Android Jetpack Compose

### What is Jetpack Compose?

Jetpack Compose is a Declarative UI framework, meaning you construct the UI by describing what to show and not how.

Instead of using XML views, you call functions that transform the data into UI elements using Kotlin.

### What are Composable Functions?

These are functions that are annotated with the @Composable, which indicates the compiler that this function is going to convert data into UI.

With this, you build small and reusable UI elements.

These functions are immutable (you can’t hold a reference to them and later update its value).

### What are the 3 Basic Layouts in Jetpack?

* **Column**: Used to arrange its children in vertical sequence.
* **Row**: Used to arrange its children in a horizontal sequence.
* **Box**: Stacks its children on top of each other. Used to create overlays or combine elements in a single space.

### What are Modifiers?

Modifiers are objects that allow you to modify the appearance, layout and behavior of composable functions.

With modifiers you can change things like background color, padding, border, the size and position of the composable within its parent layout, and handle interactions like clicks, gestures and focus changes.

### What is State?

State is any value that can change over time and needs to be reflected in the UI. State is updated in response to events.

1. **MutableState**

Use **mutableStateOf** to create a state variable. This returns a MutableState object that holds a value and notifies Compose when the value changes.

1. **Remembering State**

Use **remember** to keep state across recompositions. It ensures that the state is retained as long as the composable is in the composition.

1. **State in ViewModels**

Use ViewModel to manage state that needs to survive configuration changes like screen rotations. This helps in maintaining a consistent state across the app lifecycle.

### What is Recomposition?

Recomposition is the process of updating the UI when the state of the app changes.

* When the state that a composable function depends on changes, the function is re-invoked to update the UI with the new state.
* It is efficient because it only re-renders the parts of the UI that need to be updated, instead of the entire screen.
* With **remember** and **mutableStateOf** you can control when and how recomposition occurs.

### What is State Hoisting?

It is the process of moving the state to a higher level in the composable hierarchy to make it accessible to multiple composables. This promotes better state management and reusability.

In this pattern, you move the state to a composable’s caller to make the first composable stateless. The general pattern replaces the state variable with two parameters:

* **value: T** – The current value to display.
* **onValueChange: (T) -> Unit** – an event that requests the value to change with a new value T.

Example:

@Composable

fun Counter() {

var count by remember { mutableStateOf(0) }

CounterDisplay(count) { count++ }

}

@Composable

fun CounterDisplay(count: Int, onIncrement: () -> Unit) {

Button(onClick = onIncrement) {

Text("Count: $count")

}

}

### What are the rules for Hoisting the State?

The main rules for hoisting state are:

1. **Hoist State to the Lowest common ancestor.**

Hoist state to the lowest common ancestor of all composables that need to read or write that state

1. **Hoist State to the Highest Level it may be changed.**

Hoist state to at least the highest level where it may be changed. This ensures that the state is accessible to all composables that need to modify it

1. **Expose Immutable State and Events**

From the state owner, expose immutable state and events to modify the state. This helps in maintaining a clear and predictable state flow.

1. **Use ViewModel for Business Logic**

When state involves business logic or needs to survive configuration changes, hoist it to a ViewModel

### How can you Persist the State?

There are several ways to persist the State:

1. **remember**

With remember you can store state in memory for the lifetime of the composable, but it doesn’t survive configuration changes.

1. **rememberSaveable**

Similar to remember but it saves the state to a Bundle and restores it after configuration changes or process death.

1. **ViewModel**

Use ViewModel to manage state that needs to survive configuration changes and be shared across multiple composables. This is lifecycle-aware state.

1. **SavedStateHandle**

Use SavedStateHandle within a ViewModel to persist state across process death.

1. **DataStore and SharedPreferences**

Uset DataStore or SharedPreferences for persisting simple key-value pairs across app launches.

1. **Room Database**

Use Room for more complex data persistence needs, such as storing and retrieving structured data.

### What is a State Holder?

A State holder is an object that holds and manages the state for your UI. t is responsible for storing the data that your composables need to display and react to. State holders ensure that the state is preserved across recompositions and configuration changes, such as screen rotations.

