CMPS 312

Navigation

Dr. Abdelkarim Erradi

CSE@QU

Navigation

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

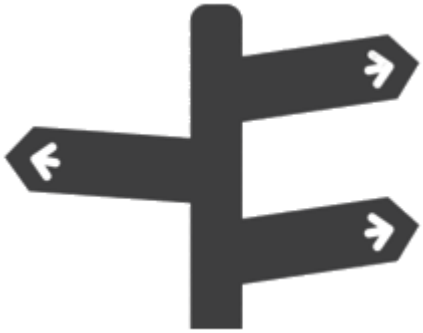
Designing effective navigation = Simplify the user journey

Outline

1. Jetpack Compose Navigation
2. Navigation UI Components
3. Floating Windows
4. Responsive UI

Jetpack Compose Navigation

Used for navigating between destinations within an app



Back

https://developer.android.com/jetpack/compose/navigation

Single Activity with Multi-Screens

- App UI = { **1 Activity** + Multi-Screens }
  - A Screen is a composable that represents **a portion of the UI**
- The Navigation Component enables implementing Single Activity App with the ability to navigate between the app screens (also called **destinations**)
- Requires the following dependency in app module's *build.gradle* file:

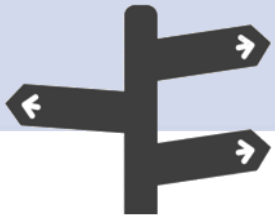
```
implementation "androidx.navigation:navigation-compose::<version>"
```

Navigation uses 2 main Classes

NavHost

- Defines the app **Navigation Graph** = possible **routes** a user can take through the app

- Acts as a **container** to load the screen associated with the **route** requested by the NavController



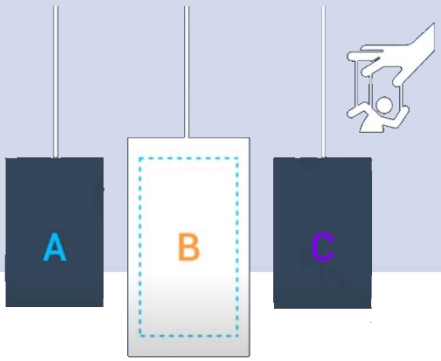
NavController

- Used to request navigating to a particular route

e.g.,

```
navController.navigate("friends")
```

- Keeps track of the **back stack** of visited screens



Creating a NavHost

- **`NavHost`** (typically added to the Main Screen) is used to define a **navigation graph** to specify the possible **routes** within the app
  - A **route** associates a **path** name to a specific screen
    - Each app route should have a unique name
    - One of the route will be used as the start destination
- The Nav Graph is defined using the **`composable`**() function to map each **route** to the associated **screen**

```
NavHost(navController = navController, startDestination = "profile") {
    composable("profile") { ProfileScreen() }
    composable("orders") { OrdersScreen() }
    /*...*/
}
```

Navigate to a destination using NavController

- **navController** object is created in the Main Screen using the **rememberNavController**()

```
val navController = rememberNavController()
```

- navController.**navigate**(***destinationRoute***) method is used to navigate to a specific destination
  - The requested destination screen will be loaded by the **NavHost**
- navController.**navigateUp**() navigates to the previous screen

```
@Composable
fun Profile(navController: NavController) {
    /*...*/
     Button(onClick = {
                navController.navigate("friends")
         })
    {  Text(text = "Show Friends") }
}
```

Navigate with arguments

- To pass arguments to a destination e.g., get the profile for user 123 `navController.navigate("profile/123")`
  - First add the argument placeholder to the destination route

  e.g., The user profile destination takes a *userId* argument to determine which user to display

```
NavHost( ...) {
    composable("profile/{userId}") {...}
}
```

- By default, all arguments are parsed as strings. You can specify another type by using the arguments parameter

```
composable("profile/{userId}",
    arguments = listOf(navArgument("userId") { type = NavType.IntType })
) { ... }
```

Extract the Nav Arguments from the Nav BackStackEntry

- Nav BackStackEntry represent an entry in the back stack
  - The route provides access the current **backStackEntry** to extract the navigation arguments

```
composable("profile/{userId}",
    arguments = listOf(navArgument("userId") { type = NavType.IntType })
) { backStackEntry ->
    // Extract the Nav Arguments from the Nav BackStackEntry
    Profile(navController, backStackEntry.arguments?.getInt("userId"))
}
```

Adding optional arguments

- Optional arguments must be added to the `composable()` as a **query parameter**

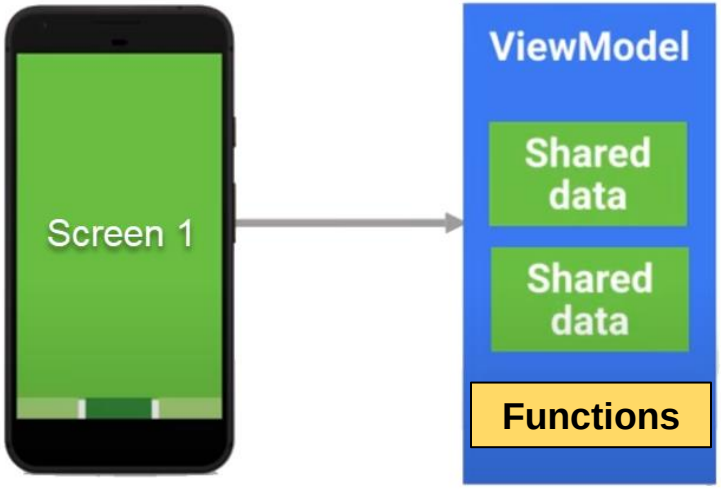
  `?argName={argName}`

  - Optional arguments must have a defaultValue, or set ***`nullable = true`***

```
composable("profile?userId={userId}",
    arguments = listOf(navArgument("userId") {
                        type = NavType.IntType
                        defaultValue = 123 })
) { backStackEntry ->
    Profile(navController, backStackEntry.arguments?.getInt("userId"))
}
```

Shared data/functions between Screens using ViewModel

- Screens can **share** data using a shared **View Model** class that extends `ViewModel()`




```
class UserViewModel : ViewModel() {
    val users = mutableStateListOf(
        User(1, "Ahmed", "Faleh", "ahmed@test.com"),
        User(2, "Fatima", "Faleh", "fatima@test.com"),
        ….
    )
}
```

```
val userViewModel = viewModel<UserViewModel>()
NavHost(
    …
    composable(Screen.Users.route) {
        UsersScreen(userViewModel)
    }
```

