

# 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



# Single Activity with Multi-Screens

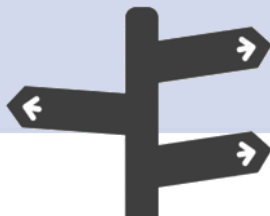
- App UI = { **1 Activity** + Multi-Screens }
  - A Screen is a composable that represents **a portion of the UI**
- The Navigation Component enables implementing Single Activity App with the ability to navigate between the app screens (also called **destinations**)
- Requires the following dependency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

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

# Navigation uses 2 main Classes

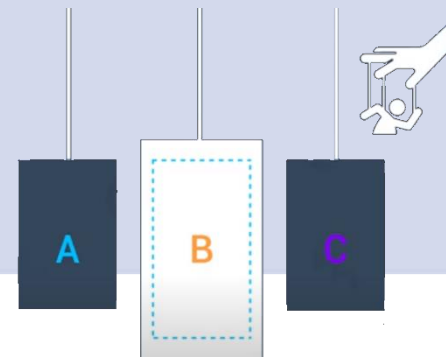
## NavHost

- 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***

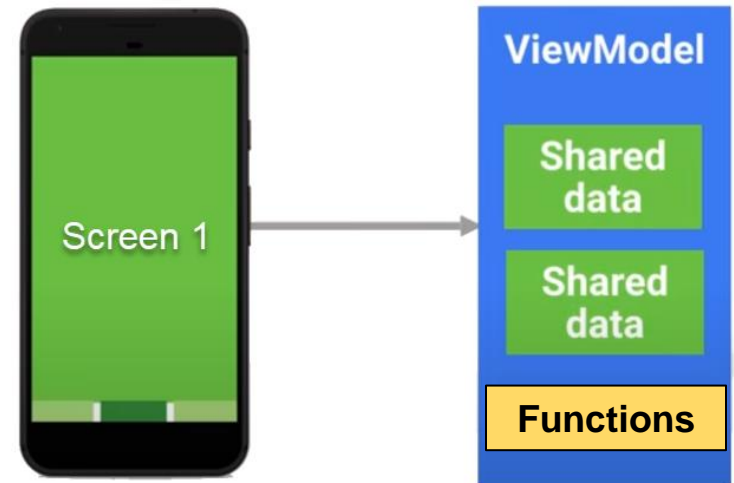
```
composable("profile?userId={userId}",
    arguments = listOf(navArgument("userId") {
        type = NavType.IntType
        defaultValue = 123 })
) { backStackEntry ->
    Profile(navController, backStackEntry.arguments?.getInt("userId"))
}
```

