CMPS 312

Navigation

Dr. Abdelkarim Erradi

CSE@QU

Navigation

The act of moving between screens of an app to complete tasks

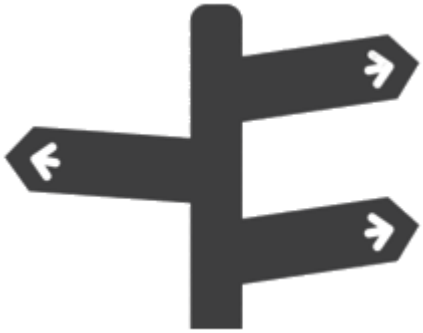
Designing effective navigation = Simplify the user journey

Outline

1. Navigation
2. Navigation Widgets
3. Responsive Navigation UI
4. Floating Windows

Navigation

Used for navigating between destinations within an app

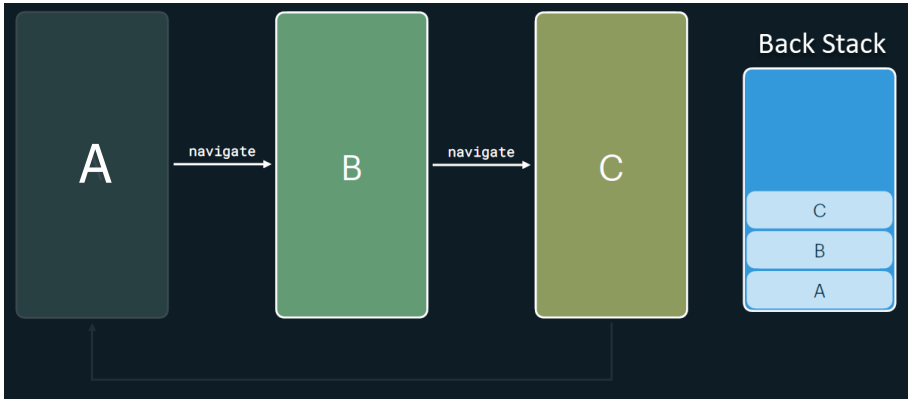


Back

https://docs.flutter.dev/ui/navigation

Navigator

- **Navigator** is used to request navigating to a particular screen
  - Keeps track of the **back stack** of visited screens



- **Navigator** is a widget that manages a stack of routes (screens) and allows navigation between them using:
  - `push`: adds a new route to the stack for displaying new screen
  - `pop`: removes the current route, returning to the previous one
  - `pushReplacement`: replaces the current route with a new one
  - `pushNamed`: Navigates to a named route defined in MaterialApp

Navigator Example

```
Navigator.of(context).push(
      MaterialPageRoute(builder: (context) {
         return FruitDetailScreen(fruit: fruit);
       },
  ),
);
```

- `Navigator.of(context)` retrieves the current Navigator instance from the widget tree
- `MaterialPageRoute` ease the transition to a new screen with platform-specific animation
  - It takes a builder function that returns the screen to navigate to
  - E.g., the builder returns an instance of `FruitDetailScreen` while passing a fruit object to its constructor

Named routes

- Named routes provide a way to navigate using **string identifiers** rather than directly using widgets
  - Makes route management more structured and scalable
  - Named routes are defined in the `MaterialApp` widget using the **`routes`** property, where each route maps a string identifier with the corresponding widget
  - `Navigator.pushNamed()`: Navigates to a named route
- See the posted example:
  - Navigate between Home, Profile, and Fruits screens using the `BottomNavigationBar`

Navigate with arguments

- When using **`Navigator.pushNamed`**, you can pass arguments to the new screen, allowing the next screen to receive and use the data
  - You can pass any data type as arguments (e.g., a string, an object) as argument when calling Navigator.pushNamed
  - E.g., navigating from a product list screen to a product details screen, the tapped `product` object is passed as argument

```
Navigator.of(context).pushNamed('/productDetails',
                    arguments: product);
```

  - On the destination screen, arguments can be retrieved using `ModalRoute.of(context)?.settings.arguments`

```
final Product product =
ModalRoute.of(context)?.settings.arguments as Product;
```

Navigator.popUntil

- By default, **`push()`** adds the new screen to the back stack (i.e., history of visited screens). To modify this behavior, use **`popUntil`** or **`pushAndRemoveUntil`** methods:
  - **`Navigator.popUntil`** pops screens (routes) from the navigation stack until it reaches a route that matches a specific condition
    - e.g., Let's say you're on a "Profile" screen and want to pop all the way back to the "Home" screen, skipping over an intermediate "Settings" screen

```
/* Pop off from the back stack up to the route named '/home' */
Navigator.of(context).popUntil(ModalRoute.withName('/home'));
```

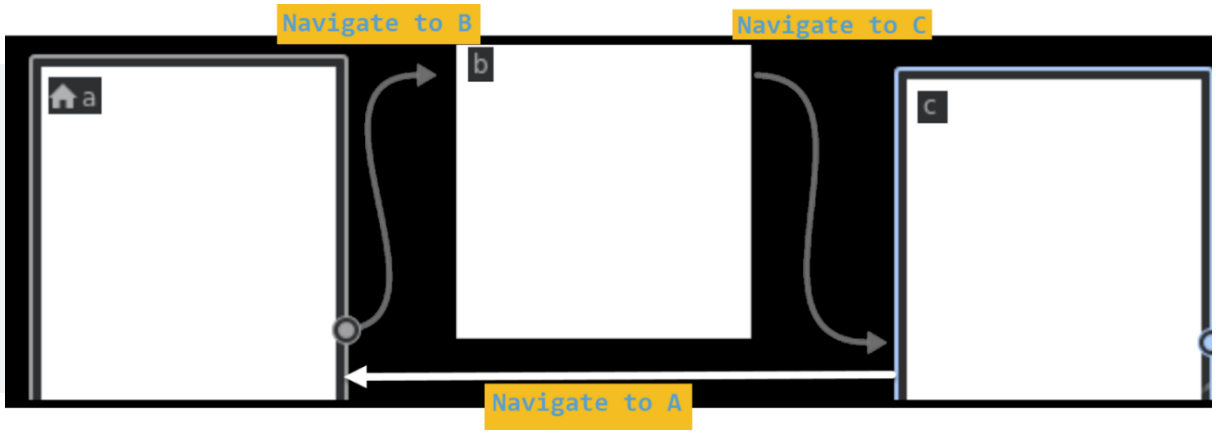
Navigator.pushAndRemoveUntil

- **Navigator.pushAndRemoveUntil** pushes a new screen onto the stack, and then removes off previously visited screen from the back stack (up to the specified route)
  - For example, after a login flow, you should **pop off all the login-related screen** of the back stack so that the Back button doesn't take users back into the login flow
  - It should go back to the Home Screen while removing all visited destinations from the back stack