NavOptions - popUpTo and popUpTo inclusive

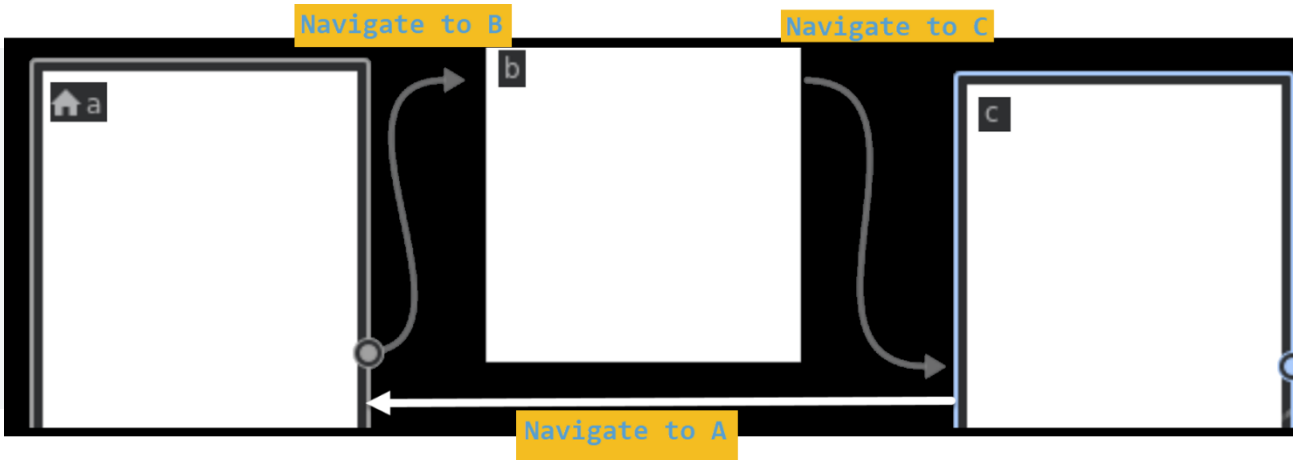
- By default, **`navigate()`** adds the new destination to the back stack (i.e., history of visited screens). To modify this behavior, pass **navigation options** to **`navigate()`** call
  - **`launchSingleTop = true`** : Navigate to the destination only if we're not already on it to avoid multiple copies of the destination screen on the back stack
  - **`popUpTo(route)`** : pop off previously visited destinations from the back stack (up to the specified route)
    - For example, after a login flow, you should **pop off all the login-related destinations** 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
    - If *`inclusive = true`* the destination specified in **popUpTo** should also be removed from the back stack

Navigation Options: popUpTo & launchSingleTop

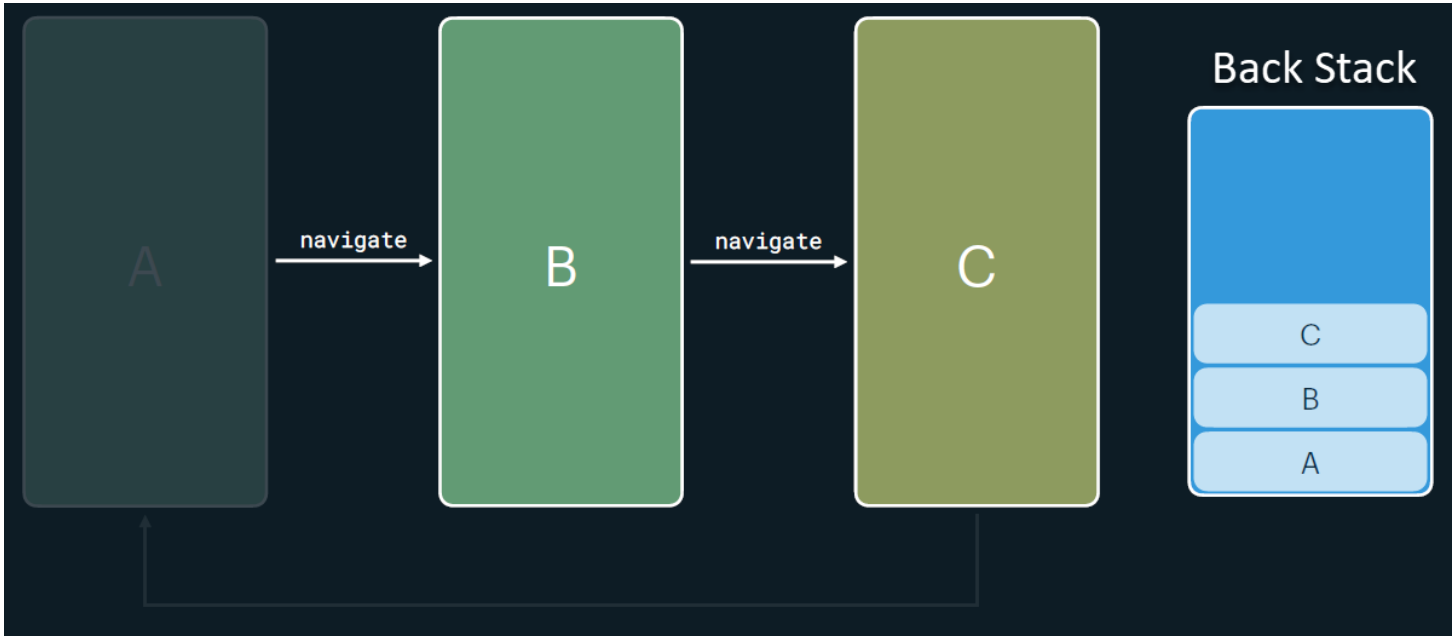
```
/* Pop everything up to the "home" destination off the back stack before
 navigating to the "friends" destination */
navController.navigate("friends") {
    popUpTo("home")
}

/* Pop off from the back stack up to and including the "home" destination
 before navigating to the "friends" destination */
navController.navigate("friends") {
    popUpTo("home") { inclusive = true }
}

/* Navigate to the "search” destination only if we’re not already on
 the "search" destination, avoiding multiple copies of the search screen
 on the back stack */
navController.navigate("search") {
    launchSingleTop = true
}
```

