NAVIGATION SKILLS

Drawer, Tabs, Sliding Menu (header & footer)

Students:
Jacopo Maria Barile (64154)
Edoardo Campanella (64157)

Professors:
Jerome Baton

GitHub Link Repository: Here

Table of Contents

Introduction **1**
Tabs **2**
Drawers **4**
Sliding menu (header && footer) **9**
Header Implementation 9
Footer Implementation 10
Demo Application **11**
Application Overview 11
Key Features 11
Technical Implementation 12
Navigation Flow 12
Screenshots 13
References **13**

Introduction

In Android app development, navigation plays a vital role in shaping the user experience. It determines how users move from one screen to another and how different parts of an application interact. Whether the app consists of a few simple pages or a complex network of interfaces, clear and consistent navigation is essential to make the experience intuitive and seamless.

To help developers manage navigation efficiently, Android Studio provides a set of tools known as the **Jetpack Navigation Components**. These components simplify the process of moving between fragments, handling the back stack and passing data between screens-tasks that used to require a lot of manual code.

At the heart of this system is the **Navigation Graph**, a visual representation of all the screens (called *destinations*) in an app and the connections between them.
Developers can easily design navigation flows by dragging and linking destinations within this graph, making it easy to visualize how users will move through the app.

Each navigation graph is hosted by a **NavHostFragment**, which acts as a container for displaying the destinations defined in the graph. Then, the **NavController** manages the navigation within this host, determining which fragment to display and handling actions such as moving forward or backward. For instance, when a user taps a button to view details about an item, the NavController can trigger a transition from the home screen to a detail screen with just one line of code(1).

By mastering these navigation skills and components, developers can create Android applications that feel cohesive, predictable and user-friendly.
The Navigation Component framework not only reduces the amount of boilerplate code, but also provides a clear visual structure for building and maintaining complex navigation flows.

In the following sections we will talk about specific navigation components, such as drawers, tabs and sliding menu, along with a simple application demo to show the importance of such elements in android studio.

Tabs

In modern Android development, **Jetpack Compose** has revolutionized how user interfaces are built, replacing XML layouts with a declarative Kotlin-based approach.
Among its many UI elements, **Tabs** play a key role in organizing content and enabling smooth navigation between different sections of an app.

Tabs in Compose are implemented using **Tab** and **TabRow** composables, which together allow developers to create elegant, interactive and fully customizable tabbed interfaces. Unlike the traditional `TabLayout` and `ViewPager` system (2), Compose Tabs are built purely in Kotlin, offering more flexibility, less boilerplate code and tighter integration with state management.

By looking at the following code snippet we can fully understand how tabs composables are used in the navigation environment and why they are really useful and advantageous.

```kotlin
@Composable
fun NavigationTabExample(modifier: Modifier = Modifier) {
    val navController = rememberNavController()
    val startDestination = Destination.SONGS
    var selectedDestination by rememberSaveable { mutableIntStateOf(startDestination.ordinal) }
    Scaffold(modifier = modifier) { contentPadding ->
        PrimaryTabRow(selectedTabIndex = selectedDestination, modifier = Modifier.padding( paddingValues = contentPadding)) {
            Destination.entries.forEachIndexed { index, destination ->
                Tab(
                    selected = selectedDestination == index,
                    onClick = {
                        navController.navigate(route = destination.route)
                        selectedDestination = index
                    },
                    text = {
                        Text(
                            text = destination.label,
                            maxLines = 2,
                            overflow = TextOverflow.Ellipsis
                        )
                    }
                )
            }
        }
        AppNavHost(navController, startDestination)
```

At the start, the composable creates a `NavController` using `rememberNavController()`.
This controller manages the app's navigation graph and keeps track of which screen (or destination) is currently being displayed, with a `startDestination` that defines the **initial tab** that should be active when the app starts (in this case, `Destination.SONGS`) and a `selectedDestination` that is a **state variable** that tracks which tab is currently selected.

The `Scaffold` is a layout structure provided by Material Design that gives your UI a consistent visual hierarchy: It can contain elements like **top bars, bottom navigation, floating action buttons** and in this case a **tab row**.

The `PrimaryTabRow` composable (from **Material 3**) displays a horizontal row of tabs at the top of the screen: It visually indicates the currently active tab and handles layout for all the

tab items, with the parameter `selectTabIndex` to determine which tab appears highlighted on the screen.

