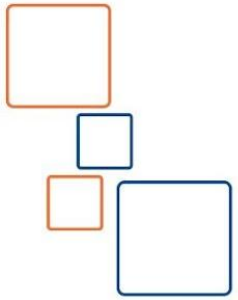
Android Applications Development Using Kotlin



Invest In Yourself,
Develop Your Career

Java™ Education and Technology Services

Course Agenda

Day1

- Jetpack Compose
- Basic UI(Text, Button, Image..)
- Lists
- Modifiers

Day2

- Permissions
- Locations

Day3

- Broadcast Receivers
- Services
  - Background Services
  - Foreground Services
  - Started Services
  - Bound Services
  - IntentService
  - JobIntentService
- Notifications

Day4

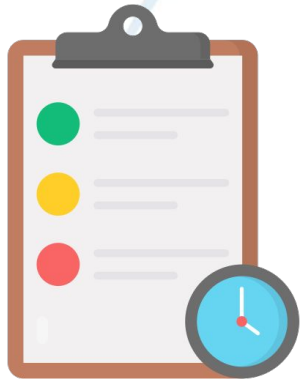
- WorkManager

Day5

- Using coroutines in Android
  - Retrofit
  - Room
  - WorkManager

Day6

- Navigation
- Layouts



Java™ Education and Technology Services

Navigation with Compose

Navigation Key Concepts

Concept	Purpose
Host	A UI element that contains the current navigation destination. That is, when a user navigates through an app, the app essentially swaps destinations in and out of the navigation host.
Graph	A data structure that defines all the navigation destinations within the app and how they connect together.
Controller	The central coordinator for managing navigation between destinations. The controller offers methods for navigating between destinations, handling deep links, managing the back stack, and more.
Destination	A node in the navigation graph. When the user navigates to this node, the host displays its content.
Route	Uniquely identifies a destination and any data required by it. You can navigate using routes. Routes take you to destinations.

How to Navigate

1. Add the needed dependencies
2. Create your screens (Destinations)
3. Define the navigation routes
4. Get a reference to the NavController
5. Call the NavHost within your activity's `setContent{ }`
6. Define the content for your NavHost using composable() function
7. Pass & receive the navigation arguments [if needed]

Dependencies

- In your build.gradle(module) add the following dependencies:

```
val nav_version = "2.8.8"
implementation("androidx.navigation:navigation-compose:$nav_version")
//Serialization for NavArgs
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.8.0")
```

- Also, in your build.gradle(module) plugin's section add:

```
kotlin("plugin.serialization") version "2.1.10"
```

Define Routes for Destinations in your app

- A route is a string that corresponds to a destination.
- This idea is similar to the concept of a URL. Just as a different URL maps to a different page on a website, a route is a string that maps to a destination and serves as its unique identifier.
- There are a finite number of screens in an app, so there are also a finite number of routes.
- You **can define an app's routes using an enum class or Sealed class**.

Define routes with Type Safe Navigation

- You need to define serializable classes or objects that represent your routes.

- To define serializable objects use `@Serializable` annotation provided by the Kotlin Serialization plugin. This plugin can be added to your project by adding these dependencies.

- Use the following rules to decide what type to use for your route:
  1. Object Declaration: Use an object for routes without arguments.
  2. Class: Use a class or data class for routes with arguments.

Routes with Type Safe Navigation - Example

```
// Define a home route that doesn't take any arguments
@Serializable
object Home

// Define a profile route that takes an ID
@Serializable
data class Profile(val id: String)
```

Define Routes for Destinations

```
@Serializable
sealed class ScreenRoute(){
    @Serializable
    object HomeScreenRoute :
ScreenRoute ()

    @Serializable
    object LoginScreenRoute :
ScreenRoute ()

    @Serializable
    object SignupScreenRoute :
ScreenRoute ()
}
```

Getting NavController

- To create a NavController when using Jetpack Compose, call rememberNavController():

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

- **You should create the NavController high in your composable hierarchy**. It needs to be high enough that all the composables that need to reference it can do so.

Add a NavHost to your app

- A NavHost is a Composable that displays other composable destinations, based on a given route.

- The syntax for NavHost is just like any other Composable.

NavHost Anatomy

```
NavHost(
        navController,
        startDestination,
        modifier,
) {
        content
}
```

An instance of the NavHostController. Used to navigate between screens. You can obtain it by calling rememberNavController() from a composable function.