popUpTo Example

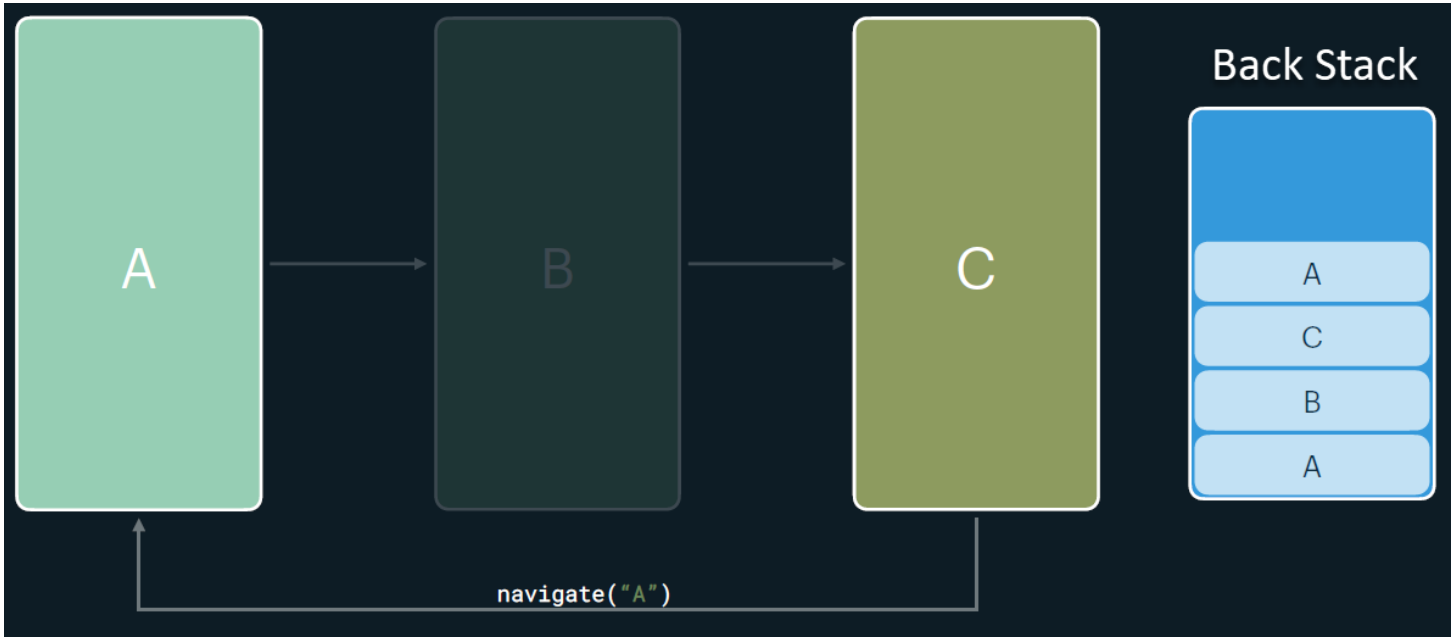
```
navController.navigate("A") {
    popUpTo("A") {
        inclusive = true
    }
}
```