Then, the function loops through a list of destinations (`Destination.entries`), which are likely defined as an **enum class** representing each section of the app (for example: Songs, Albums, Artists) and for each destination, a **Tab composable** is created:

- `selected` checks whether this tab corresponds to the currently selected one (`selectedDestination == index`).
- `onClick` defines what happens when the user taps on a tab:
    - It navigates to the corresponding route in the navigation graph using `navController.navigate()`.
    - It updates the `selectedDestination` state so the UI highlights the correct tab.
- `text` defines the tab's label, with styling to handle long titles (`maxLines = 2` and `TextOverflow.Ellipsis` ensure that text doesn't overflow visually).

This section essentially builds the **interactive navigation bar**, where tapping a tab updates the UI and changes the displayed content below.

`Finally AppNavHost` is a custom navigation host that uses the same `navController` to load the appropriate composable screen based on the current route.
For example: If the user selects the "Songs" tab, then it shows the `SongsScreen` or if they switch to "Albums", then it navigates to and displays the `AlbumsScreen`(3).

Drawers

A navigation drawer is a slide-in menu component that allows users to navigate to various sections of an app. Users can activate it by swiping from the side of the screen or by tapping a menu icon. This component follows Material Design principles and provides an intuitive way to organize app navigation in a space-efficient manner.

In Material Design, there are two types of navigation drawers (4): Standard drawers, which share space within a screen with other content, Modal drawers, which appear over the top of other content within a screen.

Navigation drawers serve multiple purposes in application design. They are particularly useful for content organization, enabling users to switch between different categories, such as in news or blogging apps. They also facilitate account management by providing quick links to account settings and profile sections in apps with user accounts.
Additionally, they support feature discovery by organizing multiple features and settings in a single menu, making them ideal for complex apps.

The `ModalNavigationDrawer` composable is used to implement a navigation drawer in Jetpack Compose. The basic structure involves using the `drawerContent` slot to provide a `ModalDrawerSheet` that contains the drawer's contents, like in the following example(5):

```
ModalNavigationDrawer(
    drawerContent = {
        ModalDrawerSheet {
            Text("Drawer title", modifier = Modifier.padding(16.dp))
            HorizontalDivider()
            NavigationDrawerItem(
                label = { Text(text = "Drawer Item") },
                selected = false,
                onClick = { /*TODO*/ }
            )
            // ...other drawer items
        }
    }
) {
    // Screen content
}
```

MaterialLayoutSnippets.kt

To control how the drawer opens and closes, `DrawerState` is used, which should be passed to `ModalNavigationDrawer` using the `drawerState` parameter.
These are suspending functions that require a `CoroutineScope`, typically instantiated using `rememberCoroutineScope()` (6):

```kotlin
val drawerState = rememberDrawerState(initialValue = DrawerValue.Closed)
val scope = rememberCoroutineScope()
ModalNavigationDrawer(
    drawerState = drawerState,
    drawerContent = {
        ModalDrawerSheet { /* Drawer content */ }
    },
) {
    Scaffold(
        floatingActionButton = {
            ExtendedFloatingActionButton(
                text = { Text("Show drawer") },
                icon = { Icon(Icons.Filled.Add, contentDescription = "") },
                onClick = {
                    scope.launch {
                        drawerState.apply {
                            if (isClosed) open() else close()
                        }
                    }
                }
            )
        }
    ) { contentPadding ->
        // Screen content
    }
}
```

MaterialLayoutSnippets.kt

At this (7) it's also possible to analyze a detailed navigation drawer.
The example contains multiple sections, icons, badges and proper state management:

- **Creating and Managing DrawerState:**

```kotlin
val drawerState = rememberDrawerState(initialValue = DrawerValue.Closed)
val scope = rememberCoroutineScope()
```

With this setup, you can:

- **Control the initial state**: By setting `initialValue` to `DrawerValue.Closed`, the drawer starts closed when the screen loads. It's possible to also use `DrawerValue.Open` to start with the drawer open.
- **Open the drawer programmatically**: Using `scope.launch { drawerState.open() }` allows you to open the drawer from anywhere in your code (same reasoning for close).
- **Check the current state**: The `drawerState.isClosed` and `drawerState.isOpen` properties let you verify whether the drawer is currently open or closed.
- **Toggle the drawer**: As shown in the IconButton's `onClick`, you can check the state and toggle accordingly.