Common State holders are the ViewModels, the Remember function and the RememberSaveable function.

### What are the main aspects about Theming and Styling?

1. **Material Design**

Jetpack Compose has built-in support for Material Design, making it easy to apply Material theming principles. Use MaterialTheme to define colors, typography and shapes for the app.

1. **Custom Themes**

You can create custom themes and define custom colors, typography and shapes. You can find everything related to the current theme in the files inside the ui/theme folder.

1. **Dynamic Theming**

There is support for dynamic theming providing light and dark themes. You can use the isSystemInDarkTheme to switch themes based on the system settings.

1. **Consistency**

Ensure consistency across your app by using the theme's colors, typography, and shapes throughout your composables. For example: MaterialTheme.typography.h1…

### How to use Material 3 to customize the colors of the app?

Material 3 introduces a new color system that revolves around five key colors:

1. **Primary:** used for key components, like the app bar, buttons and active elements. The main brand color.
2. **Secondary:** For less prominent components, like floating action buttons, selection controls and accent elements.
3. **Tertiary:** for contrasting elements and to highlight specific parts of the UI.
4. **Error:** used to indicate errors and critical states.
5. **Neutral:** used for surfaces, backgrounds and other elements that require a neutral tone.

You need to create a Color.kt file to define your color scheme.

Define the color scheme using the colors you defined.

User your color scheme in your MaterialTheme, wrapping the main composable in the MainActivity.kt with the function AppTheme().

### What is a Scaffold?

A Scaffold is a composable that provides a basic layout structure following Material Design guidelines. It helps you organize your UI by offering slots for common elements like the top app bar, bottom navigation bar, floating action button (FAB), and drawer.

**Structure of Scaffold**

* **Top App Bar**: A bar at the top of the screen, typically used for the app's title and navigation actions.
* **Bottom Bar**: A bar at the bottom of the screen, often used for navigation or actions.
* **Floating Action Button (FAB)**: A button that floats above the content, usually for primary actions.
* **Drawer**: A side panel that can be swiped in from the edge of the screen, often used for navigation.
* **Content**: The main body of the screen where your primary UI components are placed.

Example:

@Composable

fun MyScaffold() {

Scaffold(

topBar = {

TopAppBar(title = { Text("Top App Bar") })

},

bottomBar = {

BottomAppBar { Text("Bottom App Bar") }

},

floatingActionButton = {

FloatingActionButton(onClick = { /\* Handle click \*/ }) {

Icon(Icons.Default.Add, contentDescription = "Add")

}

},

drawerContent = {

Text("Drawer Content")

},

content = { paddingValues ->

Column(modifier = Modifier.padding(paddingValues)) {

Text("Main Content")

}

}

)

}

**When to Use Scaffold**

* When you need a consistent layout structure across different screens in your app.
* Ideal for complex UIs that require multiple common elements like app bars, FABs, and drawers.
* Simplifies the process of building a structured layout.

**Benefits**

* Ensures a standardized layout across your app, making it easier to maintain and extend.
* Provides built-in support for common UI patterns, reducing the need for custom implementations.
* Allows for flexible customization of each component.

### What is a Surface?

A Surface is a composable that provides a container for your content, applying Material Design styles such shapes and colors. Are used to create a visual hierarchy and structure within the app.

Some important aspects are:

* You can wrap your content inside a Surface to apply background color, elevation and shape.
* You can use the elevation parameter to add shadow and depth, making the surface appear raised.
* You can use the shape parameter to define the shape of the surface, like RoundedCornerShape.
* You can use dynamic colors from Material 3 to theme the Surface based on user preferences.
* Surfaces can respond to user interactions, like clicks or gestures.

### How to use Scrolling?

There are several methods to implement scrolling depending on the needs:

* **Vertical and Horizontal Scrolling**

Enables vertical or horizontal scrolling for a composable. You can use them when the list has a limited number of elements.

@Composable

fun VerticalScrollExample() {

Column(

modifier = Modifier

.verticalScroll(rememberScrollState())

.fillMaxSize()

) {

repeat(20) {

Text("Item $it", modifier = Modifier.padding(20.dp))

}

}

}

* **LazyColumn and LazyRow**

Efficiently displays a vertical or horizontal scrolling list of items.