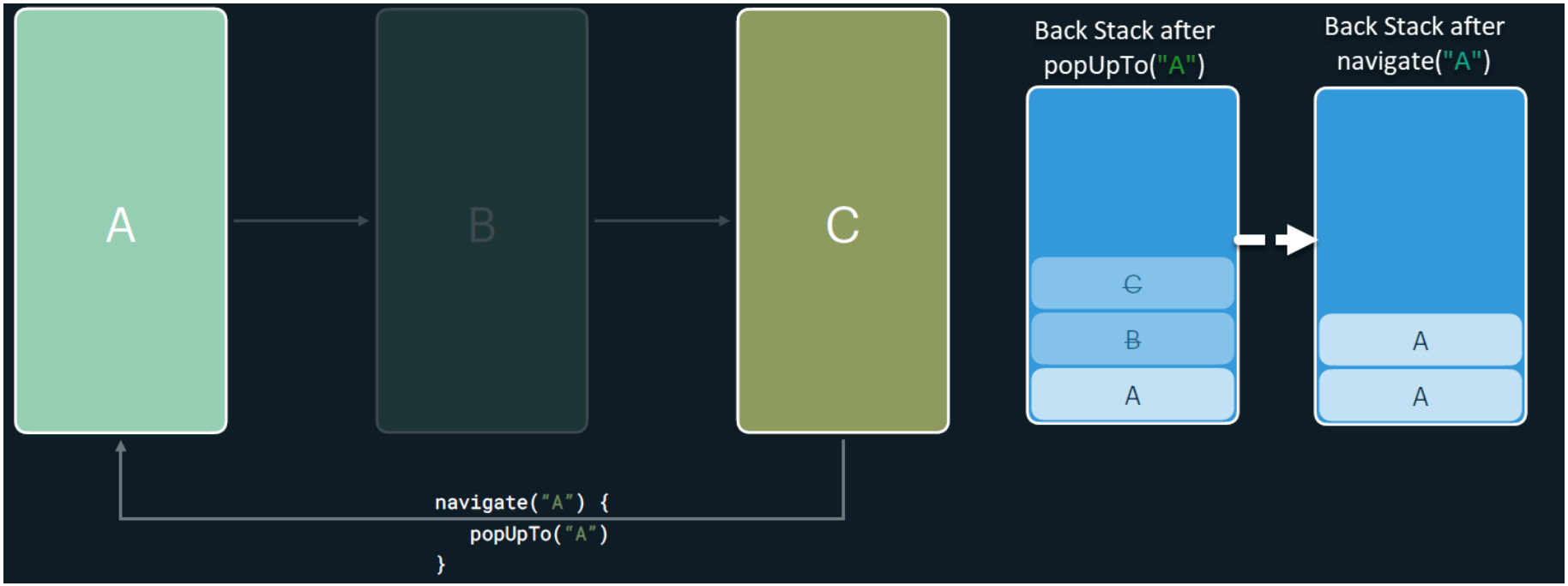


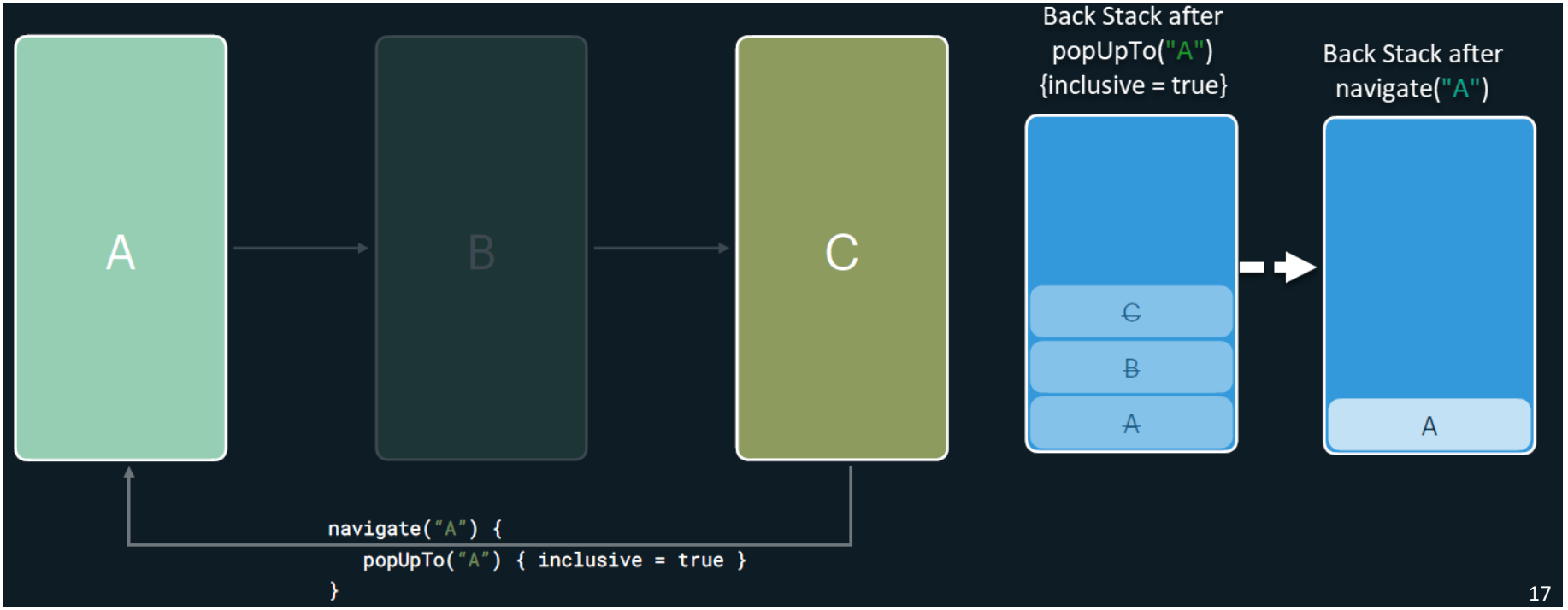
- After reaching C, the back stack contains (A, B, C). When navigating back to A, we also **popUpTo A**, which means that we remove B and C from the stack as part of the call to **navigate**(**"A"**)
  - With inclusive= true, we also pop off that first A of the stack to avoid having two instances of A




```
navController.navigate("A")
```




A
B
C
Back Stack after popUpTo("A")
C
B
A
Back Stack after navigate("A")
A
A
navigate("A") {
popUpTo("A")
}


A
B
C
Back Stack after popUpTo("A") {inclusive = true}
C
B
A
Back Stack after navigate("A")
A
navigate("A") {
popUpTo("A") { inclusive = true }
}

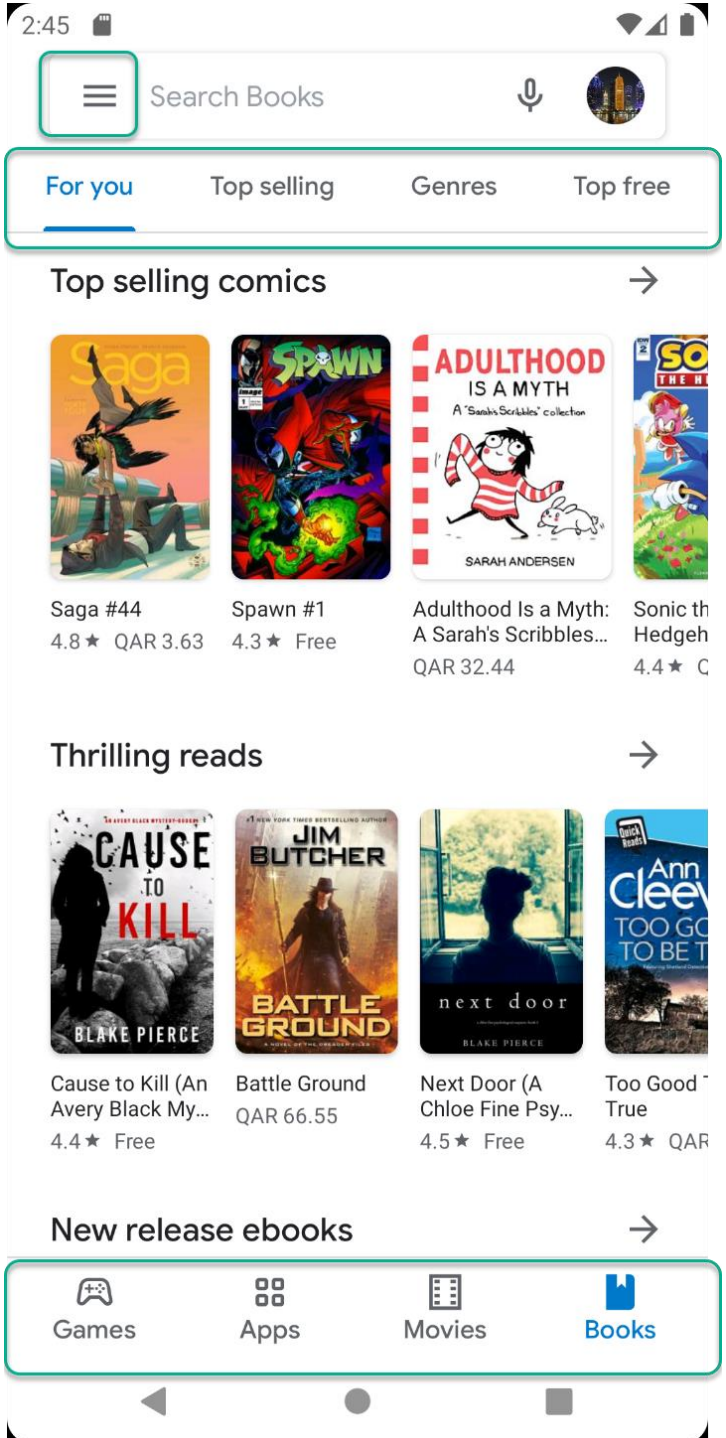
Navigation UI Components:

App Bars

Navigation Rail

Floating Action Button

Navigation Drawer

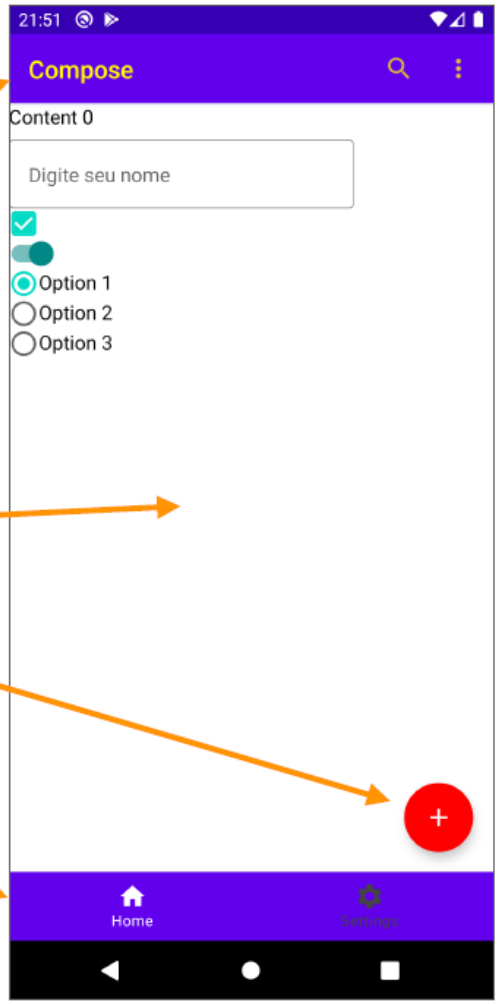


Back

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*)

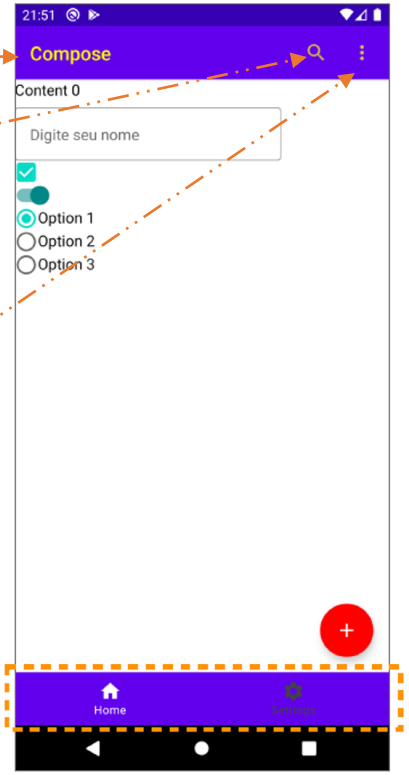
```
Scaffold(
    topBar = {...},
    floatingActionButton = {...},
    bottomBar = {...}
) {...}
```