The example also demonstrates how to organize the drawer content into logical sections:

- **Section Headers:**

```
Text("Section 1", modifier = Modifier.padding(16.dp), style = MaterialTheme.typography.titleMedium)
NavigationDrawerItem(
    label = { Text("Item 1") },
    selected = false,
    onClick = { /* Handle click */ }
)
```

It's possible to:

- **Create multiple sections**: Group related items together under descriptive headers.
- **Use different typography styles**: Apply `titleLarge` for the main drawer title and `titleMedium` for section headers to create visual hierarchy.
- **Add custom padding**: Control the spacing around section titles to improve readability.

- **Visual Separators:**

```
HorizontalDivider(modifier = Modifier.padding(vertical = 8.dp))
```

It's possible to:

- **Separate sections visually**: Use dividers between the title and content or between different sections.
- **Customize divider spacing**: Add padding to dividers to control the white space around them.
- **Create visual breathing room**: Proper use of dividers makes the drawer easier to scan.

The example shows two types of **navigation items**:

- **Simple Navigation Items:**

```
NavigationDrawerItem(
    label = { Text("Help and feedback") },
    selected = false,
    icon = { Icon(Icons.AutoMirrored.Outlined.Help, contentDescription = null) },
    onClick = { /* Handle click */ },
)
```

It's possible to:

- **Add text labels**: The `label` parameter accepts any composable, typically a Text component.
- **Handle clicks**: The `onClick` lambda is where you implement navigation logic.
- **Indicate selection state**: The `selected` parameter highlights which item corresponds to the current screen.

- **Enhanced Navigation Items with Icons and Badges:**

```
NavigationDrawerItem(
    label = { Text("Settings") },
    selected = false,
    icon = { Icon(Icons.Outlined.Settings, contentDescription = null) },
    badge = { Text("20") }, // Placeholder
    onClick = { /* Handle click */ }
)
```

It's possible to:

- **Add icons**: The `icon` parameter lets you add visual indicators that make items more recognizable.
- **Display badges**: The `badge` parameter can show notification counts, "new" indicators or any other small informational element.
- **Use different icon styles**: Choose from Filled, Outlined or other Material icon variants.
- **Implement auto-mirroring**: Use `Icons.AutoMirrored` for icons that should flip in right-to-left languages.

- **Scrollable Content:**

The Column wrapper shows how to handle overflow:

```
Column(
    modifier = Modifier.padding(horizontal = 16.dp)
        .verticalScroll(rememberScrollState())
) {
```

It’s possible to:

- **Make content scrollable**: The `verticalScroll` modifier ensures all items are accessible even on smaller screens.
- **Control horizontal padding**: Apply consistent padding to all drawer content.
- **Add vertical spacing**: Use `Spacer(Modifier.height(12.dp))` at the top and bottom for breathing room.

The example demonstrates how to **trigger the drawer from the app bar**:

```
TopAppBar(
    title = { Text("Navigation Drawer Example") },
    navigationIcon = {
        IconButton(onClick = {
            scope.launch {
                if (drawerState.isClosed) {
                    drawerState.open()
                } else {
                    drawerState.close()
                }
            }
        }) {
            Icon(Icons.Default.Menu, contentDescription = "Menu")
        }
    }
```

It’s possible to:

- **Add a menu button**: The hamburger icon in the top bar is the standard way to open drawers.
- **Toggle drawer state**: Check if it's closed and open it or vice versa.
- **Provide visual feedback**: The menu icon gives users a clear way to access navigation.
- **Use coroutines for smooth animations**: The `scope.launch` ensures the drawer opens/closes smoothly.

At the end, the `Scaffold` component allows us to

- **Integrate multiple UI elements**: It manages the top bar, drawer and main content together.
- **Handle padding automatically**: The `innerPadding` ensures content doesn't overlap with the top bar.
- **Pass content dynamically**: The `content` parameter allows the function to be reusable with different screen contents.
- **Add other Scaffold elements**: You could also add a bottom bar, floating action button or snackbar host.

Sliding menu (header && footer)

While the previous section covered the core structure and functionality of navigation drawers, a complete drawer implementation often benefits from dedicated header and footer sections that enhance both the visual hierarchy and user experience.