# 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 screen). To modify this behavior, pass **navigation options** to `navigate()` call
  - **`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")  
}
```

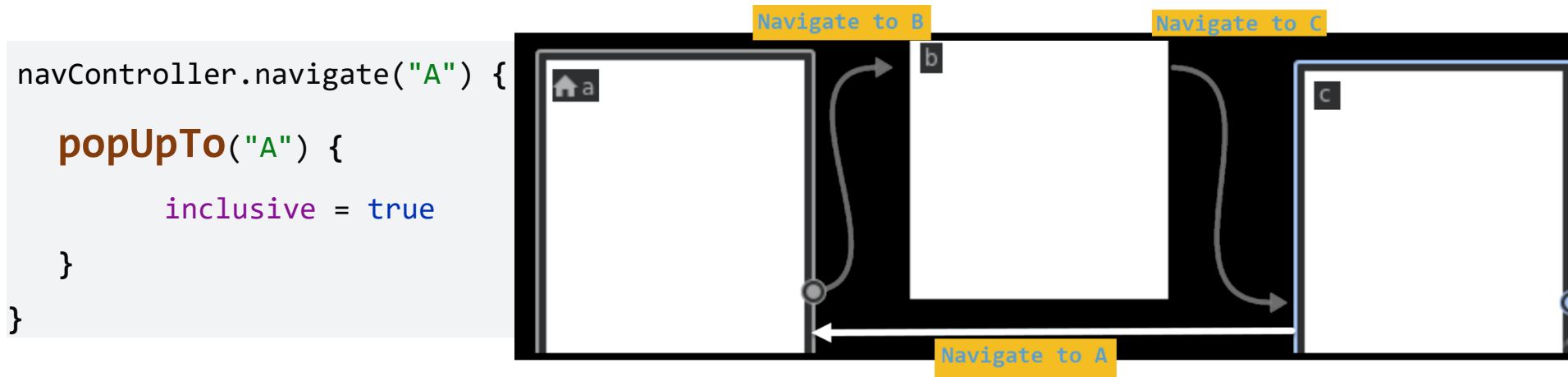
*/\* Pop off from the back stack up to and including the "home" destination before navigating to the "friends" destination \*/*

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

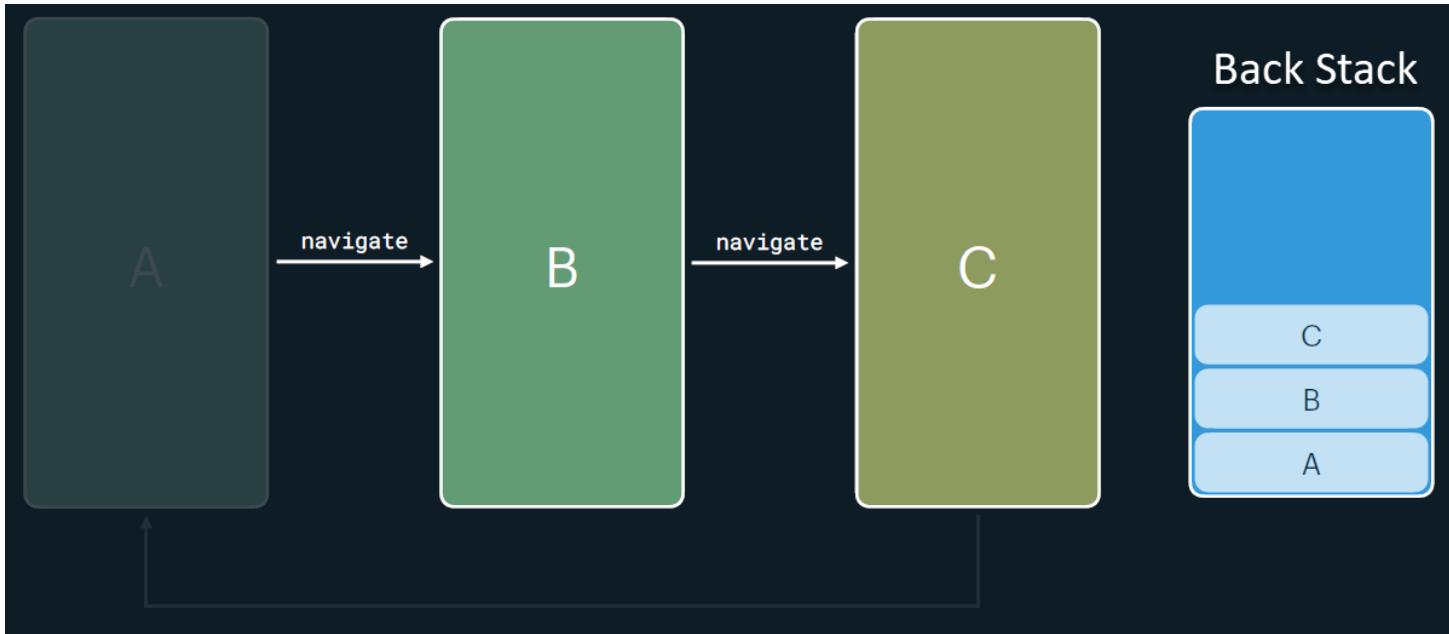
*/\* Navigate to the "search" destination only if we're not already on the "search" destination, avoiding multiple copies of the search screen on the back stack \*/*