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"
            )
        }
    }
)
```

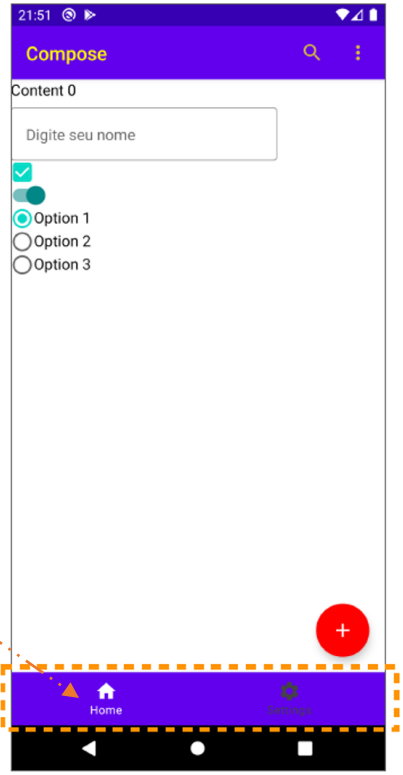


Bottom Navigation Bar

- 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**

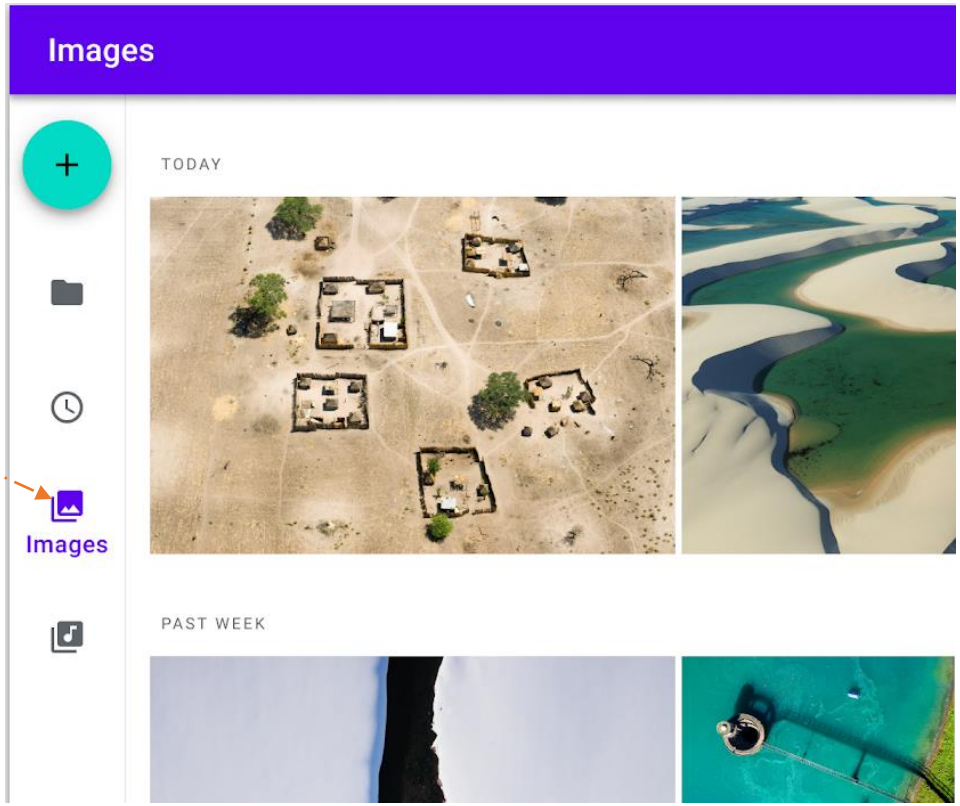
```
NavigationBar {
   NavigationBarItem(
     icon = {Icon(Icons.Default.Home,
                  contentDescription = "Home")},
     label = { Text( "Home" ) }
     onClick = { },
   )
   ...
   NavigationBarItem(
     icon = { },
     label = { }
     onClick = { },
   )
}
```



Navigation Rail

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

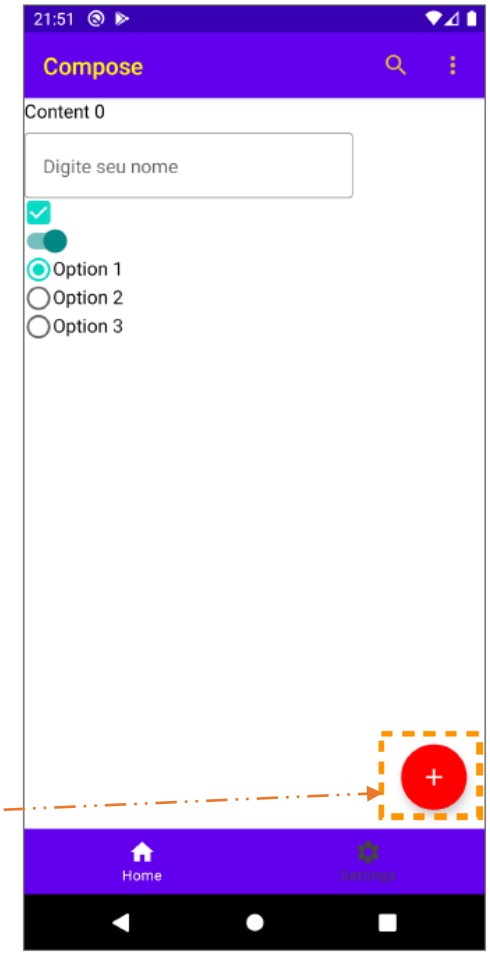
```
NavigationRail {
    ...
    NavigationRailItem(
      icon = {Icon(Icons.Default.Image,
             contentDescription = "Images")},
      label = { Text( "Images" ) }
      onClick = { },
    )
    ...
    NavigationRailItem(
      icon = { },
      label = { }
      onClick = { },
    )
}
```



Floating Action Button (FAB)

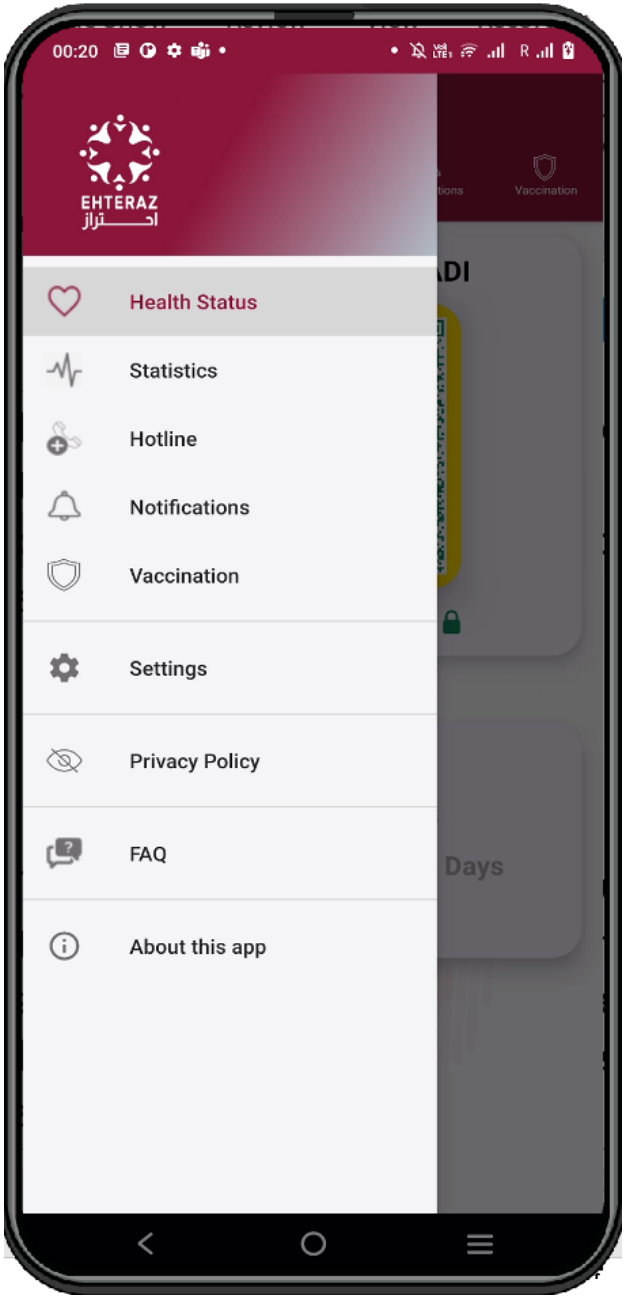
- A FAB performs the primary, or most common, action on a screen, such as drafting a new email
  - It appears in front of all screen content, typically as a circular shape with an icon in its center.
  - FAB is typically placed at the bottom right

