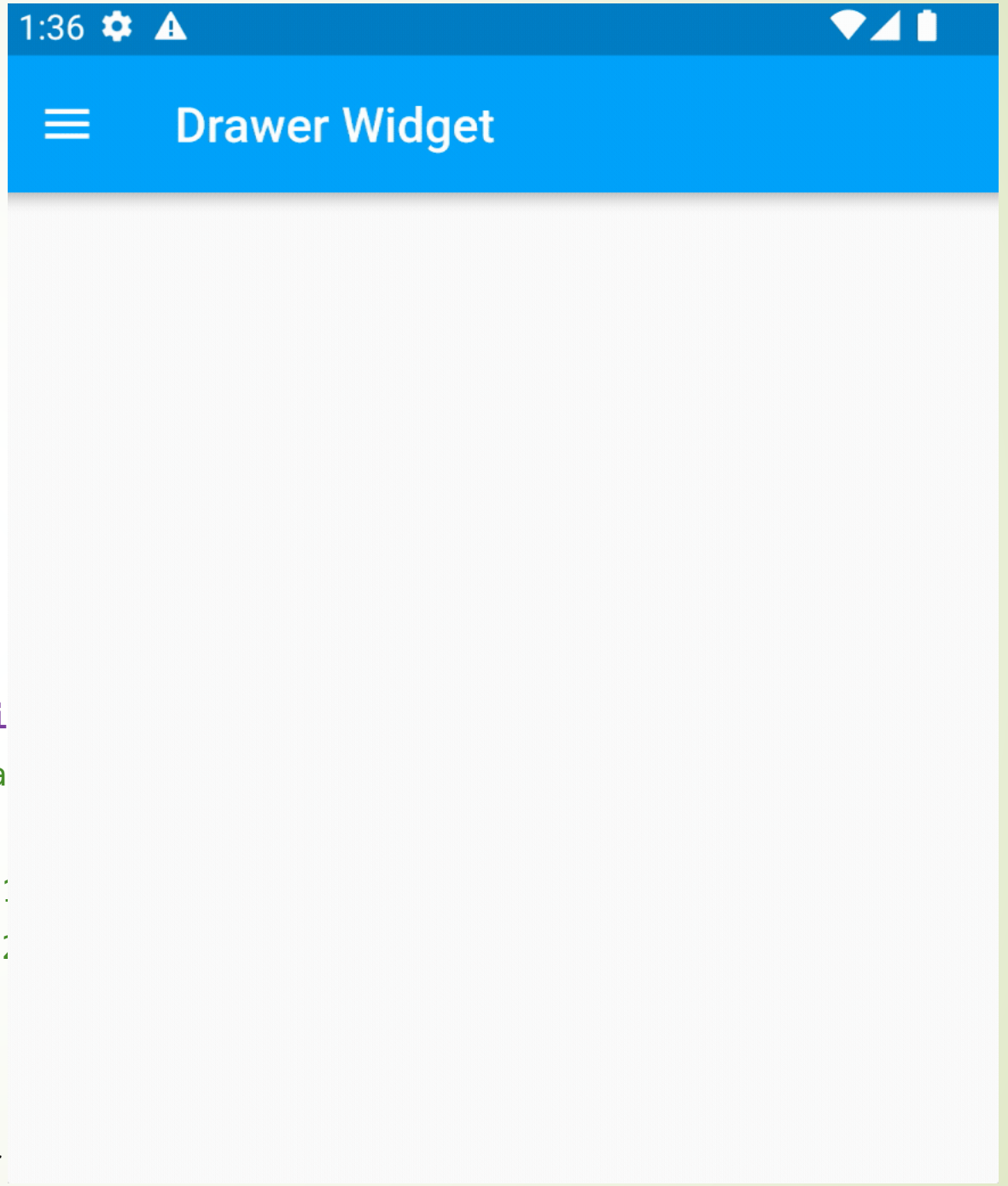
```
Scaffold(
    drawer: Drawer(
        child: Container(), // put whatever widget here
    ),
    body: Container(),
)
```

- A drawer can be any widget, but it's often best to use the Drawer widget from the material library, which adheres to the Material Design spec.