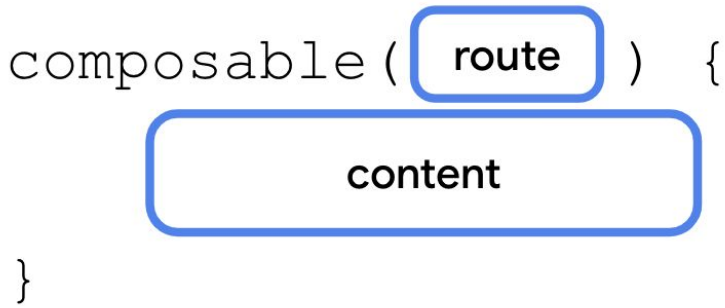
A string route defining the destination shown when the app first displays the NavHost

Like other composables, NavHost also takes a modifier parameter.

Composable functions that maps the defined route to the corresponding Composable functions that shows the UI

NavHost Content

- Within the content function of a NavHost, you call the composable() function.
- The composable() function has two required parameters.
  1. route: A string corresponding to the name of a route. This can be any unique string
  2. content: Here you can call a composable that you want to display for the given route.



NavHost

```
NavHost(
    navController,
    startDestination,
    modifier,
) {
    content
}
```

```
NavHost(
    navController = rememberNavController(),
    startDestination = Screen.LoginScreen,
    modifier = modifier
) {
    composable<Screen.LoginScree
        LoginUI( )
    }
    composable<Screen.SignupScre
        SignUpUI( )
    }
    composable<Screen.HomeScreer
        HomeUI( )
    }
}
```

Content

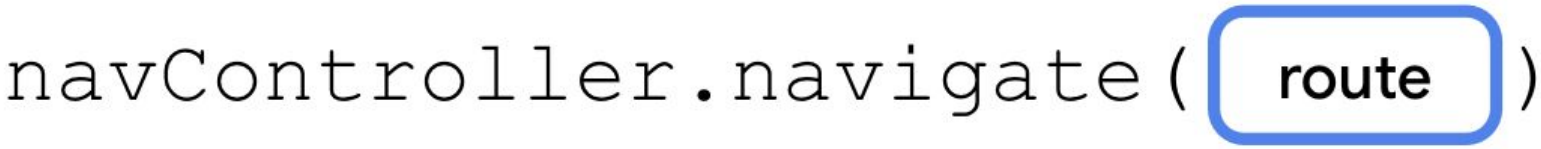
```
composable( route ) {
    content
}
composable( route ) {
    content
}
composable( route ) {
    content
}
```

Start Navigation

- our approach is to pass a function type into each composable for what should happen when a user clicks the button.
- That way, the composable and any of its child composables decide when to call the function.
- However, navigation logic isn't exposed to the individual screens in your app. All the navigation behavior is handled in the NavHost.

Navigate to A destination

- To navigate to another route, simply call the navigate() method on your instance of NavHostController.



- The navigate method takes a single parameter: a String corresponding to a route defined in your NavHost.
- If the route matches one of the calls to `composable()` in the NavHost, the app then navigates to that screen.

Pop to the start screen

- To remove all screens from the back stack and return to the starting screen.
- You can do this by calling the popBackStack() method.

```
navController.popBackStack( route , inclusive )
```

- The popBackStack() method has two required parameters.
  - route: The string representing the route of the destination you want to navigate back to.
  - inclusive: A Boolean value that, if true, also pops (removes) the specified route. If false, popBackStack() will remove all destinations on top of—but not including—the start destination, leaving it as the topmost screen visible to the user.(Whether the given destination should also be popped)

Passing Arguments between Screens

Build your graph

- Use the **`composable()`** function to define composables as destinations in your navigation graph.

```
NavHost(navController, startDestination = Home) {
        composable<Home> {
            HomeScreen(onNavigateToProfile = { id ->
                navController.navigate(Profile(id))
            })
        }
        composable<Profile> { backStackEntry ->
            val profile: Profile = backStackEntry.toRoute()
            ProfileScreen(profile.id)
        }
}
```

`toRoute()`: recreates the Profile object from the NavBackStackEntry and its arguments

Navigate to type safe route

- Finally, you can navigate to your composable using the navigate() function by passing in the instance of the route:

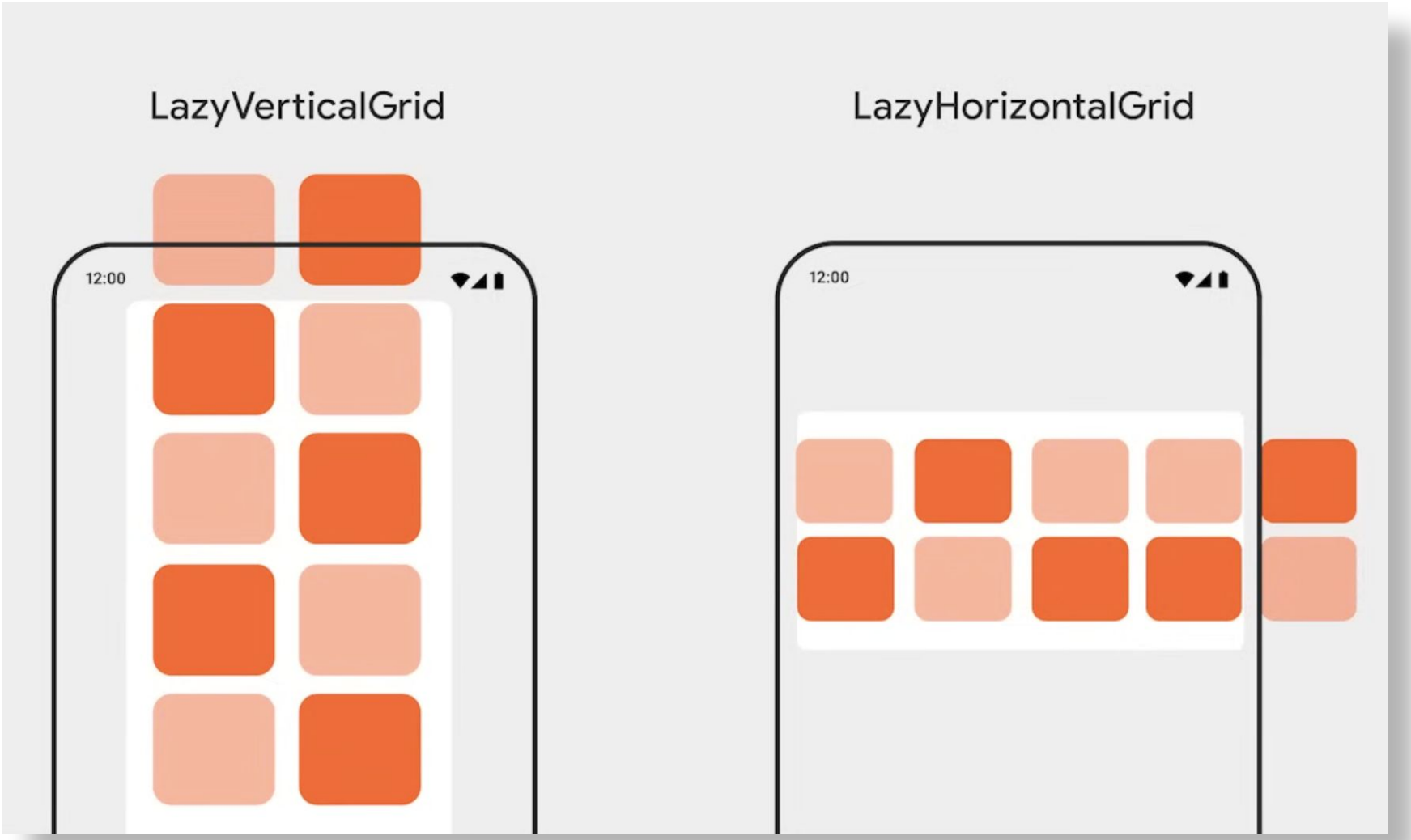
```
navController.navigate(Profile(id = 123))
```

- This navigates the user to the composable<Profile> destination in the navigation graph. Any navigation arguments, such as id, can be obtained by reconstructing Profile using NavBackStackEntry.toRoute and reading its properties.