```
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

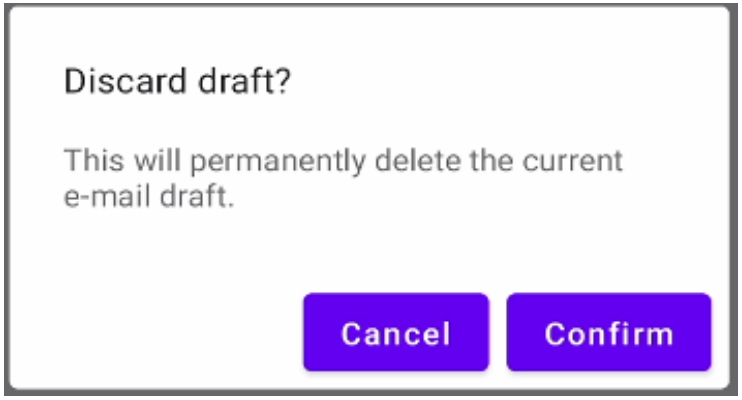
Floating Windows

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

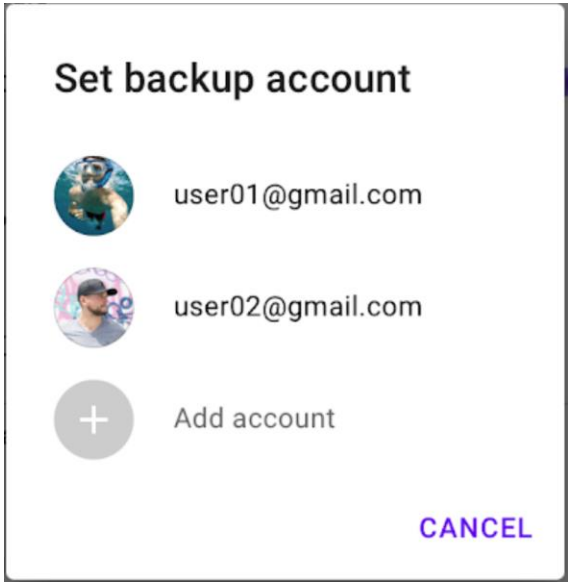
- Commonly used to **confirm high-risk actions** like deleting progress



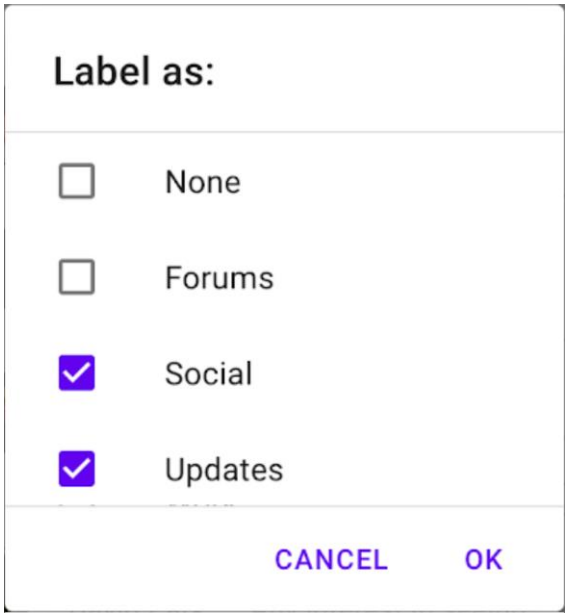

```
AlertDialog(
        onDismissRequest = {
           // Dismiss the dialog when the user clicks outside the dialog
           // or on the back button
             onDialogOpenChange(false)
        },
        title = { Text(text = title) },
        text = { Text(text = message) },
        confirmButton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

Simple dialog:

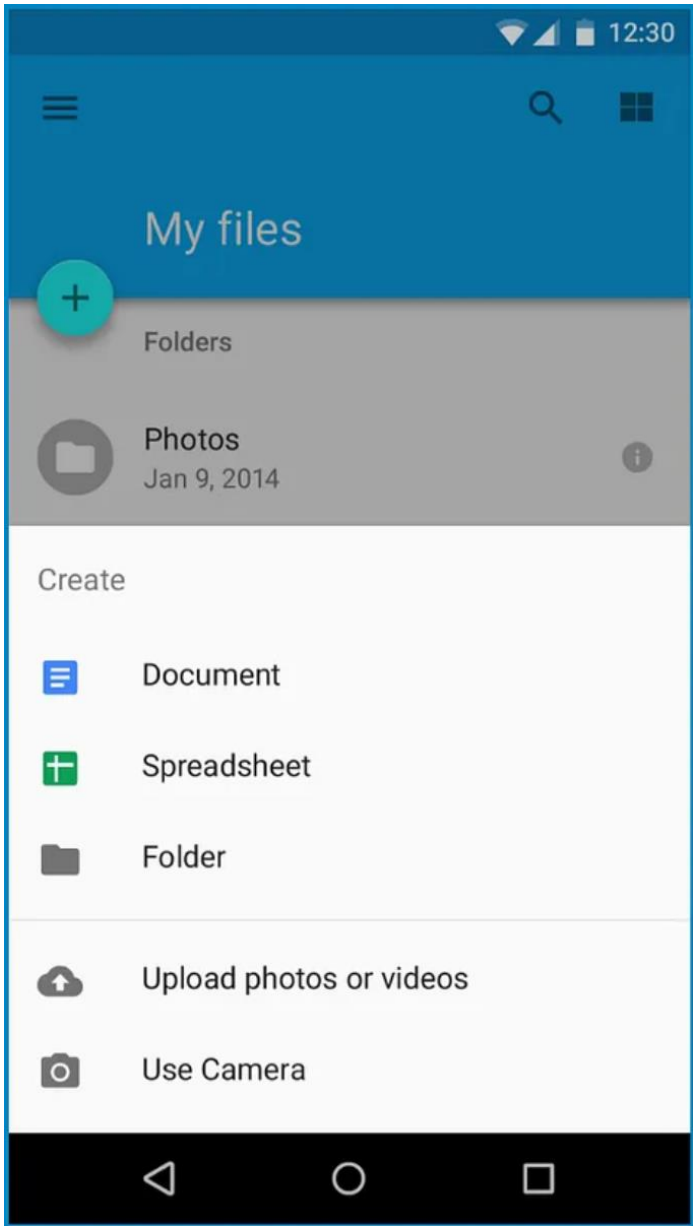
present the user with a list of actions that, when tapped, take immediate effect



Confirmation dialog (multi choice)



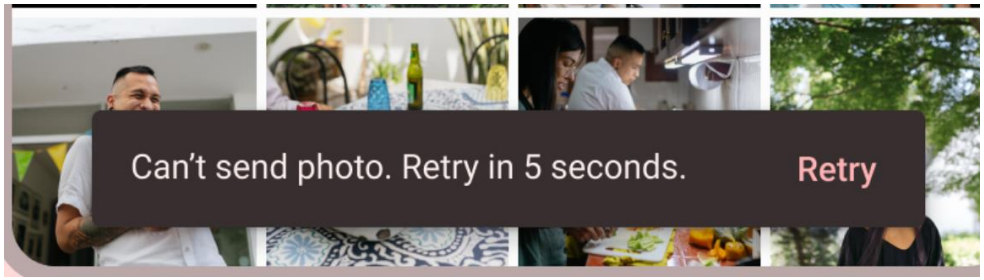
Bottom Sheets



- 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

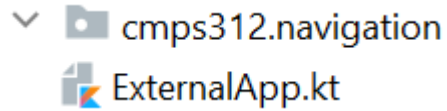
Snackbar

- 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"))
```