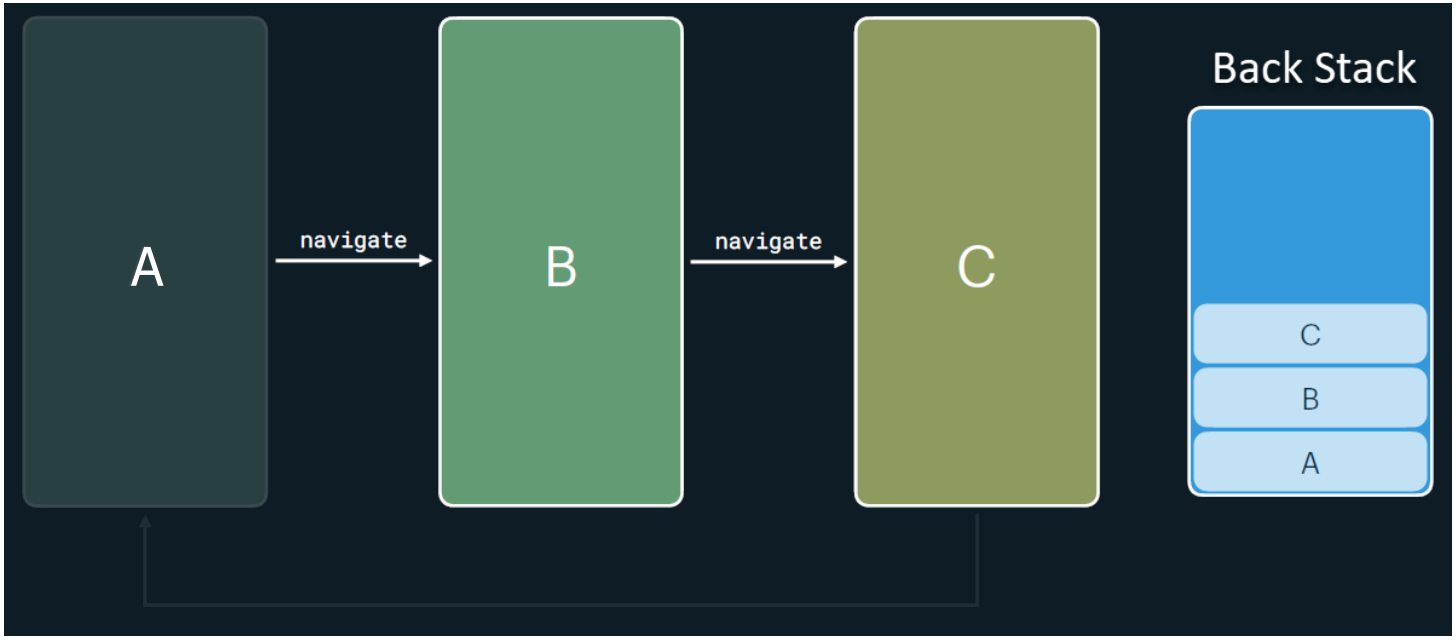
```
/* navigating to the dashboard screen the pop everything up to the
"home" screen off the back stack */
Navigator.of(context).pushAndRemoveUntil(
    MaterialPageRoute(builder: (context) => DashboardScreen()),
    // Keep popping until the '/home' route, but leave it in the stack
    ModalRoute.withName('/home')
);
```

popUntil Example

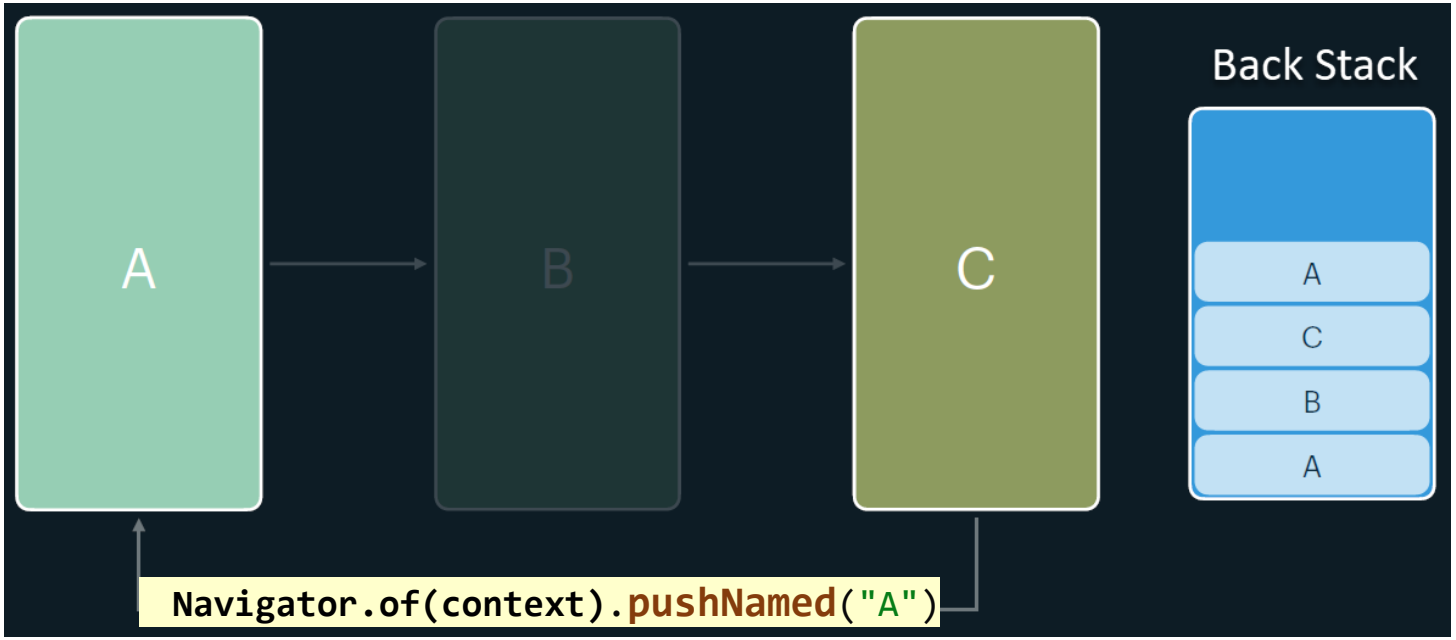
```
Navigator.of(context).
popUntil(
 ModalRoute.withName('/A')
);
```