Java™ Education and Technology Services

Lazy Grid

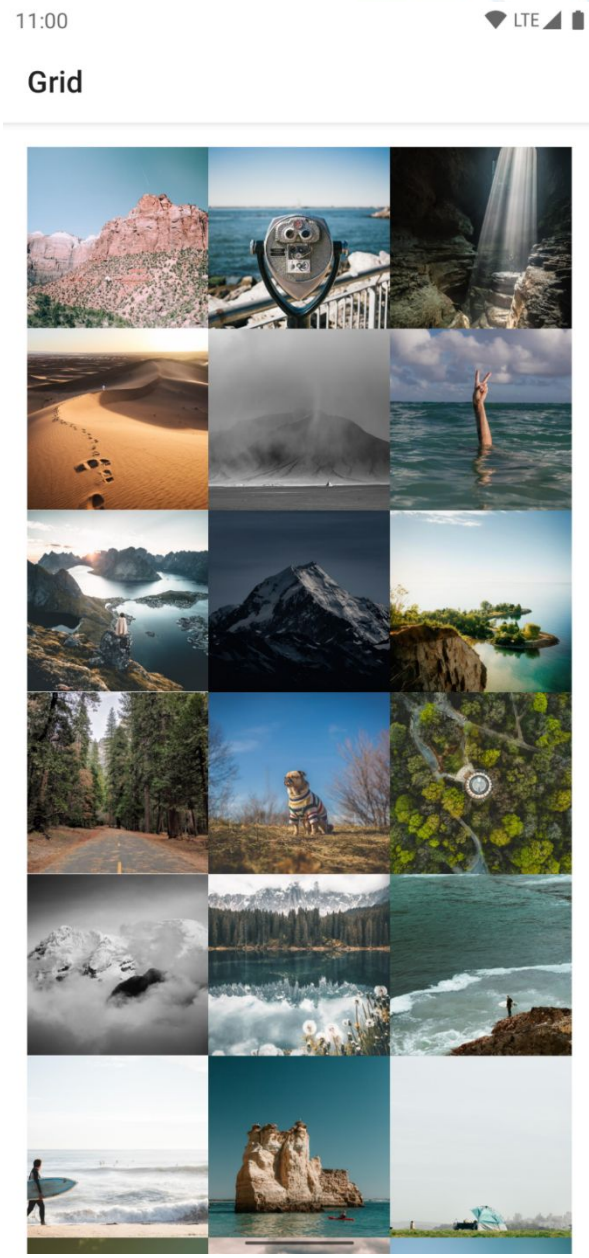
Lazy Grid



Lazy Grid

The **LazyVerticalGrid** and **LazyHorizontalGrid** composables provide support for displaying items in a grid.

A Lazy vertical grid will display its items in a vertically scrollable container, spanned across multiple columns, while the Lazy horizontal grids will have the same behaviour on the horizontal axis.



Lazy Grid

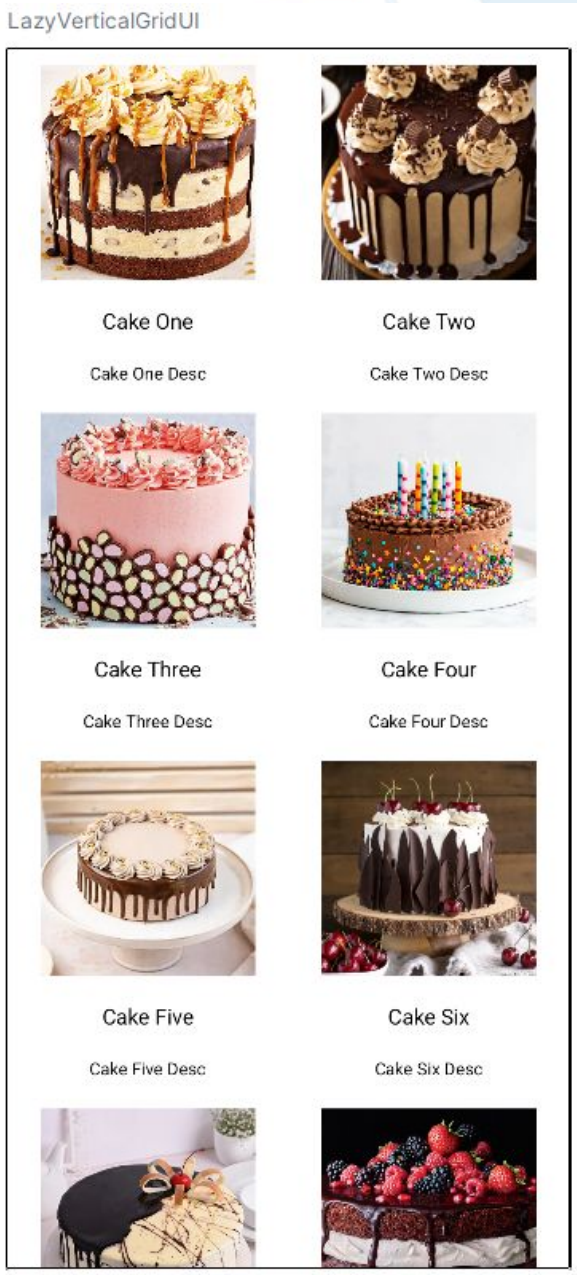
- The columns parameter in **LazyVerticalGrid** and rows parameter in **LazyHorizotalGrid** control how cells are formed into columns or rows by **GridCells.Adaptive(size) or GridCells.Fixed(count)**. The following example displays items in a grid, using **GridCells.Adaptive(size)** to set each column to be at least 128.dp wide

```
LazyVerticalGrid( columns = GridCells.Adaptive (minSize = 128.dp) ) {
     items(photos) { photo -> PhotoItem(photo) }
}
LazyHorizontalGrid( rows = GridCells.Fixed (2) ) {
     items(photos) { photo -> PhotoItem(photo) }
}
```