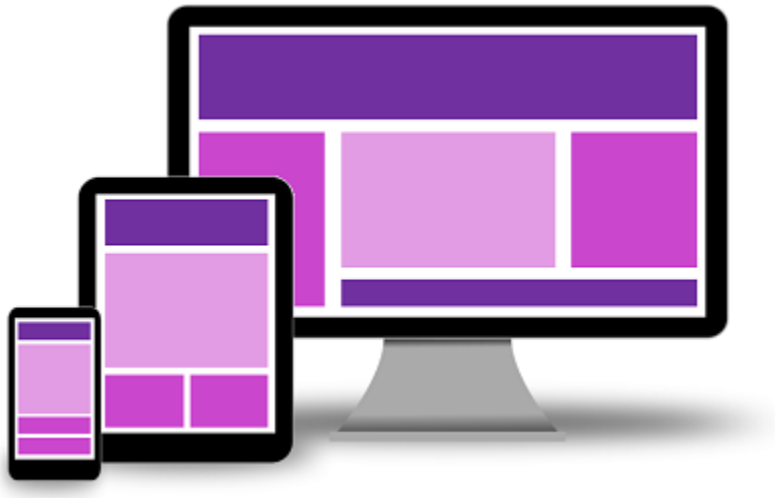
- Other common intents discussed here

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

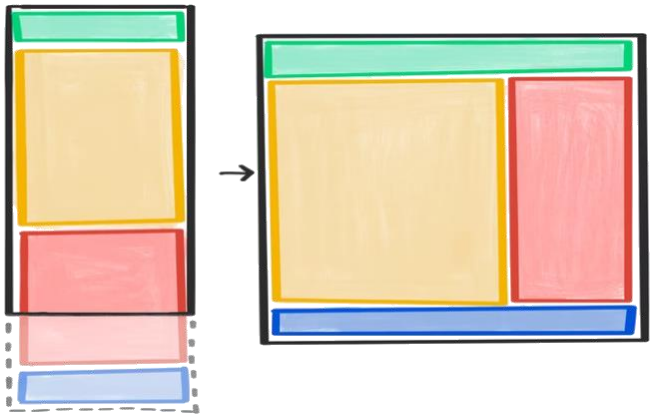
```
sealed class NavDestination(val route: String, val title: String, val icon: ImageVector? = null,
            val iconResourceId:Int? = null) {
    object Quran : NavDestination(route = "quran", title = "Quran", iconResourceId = R.drawable.ic_quran)
    object Verses : NavDestination(route = "verses",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

Responsive UI



Responsive UI

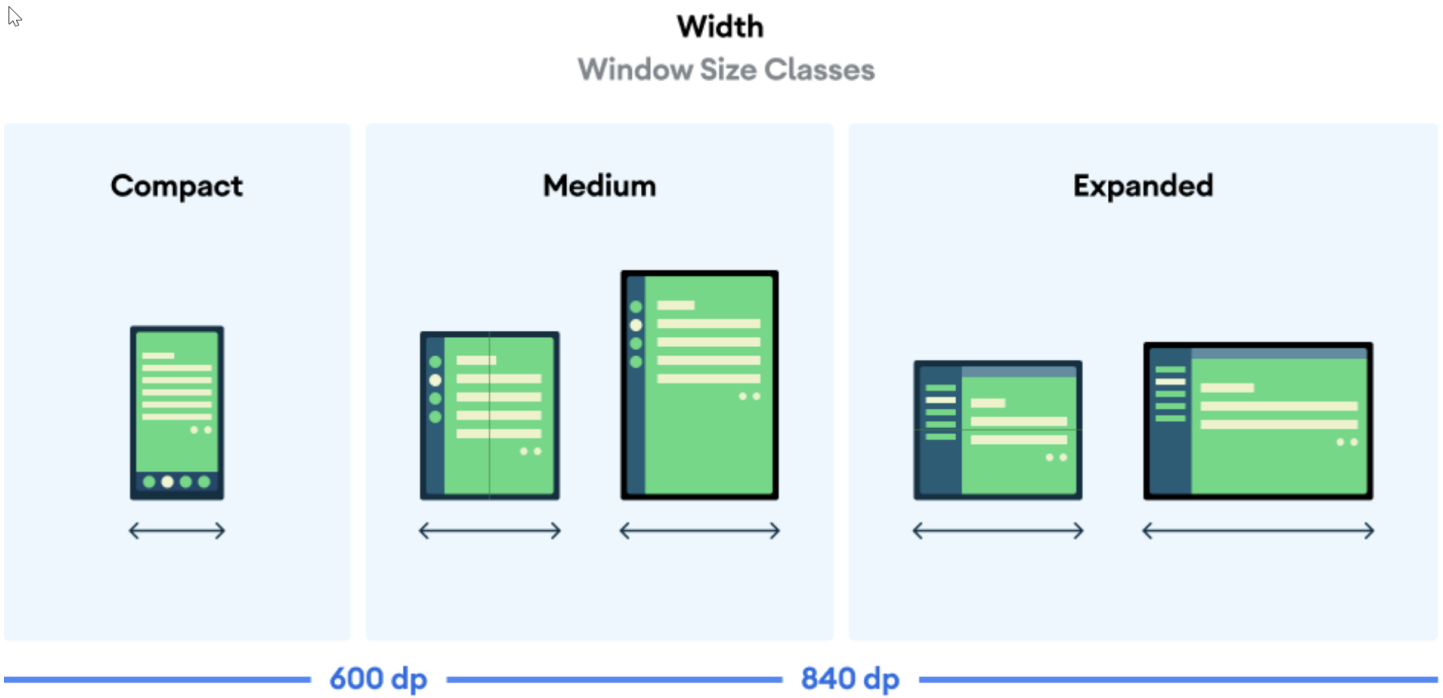
- Responsive UI = **serve different layouts for different screen sizes and orientations**
  - **Optimize the viewing experience on range of devices**: mobile, desktop, tablet, TV…
- For example, a newspaper app might have a single column of text on a mobile device, but display several columns on a larger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

Resources

- Jetpack Compose Navigation
  - https://developer.android.com/jetpack/compose/navigation
- Jetpack Compose Navigation codelab
  - https://developer.android.com/codelabs/jetpack-compose-navigation
- Responsive UI
  - https://m3.material.io/foundations/layout/applying-layout/window-size-classes