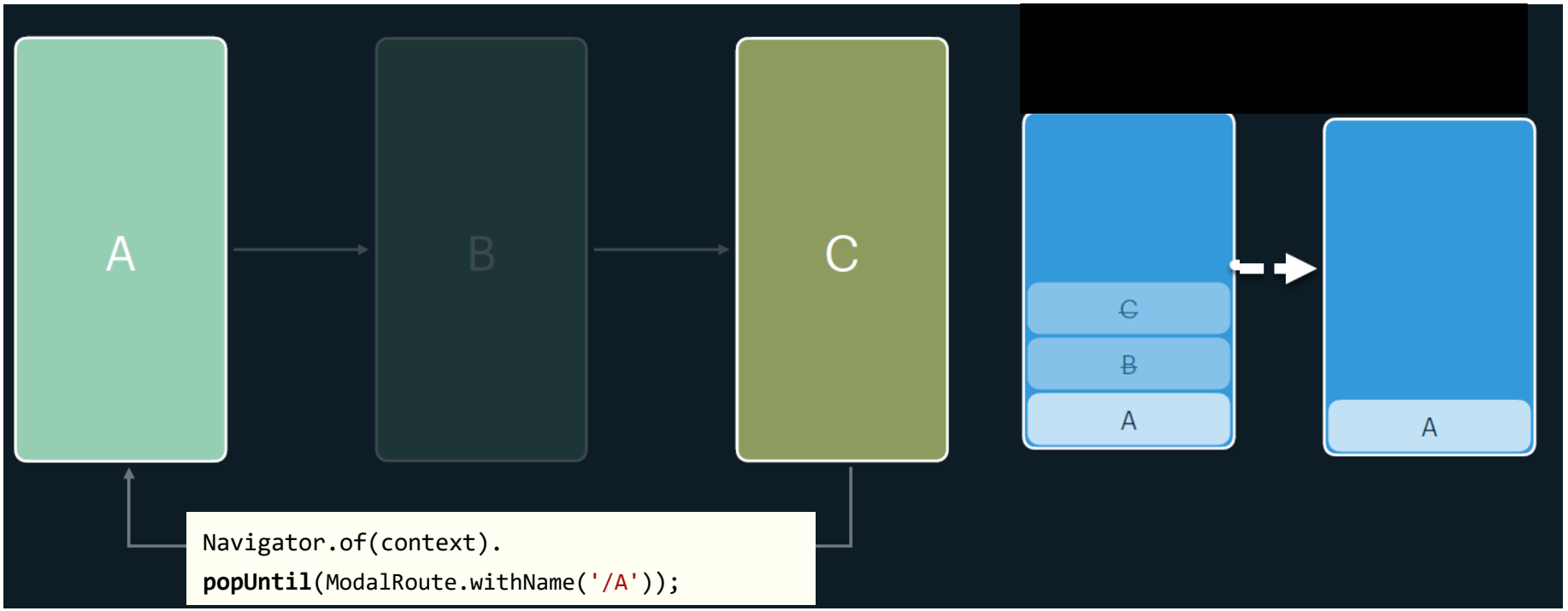


- After reaching C, the back stack contains (A, B, C). **popUntil** ‘A’ will remove B and C from the stack




```
Navigator.of(context).pushNamed("A")
```




A
B
C
C
B
A
A
Navigator.of(context).
popUntil(ModalRoute.withName('/A'));

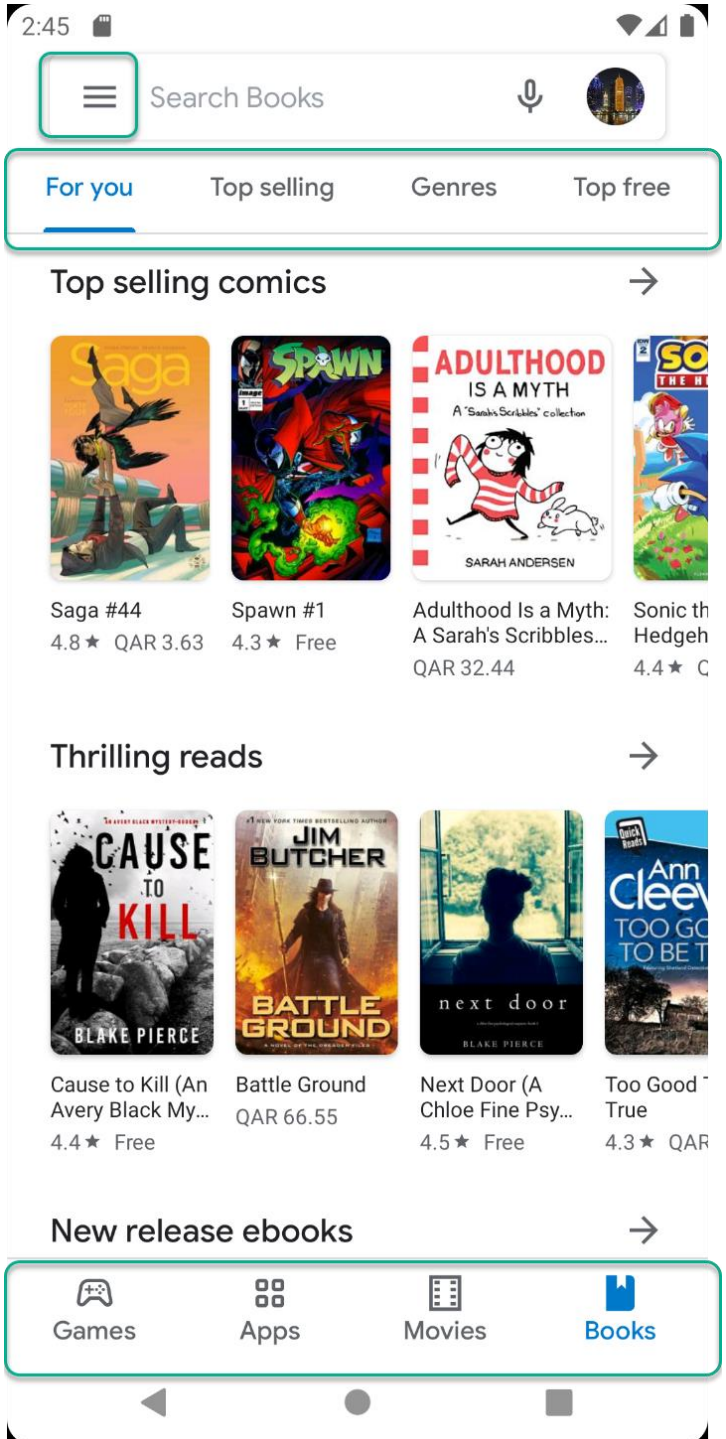
Navigation Widgets:

App Bars

Navigation Rail

Floating Action Button

Navigation Drawer

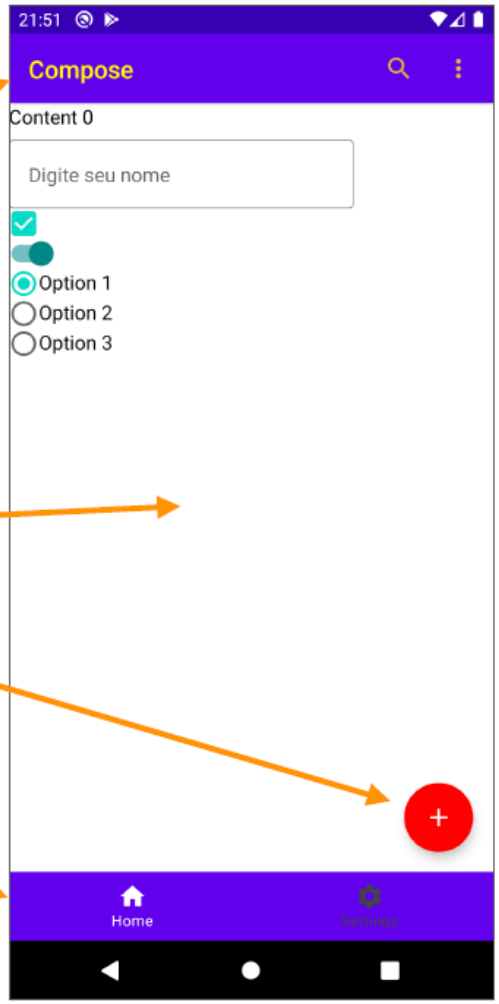


Back

Scaffold

- **Scaffold** is a **Slot-based** layout
- Scaffold is **template** to build the entire screen by adding different UI Navigation components (e.g., *topBar*, *bottomBar*, *floatingActionButton*)

```
Scaffold(
    topBar = {...},
    floatingActionButton = {...},
    bottomBar = {...}
) {...}
```

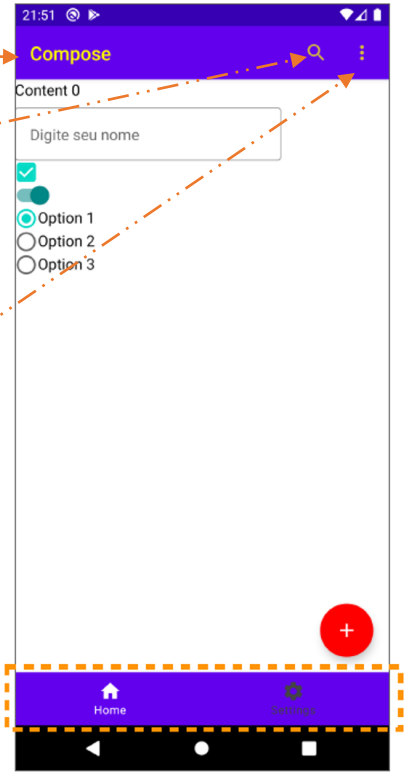


TopAppBar

- Info and actions **related to the current screen**
- Typically has Title, Drawer button / Back button, Menu items

```
TopAppBar(
    title = {
        Text(text = "Compose")
    },
    navigationIcon = {
        IconButton(onClick = { }) {
            Icon(
                imageVector = Icons.Default.Search,
                contentDescription = "Search"
            )
        }
    }

    navigationIcon = {
        IconButton(onClick = { }) {
            Icon(
                imageVector = Icons.Default.MoreVert,
                contentDescription = "More"
            )
        }
    }
)
```

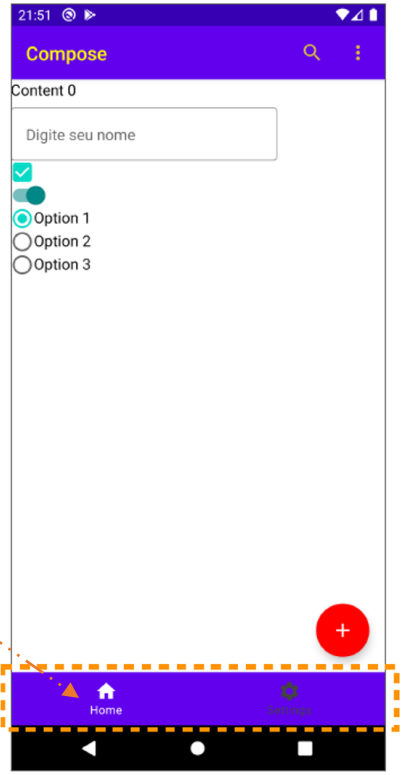


Bottom Navigation Bar

- Allow movement between the app's primary **top-level destinations** (3 to 5 options)
- Each destination is represented by an icon and an optional text label. May have notification badges