As mentioned earlier, the drawer content is organized within a `Column` that can be made scrollable using `verticalScroll`. Within this structure, headers and footers serve distinct purposes: the header establishes identity and context at the top of the drawer, while the footer provides supplementary information at the bottom.

Header Implementation

The header section typically appears at the very top of the `ModalDrawerSheet`, before any navigation items or section headers discussed previously.
A common pattern involves creating a dedicated composable function for the header(8):

```
// T for generic type to be used for the picking
@Composable
fun <T: Enum<T>> AppDrawerContent(
    drawerState: DrawerState,
    menuItems: List<AppDrawerItemInfo<T>>,
    defaultPick: T,
    onClick: (T) -> Unit
) {
    // default home destination to avoid duplication
    var currentPick by remember { mutableStateOf(defaultPick) }
    val coroutineScope = rememberCoroutineScope()

    ModalDrawerSheet {
        Surface(color = MaterialTheme.colorScheme.background) {
          Column(
              horizontalAlignment = Alignment.CenterHorizontally
          ) {
              // header image on top of the drawer
              Image(
                  painter = painterResource(id = R.drawable.app_icon),
                  contentDescription = "Main app icon"),
                  modifier = Modifier.size(150.dp)
              )
```

The header can contain various elements:

- **Brand identity**: App logo or icon that reinforces visual recognition.
- **User information**: Profile picture and username for personalized experiences in account-based applications.
- **Alignment and spacing**: Headers are commonly centered horizontally using `Alignment.CenterHorizontally` to create visual balance.

Footer Implementation

The footer section is positioned at the bottom of the drawer. A common implementation technique uses `Spacer(modifier = Modifier.weight(1f))` to push footer content to the bottom, ensuring it stays anchored even when the drawer content is scrollable(9):

```
Spacer(modifier = Modifier.weight( weight: 1f))
Text(
    text = "Developed by John Codeos",
    color = Color.White,
    textAlign = TextAlign.Center,
    fontWeight = FontWeight.Bold,
    modifier = Modifier
        .padding(12.dp)
        .align(Alignment.CenterHorizontally)
)
```

The key technique is `Spacer(modifier = Modifier.weight(1f))`, which expands to fill all available space between the navigation items and the footer, effectively anchoring the footer at the bottom regardless of the number of navigation items.
The footer typically contains:

- **Version information**: App version numbers for user reference and troubleshooting.
- **Developer credits**: Company name, developer attribution or copyright notices.
- **Legal information**: Links to terms of service or privacy policy.
- **Typography and styling**: Smaller font sizes and muted colors to differentiate from primary navigation elements.

Headers, navigation items and footers structure creates an organized navigation system where the header provides some elements at the top, the middle section offers functional navigation with proper visual hierarchy and the footer displays supplementary information at the bottom.

Demo Application

To demonstrate the practical implementation of Navigation Drawers, Tabs and Sliding Menus in Jetpack Compose, we developed a fully functional music streaming application.
The demo app follows modern Material Design 3 principles and showcases real-world usage patterns of these navigation components.

Application Overview

The Music Demo App is a Spotify-inspired interface that implements:

- **Modal Navigation Drawer** with a complete header, categorized menu items and footer section.
- **Tab-based navigation** within the Home screen to switch between Songs, Albums and Artists.
- **Integrated navigation system** using Jetpack Compose's NavController.

Key Features

Header Section: The drawer header displays user information including a circular profile avatar, username and subscription status:

Main Navigation: The drawer organizes destinations into two clear sections.
The main navigation section includes Home, Search and Your Library, providing access to primary app features. Each item features an icon, text label and visual feedback when selected through background color changes.

Library Section: A dedicated "YOUR LIBRARY" section groups related features including Liked Songs (with a distinctive green heart icon), Create Playlist and Your Podcasts.
This categorization follows Material Design principles for content organization and improves navigation discoverability.

Tab Navigation: The Home screen implements a `PrimaryTabRow` component with three tabs (Songs, Albums and Artists), each displaying a scrollable list of content cards.
The selected tab is highlighted with bold text and smooth transitions maintain state during navigation.

Footer Section: The drawer footer contains secondary actions (Settings button), version information and copyright notice. A `Spacer` with `Modifier.weight(1f)` ensures the footer remains anchored at the bottom regardless of content length.