```
navController.navigate("search") {  
    launchSingleTop = true  
}
```

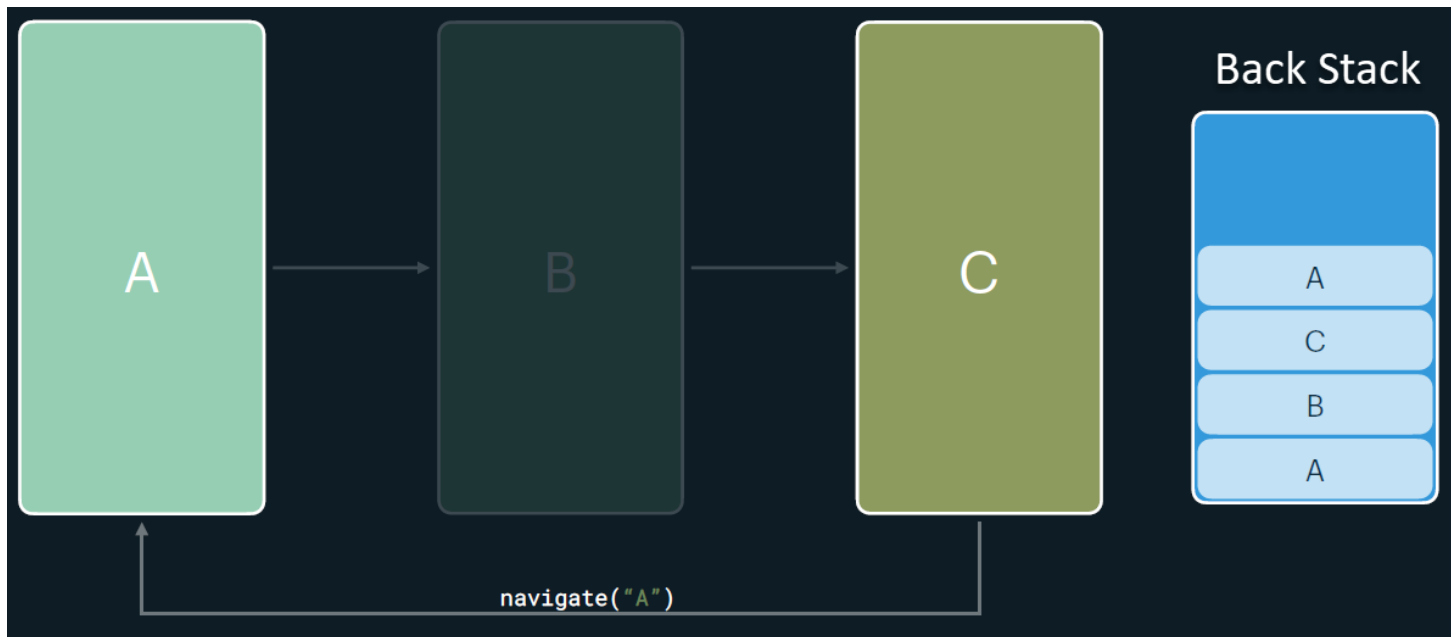
# popUpTo Example



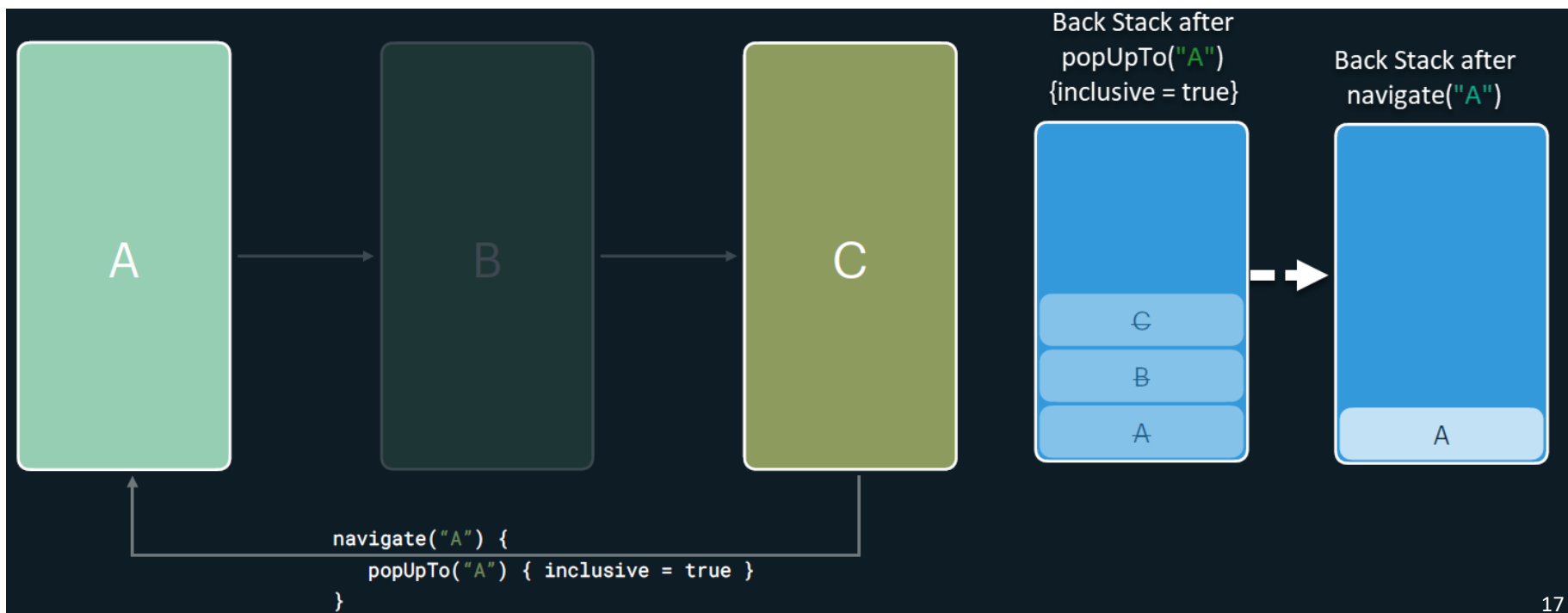
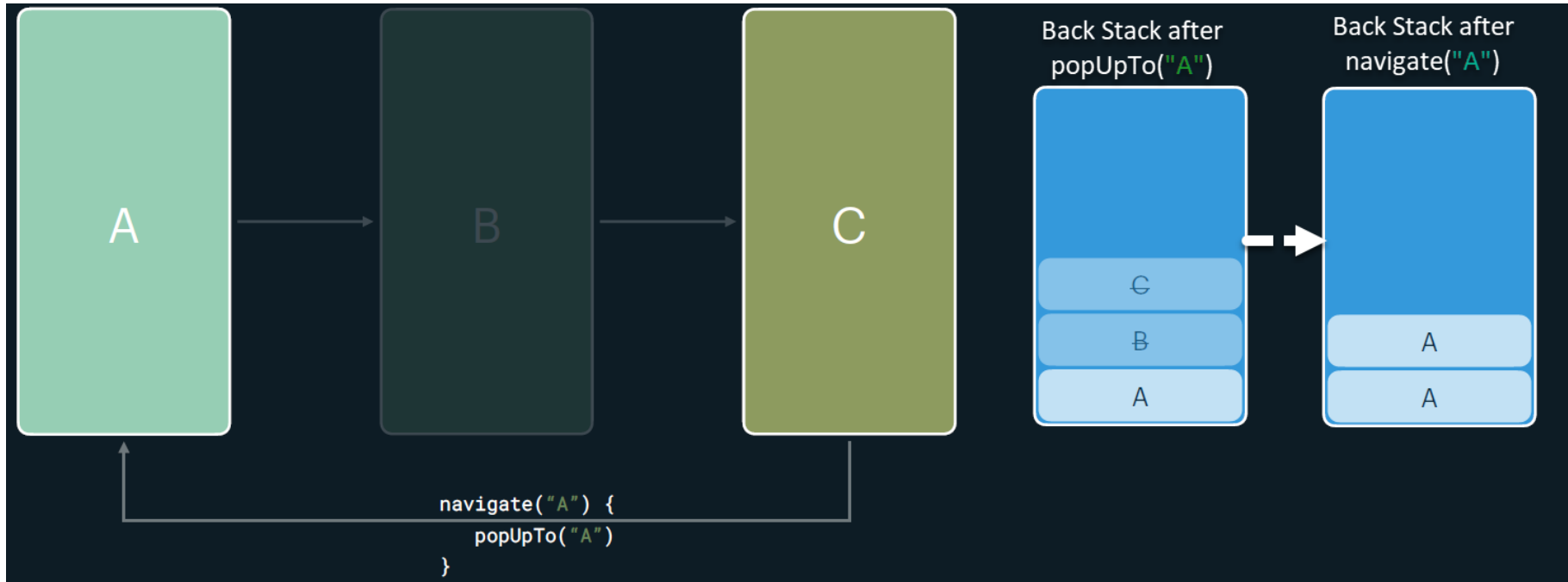
- After reaching C, the back stack contains (A, B, C). When navigating back to A, we also **popUpTo A**, which means that we remove B and C from the stack as part of the call to **navigate("A")**