- Recommended for **compact screen**

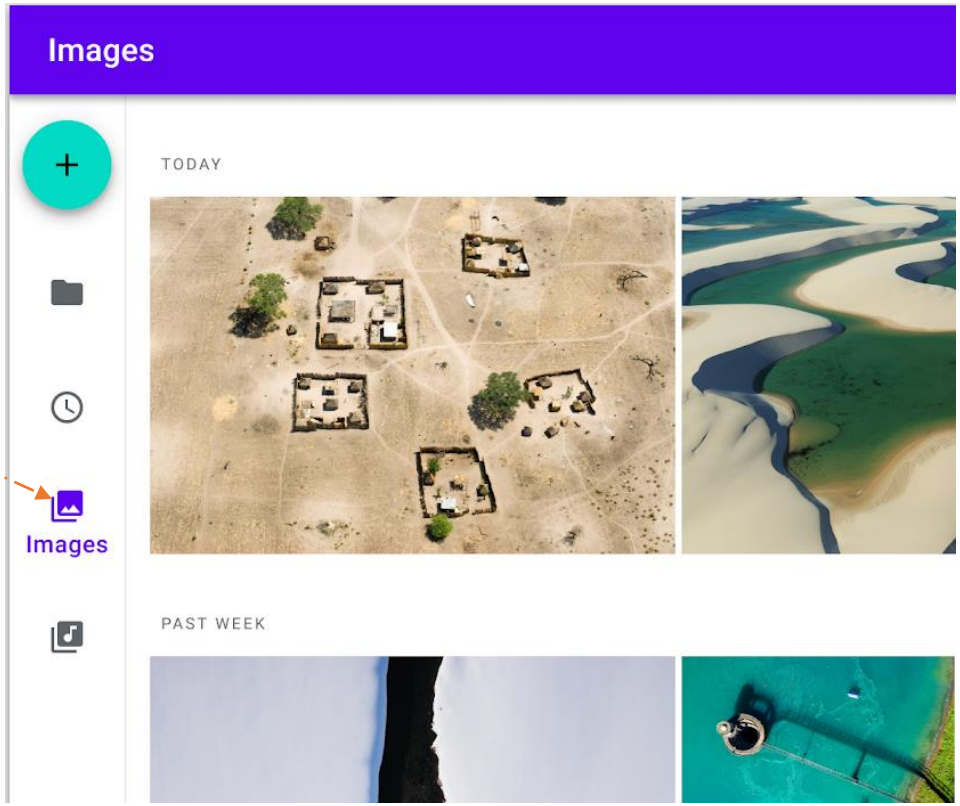
```
NavigationBar {
   NavigationBarItem(
     icon = {Icon(Icons.Default.Home,
                  contentDescription = "Home")},
     label = { Text( "Home" ) }
     onClick = { },
   )
   ...
   NavigationBarItem(
     icon = { },
     label = { }
     onClick = { },
   )
}
```



Navigation Rail

- Can contain 3-7 destinations plus an optional FAB
- Recommended for for **medium** or **expanded** screens

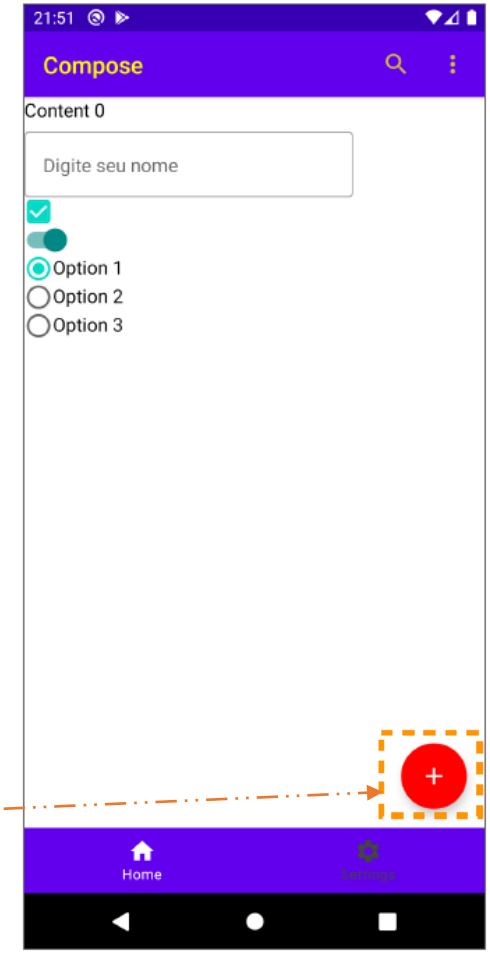
```
NavigationRail {
    ...
    NavigationRailItem(
      icon = {Icon(Icons.Default.Image,
             contentDescription = "Images")},
      label = { Text( "Images" ) }
      onClick = { },
    )
    ...
    NavigationRailItem(
      icon = { },
      label = { }
      onClick = { },
    )
}
```



Floating Action Button (FAB)

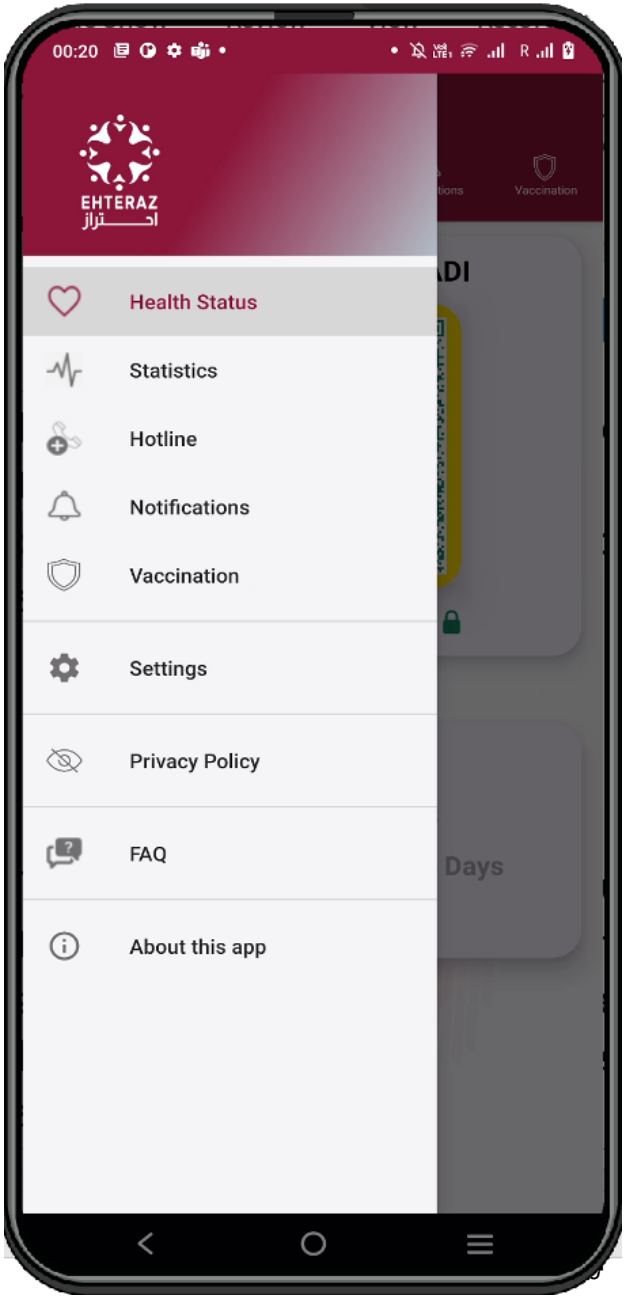
- A FAB performs the primary, or most common, action on a screen, such as drafting a new email
  - It appears in front of all screen content, typically as a circular shape with an icon in its center.
  - FAB is typically placed at the bottom right

```
FloatingActionButton(
    onClick = { … },
    backgroundColor = Color.Red,
    contentColor = Color.White
) {
    Icon(Icons.Filled.Add, "Add")
}
```