Technical Implementation

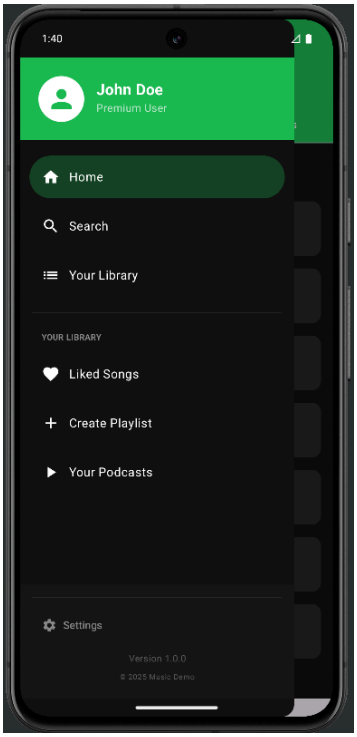
The application demonstrates several important Jetpack Compose patterns:

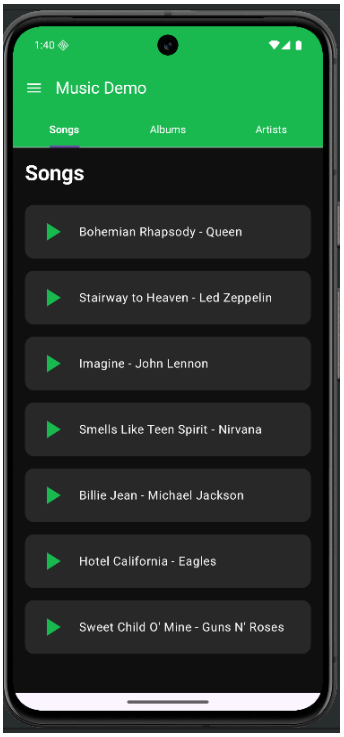
- **State Management:** Uses `rememberNavController()`, `rememberDrawerState()` and `rememberCoroutineScope()` to manage navigation and drawer state.
- **Nested Navigation:** Implements two separate `NavHost` components, one for drawer destinations and another for tab navigation within the Home screen.
- **Material Design 3:** Utilizes `ModalNavigationDrawer`, `ModalDrawerSheet`, `NavigationDrawerItem` and `PrimaryTabRow` composables.
- **Declarative UI:** All UI components are built using Compose's declarative approach
- **Coroutine Integration:** Drawer open/close animations are handled smoothly using Kotlin coroutines.

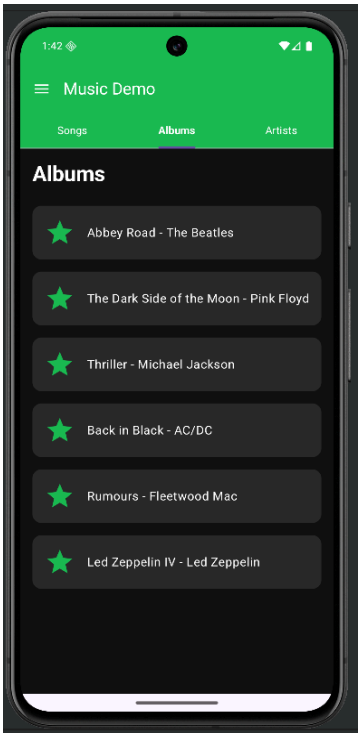
Navigation Flow

Users can access different sections through multiple paths: via the drawer menu (accessible through the hamburger icon), direct tab selection on the Home screen or the back navigation system. This allows us to demonstrate different possible options to implement our topics.
All navigation actions properly manage the back stack to ensure predictable behavior and state preservation during configuration changes.

Screenshots







References

- Navigation: https://developer.android.com/guide/navigation
- Tabs:https://abhiandroid.com/materialdesign/tablayout-example-android-studio.html#gsc.tab=0
- Tabs: https://developer.android.com/develop/ui/compose/components/tabs
- Drawers: https://developer.android.com/develop/ui/compose/components/drawer
- Header:https://tomas-repcik.medium.com/material-3-navigation-drawer-with-android-jetpack-compose-eda9285f9f4c
- Footer:https://johncodeos.com/how-to-create-a-navigation-drawer-with-jetpack-compose/