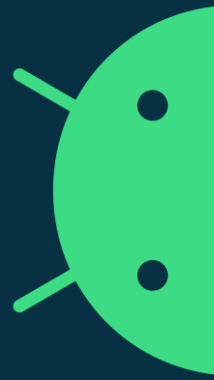
Navigation
CMPS 312



Navigating with Compose | Jetpack Compose | Android Developers

- Setup: Add the following dependency in your app module's build.gradle file.

```
implementation('androidx.navigation:navigation-compose:2.5.2')
```

- The **NavController** is the central API for the Navigation component. It is stateful and keeps track of the back stack of composables that make up the screens in your app and the state of each screen.
- You can access the state it provides via ***currentBackStackEntryAsState()***
- Each **NavController** must be associated with a single **NavHost** composable.
- The **NavHost** links the **NavController** with a *navigation graph* that specifies the composable destinations that you should be able to navigate between.
- Define a the **NavController** in the **MainActivity**.

```
private lateinit var navController: NavHostController
```

- Create the **NavController** inside the *setContent* and pass it to a Composable that you need to code to setup your navigation graph.
-

```
navController = rememberNavController()
```

```
SetupNavigation(navController)
```

NavHost

- Each composable destination in your navigation graph is associated with a route.
- Route is a String that defines the path to your composable. You can think of it as an implicit deep link that leads to a specific destination. Each destination should have a **unique** route.
- Creating the **NavHost** requires the **NavController** previously created via ***rememberNavController()*** and the route of the starting destination of your graph.
- NavHost creation uses the lambda syntax to construct your navigation graph.
- The navigation graph is constructed by calls to the composable() method.
- composable() method requires that you provide a route and the composable that should be linked to the destination.

```
NavHost(navController = navController, startDestination = "profile") {
    composable("profile") { Profile(/*...*/) }
    composable("friendslist") { FriendsList(/*...*/) }
    /*...*/
}
```

NavHost

- To navigate to a composable destination in the navigation graph, you must use the navigate method.

```
navController.navigate("friendslist")
```

- You can modify the behavior of navigate by attaching additional navigation options.

```
navController.navigate("friendslist") {
    popUpTo("home")
}
```

Pop everything up to the "home" destination off the back stack before navigating to the "friendslist" destination

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

Pop everything up to and including the "home" destination off the back stack before navigating to the "friendslist" destination

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

Navigate to the "search” destination only if we’re not already on the "search" destination, avoiding multiple copies on the top of the back stack

Navigate calls triggered by other composable functions

- The NavController's navigate function modifies the NavController's internal state.

```
@Composable
fun MyAppNavHost(
    modifier: Modifier = Modifier,
    navController: NavHostController = rememberNavController(),
    startDestination: String = "profile"
) {
    NavHost(
        modifier = modifier,
        navController = navController,
        startDestination = startDestination
    ) {
        composable("profile") {
            ProfileScreen(
                onNavigateToFriends = { navController.navigate("friendsList") },
                /*...*/
            )
        }
        composable("friendslist") { FriendsListScreen(/*...*/) }
    }
}
```

To comply with the single source of truth principle, only the composable function or state holder that hoists the NavController instance should make navigation calls.

Navigate calls triggered by other composable functions

```
@Composable
fun ProfileScreen(
    onNavigateToFriends: () -> Unit,
    /*...*/
) {
    /*...*/
    Button(onClick = onNavigateToFriends) {
        Text(text = "See friends list")
    }
}
```

Navigation events triggered from other composable functions lower in the UI hierarchy need to expose those events to the caller appropriately using functions..

You should only call navigate() as part of a callback and not as part of your composable itself, to avoid calling navigate() on every recomposition.

Navigate with arguments

- To pass an argument, add argument placeholders to your route. The argument type is String by default.

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

- The arguments parameter of composable() accepts a list of **NamedNavArguments**. You can quickly create a **NamedNavArgument** using the **navArgument** method and then specify its exact type

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

Navigate with arguments

- You should extract the arguments from the **NavBackStackEntry** that is available in the lambda of the ***composable()*** function.

```
composable("profile/{userId}") { backStackEntry ->
    Profile(navController,
backStackEntry.arguments?.getString("userId"))
}
```

- To pass the argument to the destination, you need to add append it to the route when you make the navigate call

```
navController.navigate("profile/user1234")
```

Adding optional arguments

- Optional arguments differ from required arguments in two ways:
  - They must be included using query parameter syntax ("?argName={argName}")
  - They must have a defaultValue set, or have nullability = true (which implicitly sets the default value to null)
- This means that all optional arguments must be explicitly added to the composable() function as a list.

```
composable(
    "profile?userId={userId}",
    arguments = listOf(navArgument("userId") { defaultValue = "user1234" })
) { backStackEntry ->
    Profile(navController, backStackEntry.arguments?.getString("userId"))
}
```

Integration with the bottom nav bar

- It is recommended to define a sealed class, such as Screen that contains the route and String resource ID for the destinations in your bottom nav bar

```
sealed class Screen(val route: String, @StringRes val resourceId: Int) {
    object Profile : Screen("profile", R.string.profile)
    object FriendsList : Screen("friendslist", R.string.friends_list)
}
```

- Then place those items in a list that can be used by the BottomNavigationItem

```
val items = listOf(
   Screen.Profile,
   Screen.FriendsList,
)
```