Navigation Drawer

- Navigation Drawer provides access to app **destinations** that cannot fit on the Bottom Bar , such as settings screen
  - Recommended for five or more top-level destinations
  - Quick navigation between unrelated destinations
- The drawer appears when the user touches the drawer icon ☰ in the app bar or when the user swipes a finger from the left edge of the screen

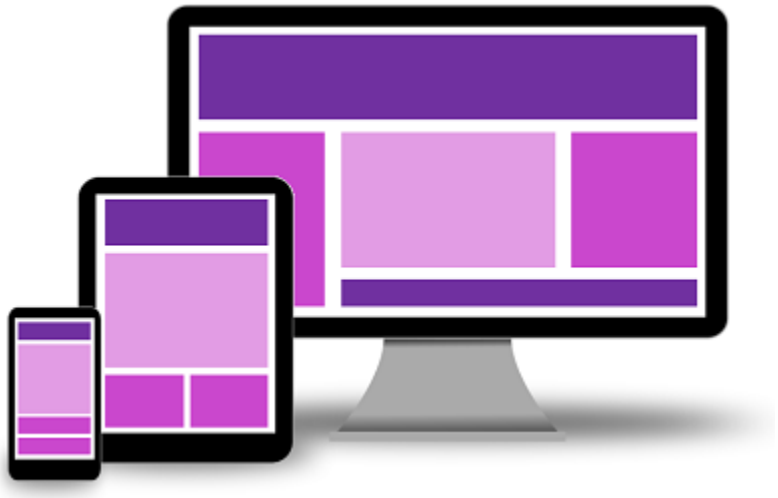


Navigation Drawer - Example

```
ModalNavigationDrawer(
    drawerContent = {
        ModalDrawerSheet {
            NavigationDrawerItem(
                label = { Text(text = "Settings" ) },
                icon = { Icon(Icons.Default.Settings,
                                    contentDescription = "Settings")
                        },
                onClick = { }
            )
            ...
        }
})
```

- See more details in the posted example

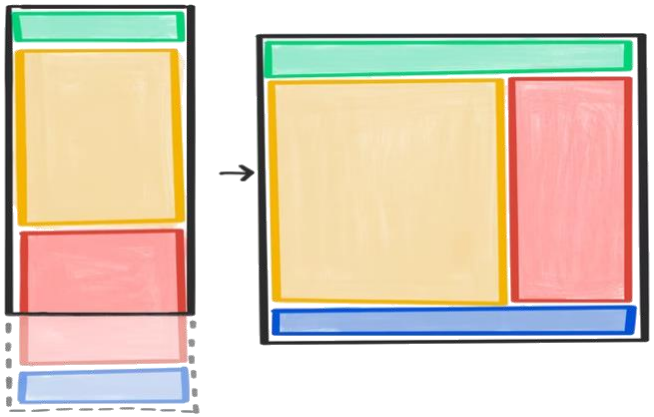
Responsive Navigation UI



Back

Responsive UI

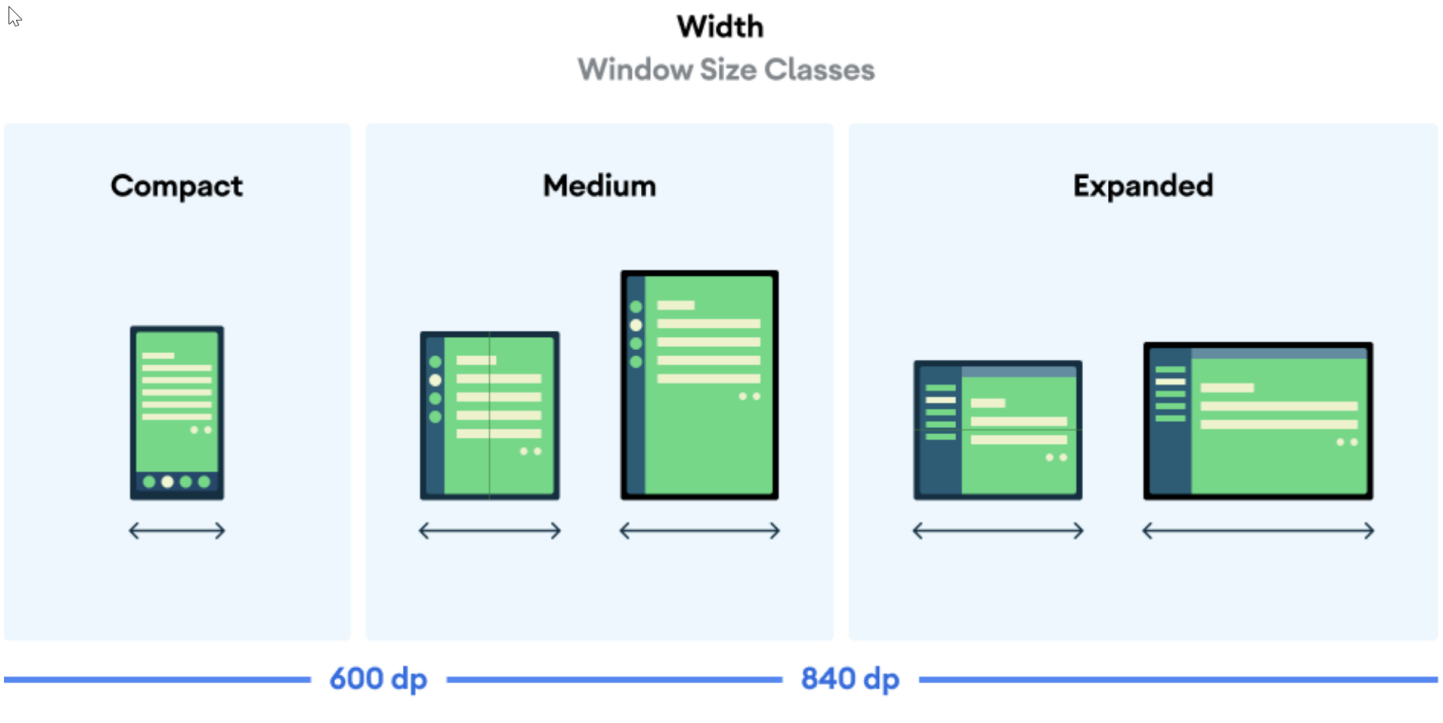
- Responsive UI = **serve different layouts for different screen sizes and orientations**
  - **Optimize the viewing experience on range of devices**: mobile, desktop, tablet, TV…
- For example, a newspaper app might have a single column of text on a mobile device, but display several columns on a larger tablet/desktop device



windowSizeClass

- `calculateWindowSizeClass` return a window size class. It can be either **compact**, **medium**, or **expanded**.

```
val context = LocalContext.current as Activity
val windowSizeClass =
      calculateWindowSizeClass(context)
```