Lazy Grid has many of the Lazy List's

LazyVerticalGrid

```
@Composable
fun LazyVerticalGridUI(list: List<Cake>) {
    LazyVerticalGrid(GridCells.Fixed(2),
          modifier = Modifier.fillMaxSize())
{
        items(list.size){
            GridItem(list.get(it))
        }
    }
}
```



Lazy staggered grid

- LazyVerticalStaggeredGrid and LazyHorizontalStaggeredGrid are composables that allow you to create a lazy-loaded, staggered grid of items.

- A lazy vertical staggered grid displays its items in a vertically scrollable container that spans across multiple columns and allows individual items to be different heights.

- Lazy horizontal grids have the same behavior on the horizontal axis with items of different widths

LazyVerticalStaggeredGrid

```
@Preview
@Composable
fun LazyVerticalStaggeredGridUI(list: List<Int>) {
    LazyVerticalStaggeredGrid(StaggeredGridCells.Fixed(2),
                    modifier = Modifier.fillMaxSize())
{
        items(list.size){
            val current = list.get(it)
            Image(painter = painterResource(current),
           contentDescription = "Content",
           contentScale = ContentScale.Inside)
        }
    }
}
```

Java™ Education and Technology Services

Scaffold

Introduction

- In Material Design, a scaffold is a fundamental structure that provides a standardized platform for complex user interfaces.

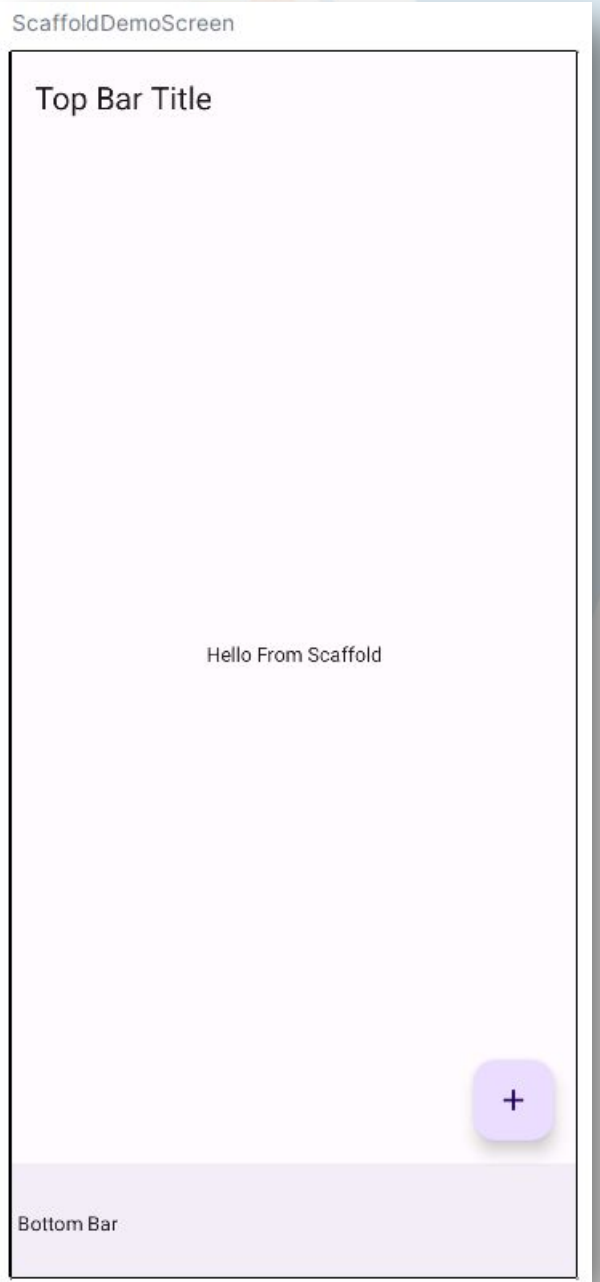
- It holds together different parts of the UI, such as app bars and floating action buttons, giving apps a coherent look and feel. Like:
  - topBar: The app bar across the top of the screen.
  - bottomBar: The app bar across the bottom of the screen.
  - floatingActionButton: A button that hovers over the bottom-right corner of the screen that you can use to expose key actions