@Composable

fun LazyColumnExample() {

LazyColumn {

items(20) { index ->

Text("Item $index", modifier = Modifier.padding(20.dp))

}

}

}

* **Scrollable Modifier**

Provides manual control over scrolling behavior, useful for custom scroll interactions.

@Composable

fun ScrollableExample() {

var offset by remember { mutableStateOf(0f) }

Box(

modifier = Modifier

.size(150.dp)

.scrollable(

orientation = Orientation.Vertical,

state = rememberScrollableState { delta ->

offset += delta

delta

}

)

) {

Text("Offset: $offset", modifier = Modifier.align(Alignment.Center))

}

}

* **Remembering Scroll State**

Retain the scroll position across recompositions.

@Composable

fun RememberScrollStateExample() {

val scrollState = rememberScrollState()

Column(

modifier = Modifier.verticalScroll(scrollState)

) {

repeat(20) {

Text("Item $it", modifier = Modifier.padding(20.dp))

}

}

}

### What options do you have to apply Animations?

There are several options to apply animations in Jetpack Compose:

* **animate\*AsState**: Animates a single value, such a color:

val color by animateColorAsState(if (isSelected) Color.Red else Color.Gray)

* **updateTransitions**: you can create complex animations by defining animated properties and transitioning between states.
* You can animate the change of the content withing a composable using AnimatedContent.
* **Animate\*AsState:** you can also combine gesture detection with animations for interactive effects.
* You can create animations that repeat indefinitely using infiniteTransition.

### What kind of Lazy Lists there are in Jetpack Compose?

There are different kind of lists and columns, like:

1. **LazyColumn**

*Description*: It is used to display items in a vertical scrolling list.

*When* to use: Ideal for long lists of vertically arranged items, such as a list of messages, contacts or articles.

*Benefits*:

* + Only renders items that are visible on the screen, improving performance.
  + Easily customize item layouts and add dividers or headers.
  + Provides smooth scrolling for large datasets

LazyColumn {

items(itemsList) { item ->

Text(item)

}

}

1. **LazyRow**

*Description:* Displays items in a horizontal scrolling list.

*When to use:* Perfect for horizontally arranged items, such as image carousels, category lists or horizontal menus.

*Benefits:*

* + Similar to LazyColumn, it only renders items that can be shown in the screen.
  + Easily customize item layouts and add spacing between items.
  + Provides smooth horizontal scrolling

LazyRow {

items(itemsList) { item ->

Text(item)

}

}

1. **LazyVerticalGrid**

*Description*: Displays items in a grid format with vertical scrolling.

*When to Use:* Suitable for displaying items in a grid, such as photo galleries, product catalogs, or dashboards.

*Benefits*:

* + Only renders visible items, improving performance.
  + Supports different grid configurations, such as fixed or adaptive cell sizes.
  + Easily customize item layouts and add spacing between items.

LazyVerticalGrid(

cells = GridCells.Fixed(2)

) {

items(itemsList) { item ->

Text(item)

}

}

1. **LazyHorizontalGrid**

*Description:* Displays items in a grid format with horizontal scrolling.

*When to Use:* Useful for horizontally scrolling grids, such as horizontal image galleries or product showcases.

*Benefits*:

* + Similar to LazyVerticalGrid, it only renders visible items.
  + Supports different grid configurations.
  + Easily customize item layouts and add spacing between items.

LazyHorizontalGrid(

cells = GridCells.Fixed(2)

) {

items(itemsList) { item ->

Text(item)

}

}

1. **LazyListScope**

*Description*: Allows for more complex and custom layouts within LazyColumn and LazyRow.

*When to Use*: Use when you need to mix different types of items, add headers, footers, or sections to your lists.

*Benefits*:

* + Combine different item types and layouts within a single list.
  + Easily add headers, footers, and section dividers.
  + Maintains the efficiency of lazy loading.