Design for window size classes instead of specific devices

- Devices fall into different window size classes based on orientation and user behavior, such as multi-window modes or unfolding a foldable device
- Start by designing for compact window class size and then adjust your layout for the next class size

Window class (width)	Breakpoint (dp)	Common devices
Compact	Width < 600	Phone in portrait
Medium	600 <= width < 840	Tablet in portrait Foldable in portrait (unfolded)
Expanded	Width >= 840	Phone in landscape Tablet in landscape Foldable in landscape (unfolded) Desktop

Responsive UI - Example

- A bottom navigation bar in a compact layout can be swapped with a navigation rail in a medium layout, and a navigation drawer in an expanded layout

Responsive UI - Example

```
val context = LocalContext.current as Activity
val windowSizeClass = calculateWindowSizeClass(context)

val shouldShowBottomBar =  windowSizeClass.widthSizeClass
       == WindowWidthSizeClass.Compact
val shouldShowNavRail = !shouldShowBottomBar
...
Scaffold(
    bottomBar = {
        if (shouldShowBottomBar)
            BottomNavBar(navController)
    }
) {
    padding -> Row(...) {
        if (shouldShowNavRail) {
            AppNavigationRail(navController)
        }
        AppNavigator(navController = navController)
     }
}
```

Floating Windows

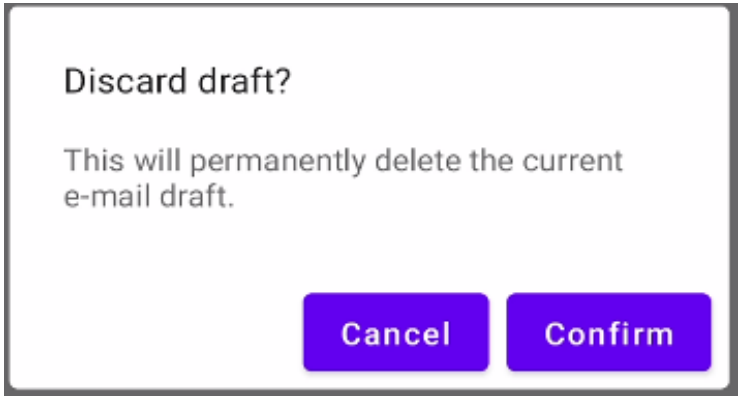
Back

Alert Dialog

- Alert dialog is a Dialog which interrupts the user with urgent information, details or actions
- Dialogs are displayed in front of app content
  - Inform users about a task that may contain **critical information** and/or **require a decision**
  - Interrupt the current flow and remain on screen until dismissed or action taken. Hence, they should be used sparingly
- 3 Common Usage:
  - **Alert dialog**: request user action/confirmation. Has a title, optional supporting text and action buttons
  - **Simple dialog:** Used to present the user with a list of actions that, when tapped, take immediate effect.
  - **Confirmation dialog:** Used to present a list of single- or multi-select choices to a user. Action buttons serve to confirm the choice(s)

Alert Dialog

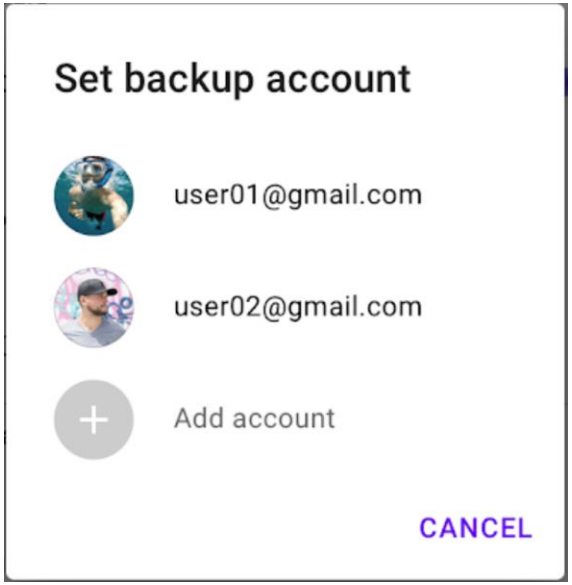
- Commonly used to **confirm high-risk actions** like deleting progress



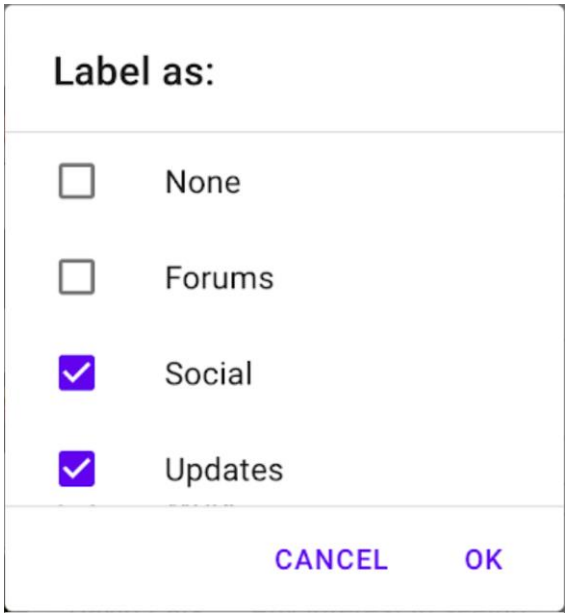

```
AlertDialog(
        onDismissRequest = {
           // Dismiss the dialog when the user clicks outside the dialog
           // or on the back button
             onDialogOpenChange(false)
        },
        title = { Text(text = title) },
        text = { Text(text = message) },
        confirmButton = {
            Button(
                onClick = {  onDialogResult(true) }) {
                Text(text = "Confirm")
            }
        },
        dismissButton = {
            Button(
                onClick = {  onDialogResult(false) }) {
                Text("Cancel")
            }
        }
    )
}
```