  - With `inclusive= true`, we also pop off that first A of the stack to avoid having two instances of A



`navController.navigate("A")`

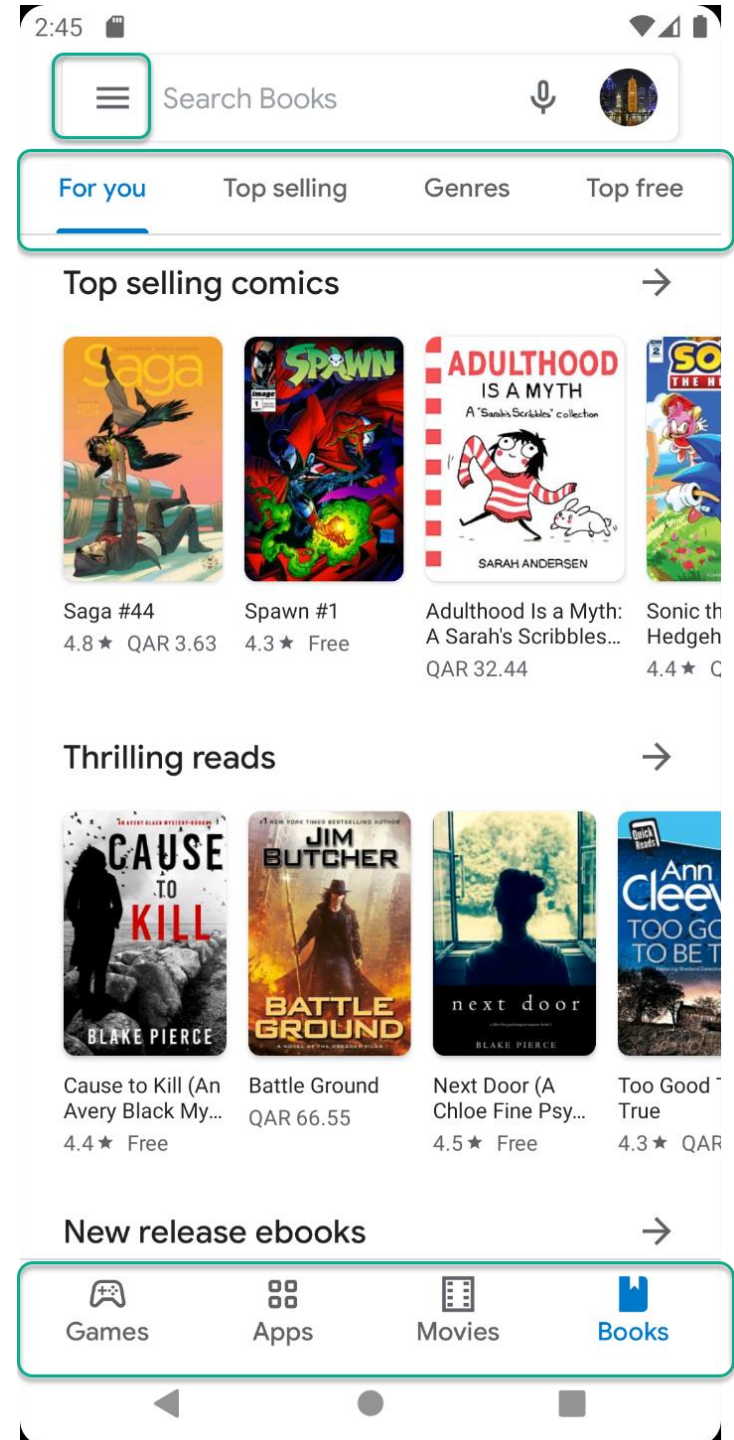






# Navigation UI Components:

- App Bars
- Navigation Rail
- Floating Action Button
- Navigation Drawer



# 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*)

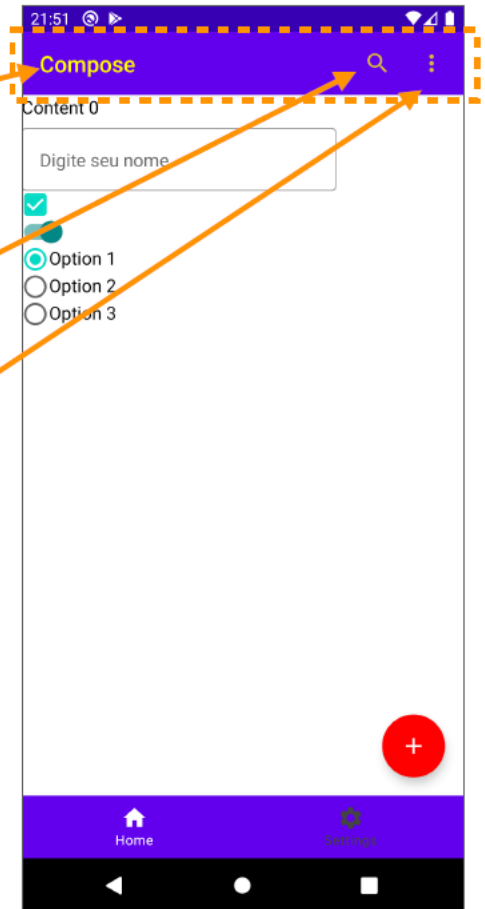