LazyColumn {

item {

Text("Header")

}

items(itemsList) { item ->

Text(item)

}

item {

Text("Footer")

}

}

### What is Navigation Rail?

Navigation Rail is a vertical navigation component that provides a way to navigate between different sections of your app. It is part of the Material Design.

The Navigation Rail is positioned on the left side of the screen, providing vertical navigation.

It typically contains icons and optional labels for each navigation item.

Optionally, it can include a FAB for primary actions.

**When to use it?**

* Ideal for devices with larger screens, such as tablets and foldable devices, where bottom navigation might be less accessible
* Useful for scenarios involving expandable displays, providing easy access to navigation icons situated in the sidebar
* Recommended for medium and expanded window size classes, as defined by Material Design guidelines

### How can you adapt to the Window Sizes?

To adapt the application to different window sizes or in the landscape mode, you need to use the calculateWindowSizeClass(). Based on the window size class, provide a different layout for Compact, Medium and Expanded screens. When the app is in portrait mode it is Compact width, when it is in landscape mode it is Expanded width.

Example:

@Composable

fun MyApp() {

val windowSizeClass = calculateWindowSizeClass(LocalContext.current)

when (windowSizeClass.widthSizeClass) {

WindowWidthSizeClass.Compact -> {

// Layout for small screens (e.g., phones)

CompactLayout()

}

WindowWidthSizeClass.Medium -> {

// Layout for medium screens (e.g., small tablets)

MediumLayout()

}

WindowWidthSizeClass.Expanded -> {

// Layout for large screens (e.g., large tablets, foldables)

ExpandedLayout()

}

}

}

### How can you use the Observable Pattern in Jetpack?

In Android Jetpack there are several ways to implement the Observable pattern:

* **LiveData**

LiveData is a lifecycle-aware observable data holder class that allows UI components to observe changes in data and update themselves automatically.

It ensures that updates are only sent to active observers, preventing memory leaks.

* **Flow**

Flow is part of the Coroutines and provides a reactive streams API. It can be used to handle asynchronous data streams and if lifecycle-aware when used with Jetpack components.

Offers more flexibility and power compared to LiveData, especially for handling complex data streams and transformations.

* **State in Jetpack Compose**

State is used to manage and observe state changes with composables. The remember and mutableStateOf APIs are commonly used to create observable state objects.

This approach is particularly useful in declarative UI frameworks like Jetpack Compose, where UI updates are driven by state change

* **ViewModel**

VM is designed to store and manage UI-related data in a lifecycle way. Works similarly to LiveData and Flow.

The VM helps separating the UI logic from business logic, making the code more modular and testable.

* **Data Binding**

A library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programatically.

It supports observable data objects.

### What is the ViewModel?

A **ViewModel** is a class in Android Jetpack that is designed to store and manage UI-related data in a way that is aware of the lifecycle of activities and fragments. This helps to keep your data safe and available even when the screen is rotated or the app is temporarily closed.

**How does it work?**

* The ViewModel is aware of the lifecycle of the UI components. This means it can survive configuration changes, such as screen rotations, without losing its data.
* It holds and manages UI-related data. For example, if your app displays a list of items, the ViewModel can store this list. When the UI needs to display the list, it can get the data from the ViewModel.
* You separate the UI logic from the business logic, making the code cleaner, easier to manage and more testable.

**Example**

Create a ViewModel class

class CounterViewModel : ViewModel() {

val counter = MutableLiveData(0)

fun increment() {

counter.value = (counter.value ?: 0) + 1

}

}

Use the ViewModel in a composable function

@Composable

fun CounterScreen(viewModel: CounterViewModel = viewModel()) {

val counter by viewModel.counter.observeAsState(0)

Column(

modifier = Modifier.fillMaxSize()

) {

Text(text = "Counter: $counter")

Button(onClick = { viewModel.increment() }) {

Text("Increment")

}

}

}

### What are the Different Compose Phases?

Compose has a unique way of handling UI updates through a series of phases.

* 1. **Composition**

What to show.

The phase where the UI tree is built.

When a composable function is called, it generates a tree of UI elements. This tree is then used to determine what should be displayed on the screen.