Scaffold Function Parameters

```
@Composable
fun Scaffold(
    modifier: Modifier = Modifier,
    topBar: @Composable () -> Unit = {},
    bottomBar: @Composable () -> Unit = {},
    snackbarHost: @Composable () -> Unit = {},
    floatingActionButton: @Composable () -> Unit = {},
    floatingActionButtonPosition: FabPosition = FabPosition.End,
    containerColor: Color = MaterialTheme.colorScheme.background,
    contentColor: Color = contentColorFor(containerColor),
    contentWindowInsets: WindowInsets = ScaffoldDefaults.contentWindowInsets,
    content: @Composable (PaddingValues) -> Unit
) {
```

Simple Screen Using Scaffold

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ScaffoldDemoScreen(modifier: Modifier = Modifier) {
    Scaffold (
        topBar = { TopAppBar(title = { Text(text = "Top Bar Title") })},
        bottomBar = { BottomAppBar(){ Text("Bottom Bar") } },
        floatingActionButton = { FloatingActionButton(onClick = {}){
            Icon(painter = painterResource(R.drawable.baseline_add_24),
         contentDescription = "add") }
    },
        floatingActionButtonPosition = FabPosition.End)
    { contentPadding →
       Column(modifier = Modifier.fillMaxSize().padding(contentPadding),