```
Scaffold(  
  topBar = {...},  
  floatingActionButton = {...},  
  bottomBar = {...}  
) {...}
```



# AppBar

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

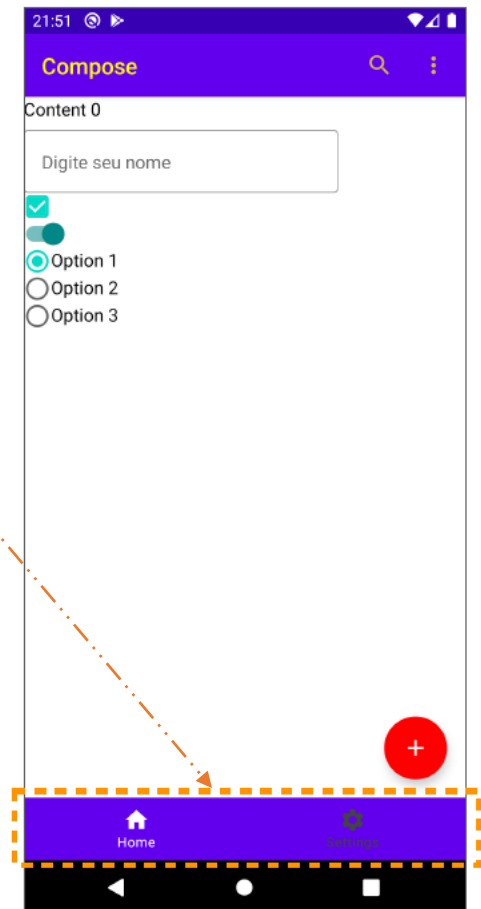
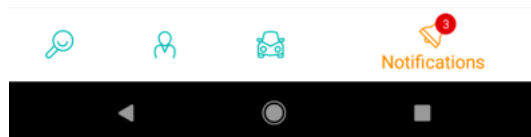
```
AppBar(  
  title = { Text(text = "Compose") },  
  backgroundColor = MaterialTheme.colors.primary,  