* 1. **Recomposition**

This phase occurs when the state of the UI changes and the UI needs to be updated.

When a state change is detected, Compose will re-execute the composable functions that depend on that state. Only the parts of the UI that need to be updated will be recomposed

* 1. **Layout**

Where to place it.

This phase determines the size and position of each UI element.

* 1. **Drawing**

The last phase where the UI elements are rendered on the screen.

Compose takes the layout informaion and draws the UI elements on the canvas.

### What is the Unidirectional Data Flow

Unidirectional data flow (UDF) in Android Jetpack Compose is a design pattern where data flows in a single direction: from a source of truth (like a ViewModel) to the UI, and user interactions flow back to the source of truth. This ensures a clear and predictable flow of data, making the app easier to understand and debug.

In simple terms, it means:

1. **State** -> **UI**: The state drives the UI.
2. **UI** -> **Events**: User interactions generate events.
3. **Events** -> **State**: Events update the state.

This cycle helps maintain a consistent and manageable UI state.

### What is a Coroutine?

A Coroutine is a lightweight thread that allows you to write asynchronous code in a sequential manner, simplifying tasks like network calls, database operations or any long-running tasks that could block the main thread.

The coroutines can be suspended and resumed without blocking the main thread.

Coroutines are lifecycle-aware, ensuring that tasks are cancelled when the composable is not in use.

Example:

1. Start launching a coroutine using the launch function withing a coroutine scope:

CoroutineScope(Dispatchers.IO).launch {

// Perform a long-running task

}

1. Coroutines can call suspending functions

suspend fun fetchData(): String {

delay(1000) // Simulate a network call

return "Data fetched"

}

1. In Jetpack Compose, you can use coroutines to handle asynchronous tasks within composables

@Composable

fun MyComposable() {

val scope = rememberCoroutineScope()

var data by remember { mutableStateOf("Loading...") }

LaunchedEffect(Unit) {

data = fetchData()

}

Text(text = data)

}

### How does the Navigation work in Compose?

To navigate through different screens (or composables) you need some components:

1. **NavController**

The **NavController** is the central component that manages app navigation. It keeps track of the back stack of composable destinations and orchestrates navigation actions. You can obtain a NavController instance using the *rememberNavController* function:

@Composable  
fun RallyApp() {

    RallyTheme {

        var currentScreen: RallyDestination by remember { mutableStateOf(Overview) }

        val navController = rememberNavController()

        Scaffold(

            // ...

        ) {

            // ...

       }

}

1. **NavHost**

The **NavHost** is a container composable that hosts the navigation graph. It connects the NavController with the composable destinations defined in your navigation graph. You set up a NavHost like this

NavHost(

navController = navController,

startDestination = "home"

) {

composable("home") { HomeScreen(navController) }

composable("details/{itemId}") { backStackEntry ->

val itemId = backStackEntry.arguments?.getString("itemId")

DetailsScreen(navController, itemId)

}

}

1. **Routes**

**Routes** are the unique identifiers for each composable destination in your navigation graph. They are used to define the paths and parameters for navigation. For example, in the above NavHost setup, "home" and "details/{itemId}" are routes. You navigate between routes using the NavController:

navController.navigate("details/$itemId")

### How do you create a simple UI Test?

* First, add the dependencies in the build.gradle
* Create a Test Class in your androidTest directory.
* Write the test, using the ComposeTestRule to set the content of your composable and perform UI interactions:
  + **Set Content**: Use composeTestRule.setContent to set the composable content you want to test.
  + **Find Nodes:** Use onNodeWithText to find UI elements by their text.
  + **Perform Actions:** Use performClick to simulate user interactions.
  + **Assertions:** Use assertExists to verify the presence of UI elements.

Example:

@Test

fun testButtonClick() {

composeTestRule.setContent {

MyApp()

}

// Verify initial state

composeTestRule.onNodeWithText("Home Screen").assertExists()

// Perform click action

composeTestRule.onNodeWithText("Go to Details").performClick()

// Verify new state

composeTestRule.onNodeWithText("Details Screen for item: 123").assertExists()

}