           horizontalAlignment = Alignment.CenterHorizontally,
           verticalArrangement = Arrangement.Center)
    {
           Text(text = "Hello From Scaffold")
       }
    }
}
```



Self Study Topics

Scaffold

- https://medium.com/@ramadan123sayed/understanding-scaffold-in-jetpack-compose-a-comprehensive-guide-e248c2406412
- https://developer.android.com/develop/ui/compose/components/scaffold

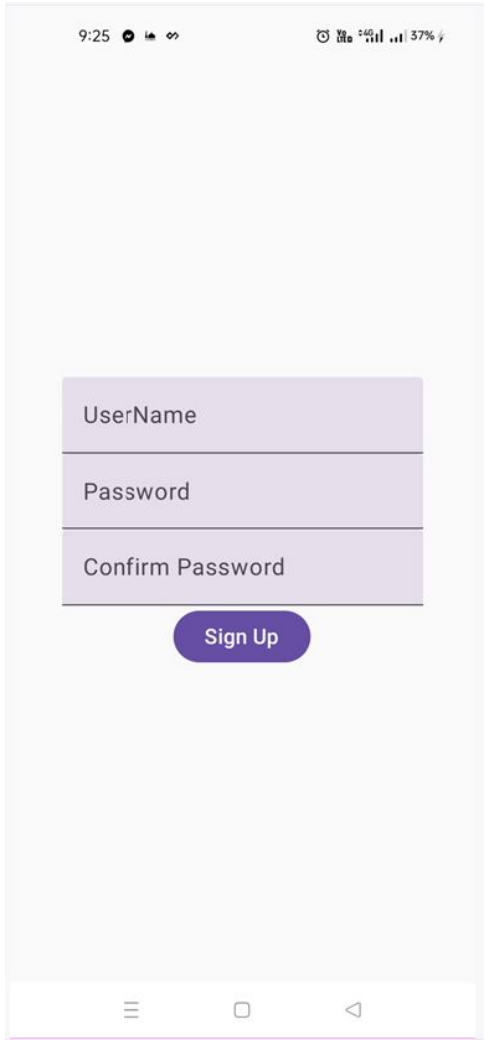
BottomNavBar with Navigation

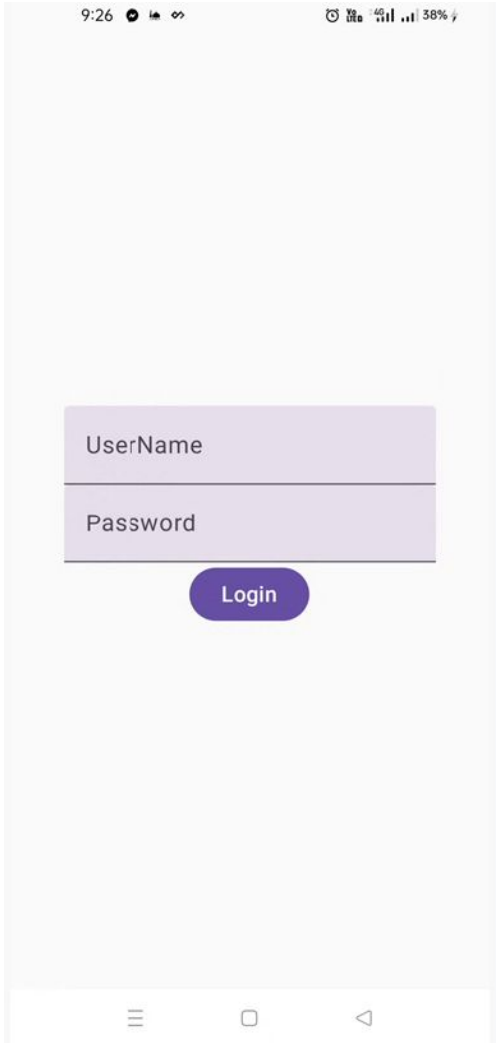
- https://medium.com/@santosh_yadav321/bottom-navigation-bar-in-jetpack-compose-5b3c5f2cea9b

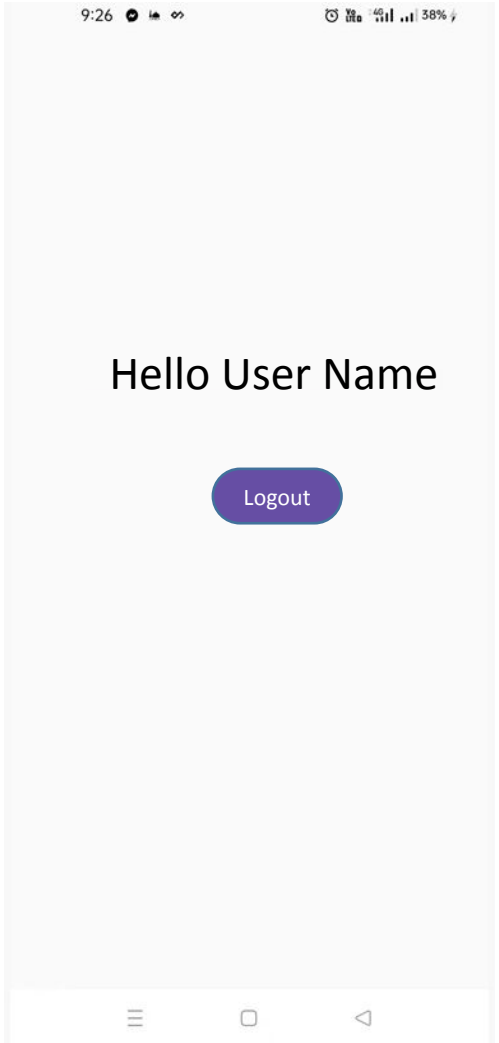
Adaptive Layouts in Compose

- https://developer.android.com/develop/ui/compose/layouts/adaptive
- https://codelabs.developers.google.com/jetpack-compose-adaptability#0

Lab


9:25
37%
UserName
Password
Confirm Password
Sign Up


9:26
38%
UserName
Password
Login


9:26
38%
Hello User Name
Logout

Thank You

For Your Next Steps

https://developer.android.com/

https://www.youtube.com/@AndroidDevelopers

https://medium.com/androiddevelopers

https://www.linkedin.com/showcase/androiddev/

https://www.kodeco.com/

Thank you