Simple dialog:

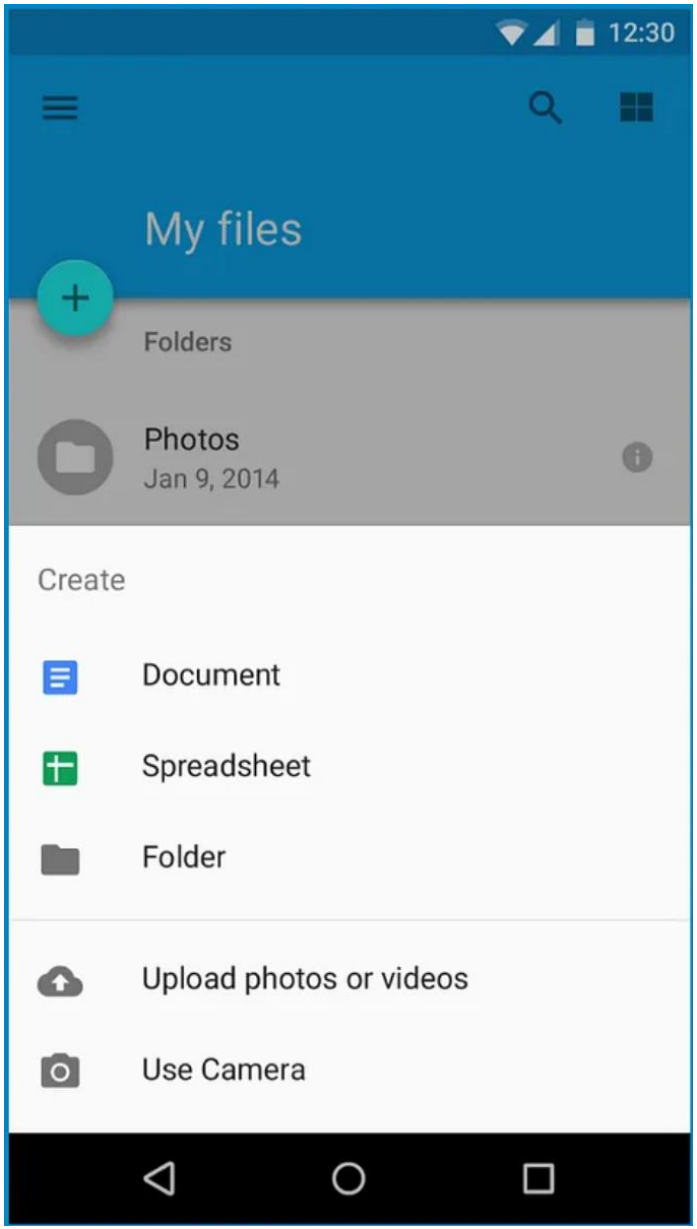
present the user with a list of actions that, when tapped, take immediate effect



Confirmation dialog (multi choice)



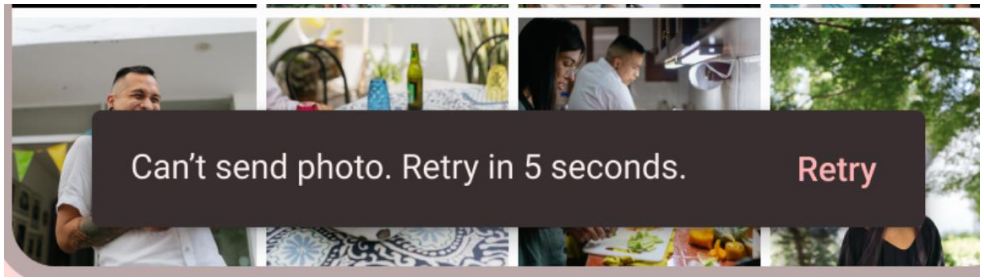
Bottom Sheets



- Bottom sheets show secondary content / actions anchored to the bottom of the screen
- Content should be additional or secondary (not the app's main content)
- Bottom sheets can be dismissed in order to interact with the main content
- See more details in the posted example

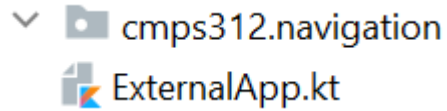
Snackbar

- Snackbars show **short updates** about app processes at the bottom of the screen



- Do not interrupt the user’s experience
- Can disappear on their own or remain on screen until the user takes action
- See more details in the posted example

Routing to External App



- **Intent** can be used to route a request to another app
  - Specify an **Action** and the **Parameters** expected by the action
  - Implicit intents can be handled by **a component in an installed app** registered to handle that intent type

- **Dial a number:**

```
val intent = Intent(Intent.ACTION_DIAL).apply {
    data = Uri.parse("tel:$phoneNumber")
}
context.startActivity(intent)
```

- **Open a Uri**

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("https://www.qu.edu.qa"))
startActivity(intent)
```

- **Share content**

```
val intent = Intent(Intent.ACTION_SEND).apply {
    putExtra(Intent.EXTRA_TEXT, content)
    type = "text/plain"
}
context.startActivity(Intent.createChooser(intent, "Share via"))
```

- Other common intents discussed here

Using Sealed Class to Enumerate the App Destinations

- A sealed class allows defining subclasses, but they must be in the same file as the sealed class
  - It is like enum class but more flexible as it allows subclasses to have different properties and methods
  - A sealed class cannot be instantiated directly
- A sealed class is often used to enumerate the app destination as shown in the example below

```
sealed class NavDestination(val route: String, val title: String, val icon: ImageVector? = null,
            val iconResourceId:Int? = null) {
    object Quran : NavDestination(route = "quran", title = "Quran", iconResourceId = R.drawable.ic_quran)
    object Verses : NavDestination(route = "verses", title = "Surah Verses", iconResourceId = R.drawable.ic_quran)
    object Search : NavDestination(route = "search", title = "Search", icon = Icons.Outlined.Search)
    object Settings : NavDestination(route = "settings", title = "Settings", icon = Icons.Outlined.Settings)
}
```

Resources

- Flutter Navigation
  - https://docs.flutter.dev/ui/navigation
- Flutter Navigation hands-on practice
  - https://docs.flutter.dev/cookbook#navigation
- Declarative navigation using go_router package