  contentColor = Color.Yellow,  
  actions = {  
    IconButton(onClick = {}) {  
      Icon(Icons.Default.Search, "Search")  
    }  
    IconButton(  
      onClick = { ... }  
    ) {  
      Icon(Icons.Filled.MoreVert, "More")  
      DropdownMenu(...)  
    }  
  }  
)
```



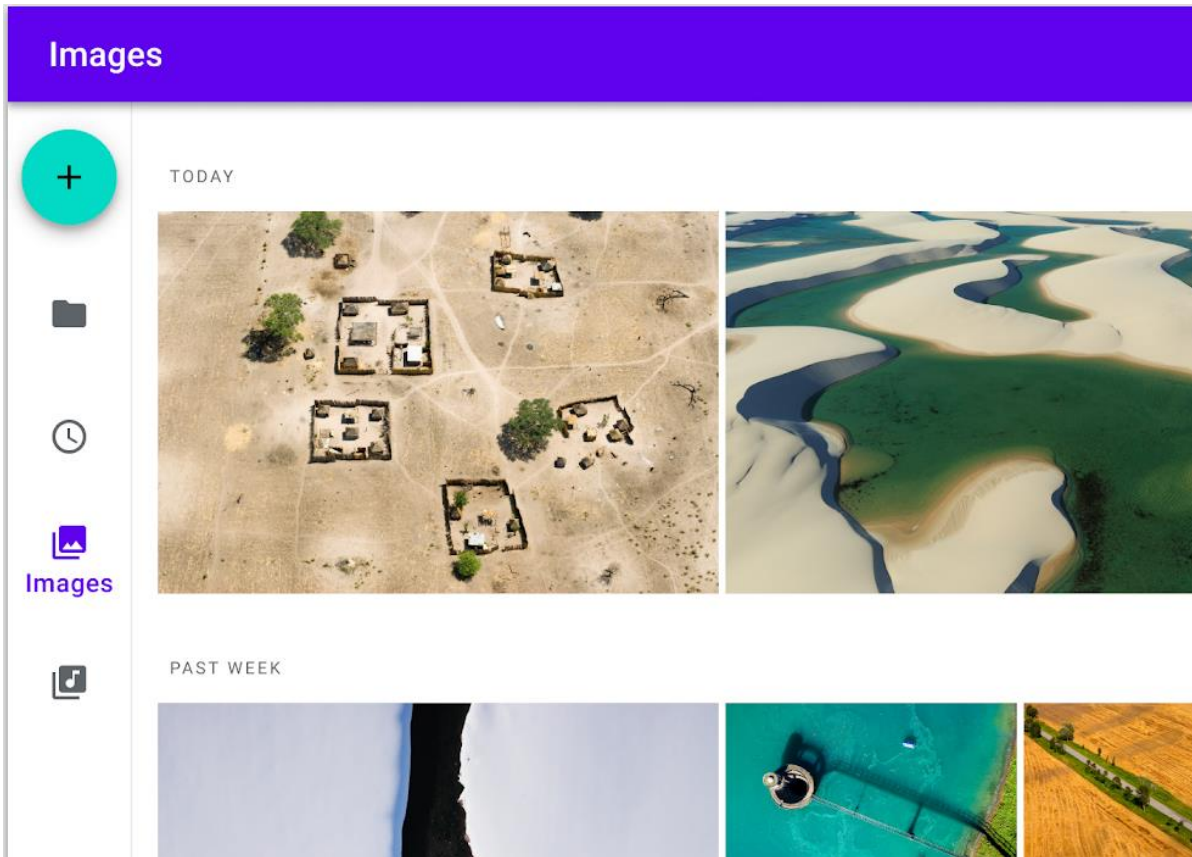
# BottomAppBar

- 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