Drawer widget

- Inside the Drawer widget, you can add your menu items as a ListView.

```
Drawer(
    child: ListView(
        padding: EdgeInsets.zero,
        children: [
            DrawerHeader(
                decoration: BoxDecorati
                child: Text('Drawer Hea
            ),
            ListTile(title: Text('Item 1
            ListTile(title: Text('Item 2
        ],
    ),
)
```



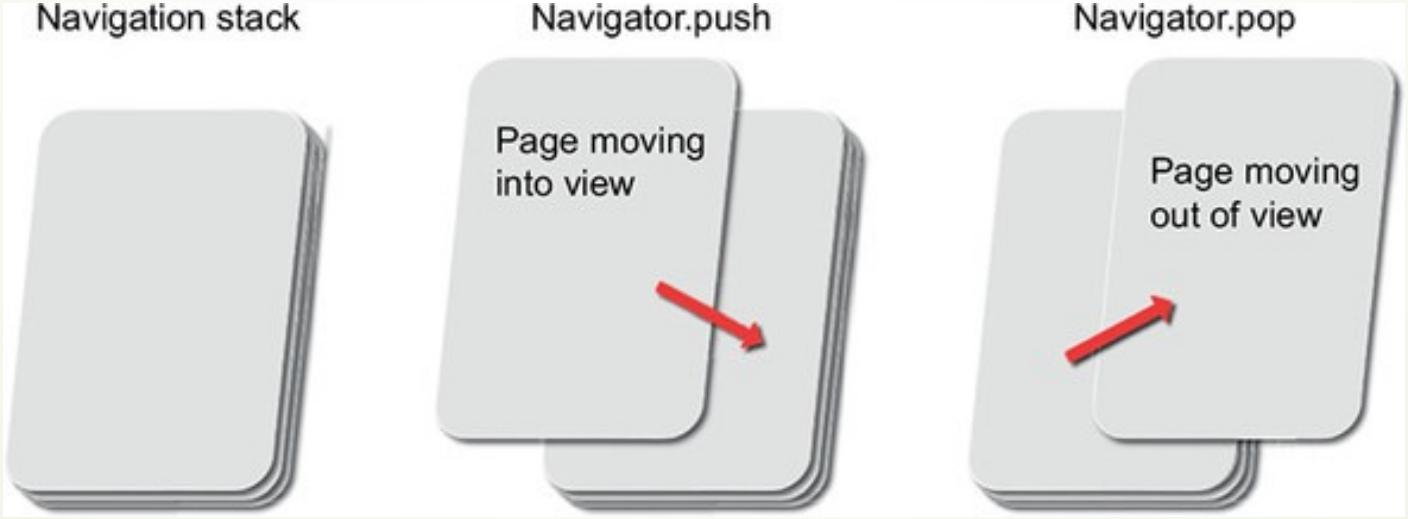
Drawer widget

- Close the drawer programmatically
  - After a user taps an item, you might want to close the drawer. You can do this by using the Navigator.
  - When a user opens the drawer, Flutter adds the drawer to the navigation stack. Therefore, to close the drawer, call Navigator.pop(context).

```
ListTile(
    title: const Text('Item 1'),
    onTap: () {
        Navigator.pop(context)
    },
)
```

Drawer widget

- The Navigator object in Flutter is a widget that manages a stack of pages or routes in your app. It is responsible for pushing new routes onto the stack when a user navigates to a new screen, and popping routes off the stack when a user goes back to a previous screen.
- When you call Navigator.pop(context), it pops the top-most route off the Navigator's stack, which in this case is the Drawer itself.



Drawer widget

- You can open and close a Drawer programmatically by accessing the ScaffoldState object that manages the Scaffold widget that contains the Drawer.

```
var _scaffoldKey = GlobalKey<ScaffoldState>();

Scaffold(
    key: _scaffoldKey,
    body: Container()
)

FloatingActionButton(
    onPressed: () {
        _scaffoldKey.currentState?.openDrawer();
        // _scaffoldKey.currentState?.closeDrawer()
    },
)
```

Key in Flutter

- a Key is an object that helps Flutter to identify and track widget instances in a widget tree. Every widget in a widget tree must have a Key object that uniquely identifies it. The Key object is used by Flutter to associate widget instances with their state objects and to efficiently update the widget tree.
- The Key class is an abstract class that provides a basic interface for implementing keys. The most commonly used subclass of Key is ValueKey, which takes a value of any type and uses it as the key:

```dart
final key = ValueKey('someValue');
```

- A GlobalKey is a subclass of Key that can be used to access the state of a widget from outside its subtree. A GlobalKey can be used to obtain the State object of a widget or to access the ScaffoldState object of a Scaffold widget, for example.

GlobalKey in Flutter

- A GlobalKey is a subclass of Key that can be used to identify a widget and access its state from outside its parent widget's build method.
- A GlobalKey is often used to access the state of a StatefulWidget or to manipulate a Scaffold widget's state, for example:

```
var _myAppKey = GlobalKey<_MyAppState>();
```

- In this example, a GlobalKey object is created for a widget called MyAppState. The GlobalKey is assigned to a variable called _myAppKey.
- Once you have created a GlobalKey, you can use it to access the state of the widget associated with it:

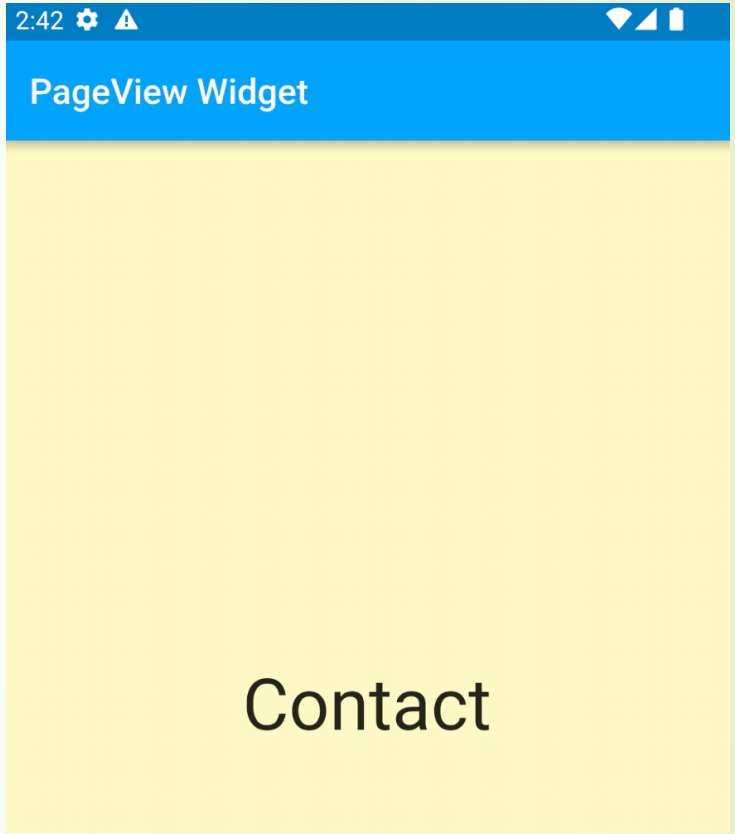
```
_myAppKey.currentState.doSomething();
```

- In this example, the currentState property of the GlobalKey is used to obtain the State object of the MyAppState widget. The `doSomething()` method is then called on the State object.

PageView

PageView widget

- The PageView widget is a powerful and flexible widget in Flutter that allows users to swipe horizontally through a series of pages. It's often used in mobile apps to display onboarding screens, product catalogs, and image galleries.
- The PageView widget is easy to use and highly customizable, with many options for controlling the animation, layout, and scrolling behavior of the pages. You can use the PageView widget with any type of widget, including images, text, forms, and custom widgets.



PageView widget

- Here's a simple example of how to use the PageView widget in Flutter

```
Scaffold(
    appBar: AppBar(title: const Text('PageView Widget')),
    body: PageView(
        children: [
            _createPageItem('Home'),
            _createPageItem('History'),
            _createPageItem('Contact'),
        ],
    ),
)
```

```
Widget _createPageItem(label) {
    return Container(
        alignment: Alignment.center,
        child: Text(label),
    );
}
```

PageView widget

- You can also use the PageView.builder constructor to build pages dynamically based on an index. This is useful when you have a large number of pages or when the content of the pages is generated dynamically.

```
Scaffold(
    appBar: AppBar(title: const Text('PageView Widget')),
    body: PageView.builder(
        itemCount: 10,
        itemBuilder: (ct, idx) => _createPageItem('Item $idx'),
    )
)
```

- In this example, a PageView.builder widget is created with 10 pages, each represented by a Container widget with a label containing the position of the item.

PageView widget

- The PageView widget uses a PageController to manage the animation and scrolling behavior of the pages. You can use the PageController to programmatically control the current page, change the animation duration, and listen to page scroll events.

```
final PageController _controller = PageController();
int _currentPage = 0;

@override
void dispose() {
    _controller.dispose();
    super.dispose();
}

PageView(controller: _controller, …)
```

PageView widget

- Change page programmatically:
  - In this example, the _goToPage method is used to change the current page of the PageView. This method uses the _controller.animateToPage method to animate the transition to the new page. The duration parameter specifies the length of the animation, and the curve parameter specifies the easing curve for the animation.

```
void _goToPage(int page) {
    _controller.animateToPage(
        page,
        duration: Duration(milliseconds: 500),
        curve: Curves.ease);
}
```

PageView widget

- There are two common ways to listen for page change events of a PageView widget: using the onPageChanged property or using a PageController with a listener.
- Using the onPageChanged property:
  - The PageView widget has an onPageChanged property that allows you to register a callback function that will be called whenever the user scrolls to a new page. This function takes an integer parameter that represents the index of the new page.

```
PageView(
    onPageChanged: (int pageIndex) {
        print('Current page: $pageIndex');
    },
    children: [..],
)
```

PageView widget

- Using a PageController with a listener:
  - The PageController class provides a addListener method that allows you to register a listener that will be called whenever the page changes, whether it was triggered by user input or programmatically. This function takes a callback function that receives a double parameter representing the new page position.

```
final PageController _controller = PageController();

_controller.addListener(() {
    final int currentIndex = _controller.page;
    print('Current page: $currentIndex');
});
```