```
BottomAppBar(  
  backgroundColor = MaterialTheme.colors.primary,  
  content = {  
    BottomNavigationItem(  
      icon = { Icon(Icons.Filled.Home) },  
      selected = selectedTab == 0,  
      onClick = { selectedTab = 0 },  
      selectedContentColor = Color.White,  
      unselectedContentColor = Color.DarkGray,  
      label = { Text(text = "Home") }  
    )  
    BottomNavigationItem(...)  
  }  
)
```



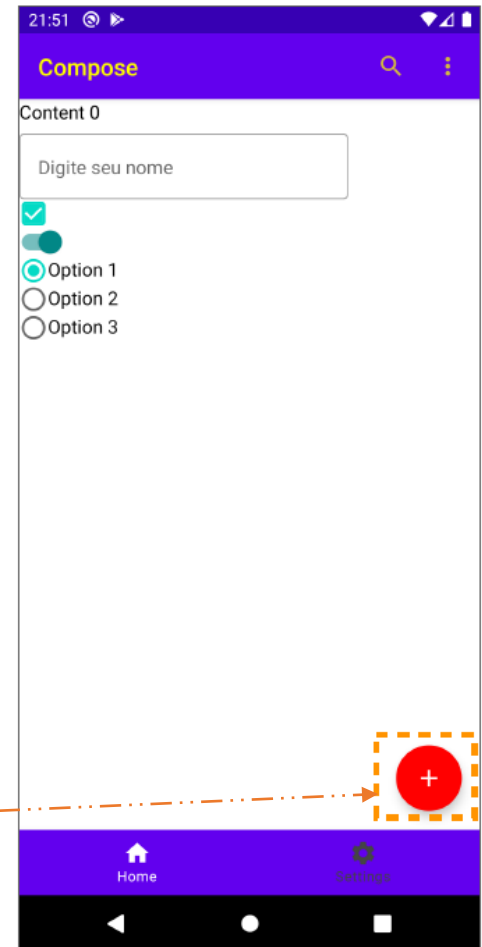
# Navigation Rail



# Floating Action Button (FAB)

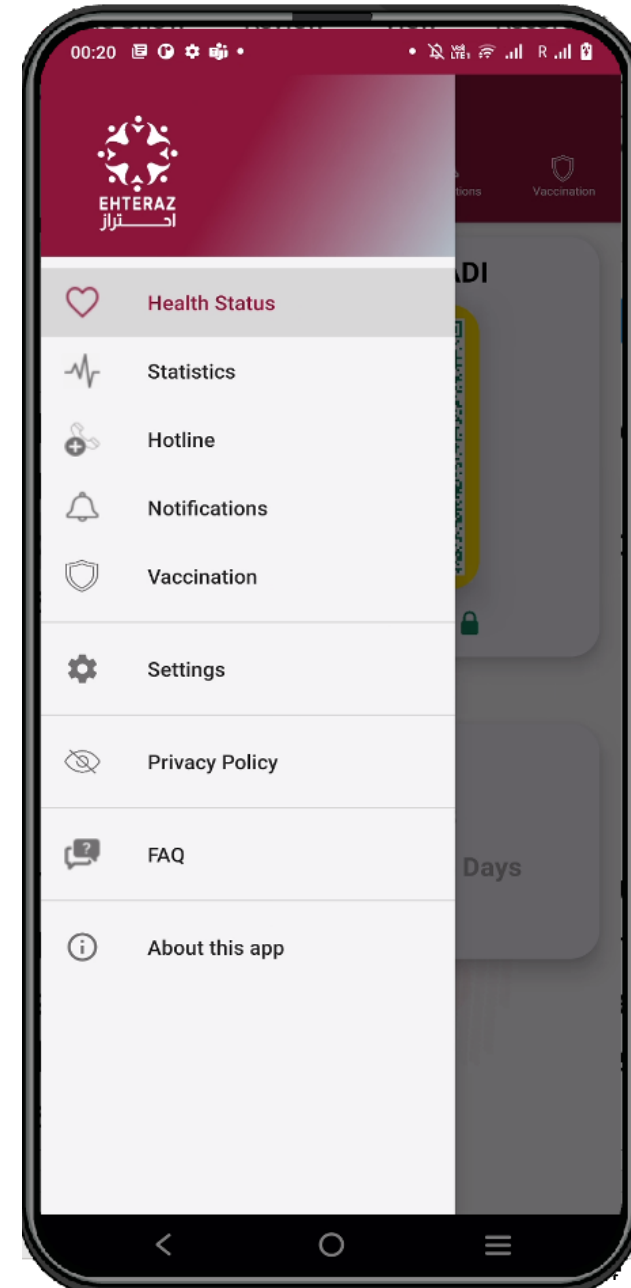
- A FAB performs the primary, or most common, action on a screen, such as drafting a new email
  - It appears in front of all screen content, typically as a circular shape with an icon in its center.
  - FAB is typically placed at the bottom right

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



# Navigation Drawer

- Navigation Drawer provides access to **primary 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

```
Column {  
    // Header  
    Image(  
        painter = painterResource(id = R.drawable.img_Logo),  
        ...  
    )  
  
    // Generate a Row for each navDrawer item  
    navDrawerItems.forEach { item ->  
        DrawerItem(item = item, onItemClick = { ... } )  
        ...  
    }  
}
```

- See more details in the posted navigation example

# Floating Windows



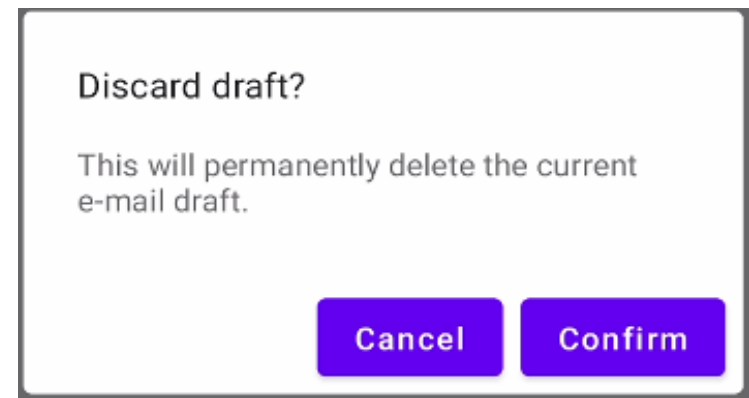
# Alert Dialog

- 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


## 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

**Set backup account**

 user01@gmail.com

 user02@gmail.com

 Add account

**CANCEL**

# Confirmation dialog (multi choice)

**Label as:**

☐ None

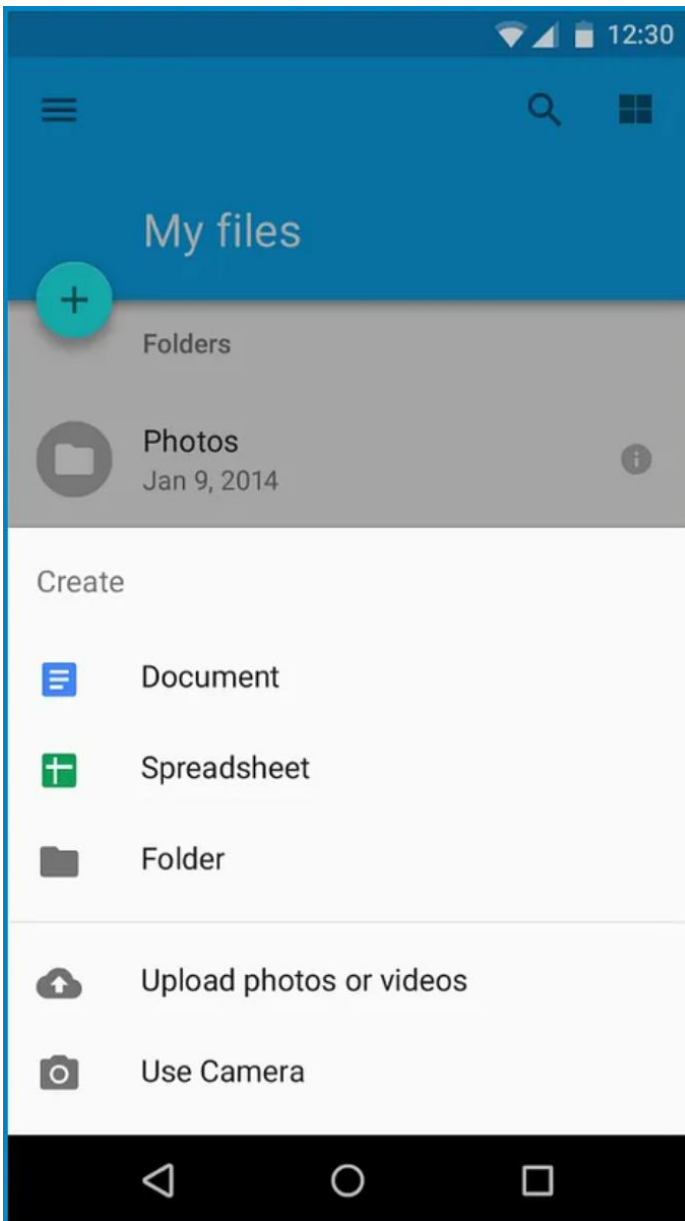
☐ Forums

☒ Social

☒ Updates

**CANCEL** **OK**

# Bottom Sheets



# Snackbar

# 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

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

- **Dial a number:**

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

- **Open a Uri**

```
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 Screens

- 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 screens as shown in the example below

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

# Resources

- Jetpack Compose Navigation
  - <https://developer.android.com/jetpack/compose/navigation>
- Jetpack Compose Navigation codelab
  - <https://developer.android.com/codelabs/jetpack-compose-navigation>