Android Jetpack Compose Notes

# Compose Essentials

Jetpack Compose is Declarative UI Framework. You declare what you UI should contain and compose creates the elements UI using Kotlin.

You construct the UI by describing what and not how.

You don’t need to use XML Views, the UI will be described in code using Kotlin.

In Compose, UI elements are functions, known as composables, and not objects. That means you can’t find references to them and mutate them. Instead, UI elements are controlled by the state or arguments you pass.

We don’t tell Compose how it should render states. To do that, we use Event Handlers, which decides if the UI element’s state should be changed. If the UI state changes, the functions or UI elements that depend on that state is re-executed, this is called recomposition.

## Composable Functions

A composable function is a function that has a @Composable annotation. This annotation indicates the compiler that this function is intended to convert data into UI.

The composable functions allow you have reusable components.

Composables are immutable, that means you can’t hold a reference to them and later update its value.

Recomposition happens when a composable is re-invoked with different functions parameters but it can also happen when internal state in the function changes.

MutableState, remember and rememberSaveable can be used to store a component state and have Compose automatically track and recompose changes.

Composable functions can execute in any order.

Composable functions can also run in parallel.

Recomposition skips as much as possible.

Composable functions might run frequently.

With Compose, an Activity remains the entry point to an Android app. In our project, MainActivity is launched when the user opens the app (as it's specified in the AndroidManifest.xml file). You use setContent to define your layout, but instead of using an XML file as you'd do in the traditional View system, you call Composable functions within it.

To use the Android Studio preview, you just have to mark any parameterless Composable function or functions with default parameters with the @Preview annotation and build your project. Example:

@Preview(showBackground = true, name = "Text preview")  
@Composable  
fun GreetingPreview() {  
    BasicsCodelabTheme {  
        Greeting(name = "Android")  
    }  
}

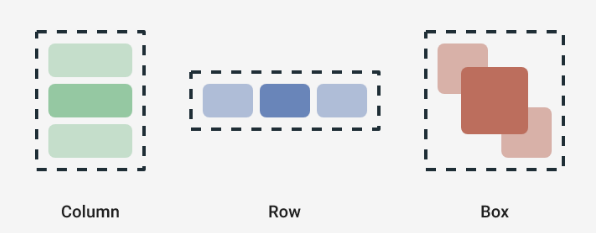
Surface and MaterialTheme are concepts related to [Material Design](https://m3.material.io/" \t "_blank), which is a design system created by Google to help you create user interfaces and experiences.

@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Surface(color = MaterialTheme.colorScheme.primary) {  
        Text(  
            text = "Hello $name!",  
            modifier = modifier  
        )  
    }  
}

As a best practice, your function should include a Modifier parameter that is assigned an empty Modifier by default.

## Columns and Rows

The 3 basic standard layout elements in Compose are Column, Row and Box



Examples:

@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Surface(  
        color = MaterialTheme.colorScheme.primary,  
        modifier = modifier.padding(vertical = 4.dp, horizontal = 8.dp)  
    ) {  
        Column(modifier = Modifier.fillMaxWidth().padding(24.dp)) {  
            Text(text = "Hello ")  
            Text(text = name)  
        }  
    }  
}

**Examples for a Row**

In this code, you are using a row and placing an ElevatedButton at the end of the row by using a weight to the first element (the Column). The weight modifier makes the element fill all available space, making it *flexible*, effectively pushing away the other elements that don't have a weight, which are called *inflexible*. It also makes the fillMaxWidth modifier redundant.

import androidx.compose.foundation.layout.Row  
import androidx.compose.material3.ElevatedButton  
// ...  
  
@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Surface(  
        color = MaterialTheme.colorScheme.primary,  
        modifier = modifier.padding(vertical = 4.dp, horizontal = 8.dp)  
    ) {  
        Row(modifier = Modifier.padding(24.dp)) {  
            Column(modifier = Modifier.weight(1f)) {  
                Text(text = "Hello ")  
                Text(text = name)  
            }  
            ElevatedButton(  
                onClick = { /\* TODO \*/ }  
            ) {  
                Text("Show more")  
            }  
        }  
    }  
}

### Recompose

Remember that the use of **remember** is to guard a state after recomposition.

You can think of internal state as a private variable in a class.

The composable function will automatically be "subscribed" to the state. If the state changes, composables that read these fields will be recomposed to display the updates

@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    val expanded = remember { mutableStateOf(false) }  
    val extraPadding = if (expanded.value) 48.dp else 0.dp  
    Surface(  
        color = MaterialTheme.colorScheme.primary,  
        modifier = modifier.padding(vertical = 4.dp, horizontal = 8.dp)  
    ) {  
        Row(modifier = Modifier.padding(24.dp)) {  
            Column(  
                modifier = Modifier  
                    .weight(1f)  
                    .padding(bottom = extraPadding)  
            ) {  
                Text(text = "Hello ")  
                Text(text = name)  
            }  
            ElevatedButton(  
                onClick = { expanded.value = !expanded.value }  
            ) {  
                Text(if (expanded.value) "Show less" else "Show more")  
            }  
        }  
    }  
}

### State Hoisting

In Composable functions, state that is read or modified by multiple functions should live in a common ancestor—this process is called **state hoisting**. To *hoist* means to *lift* or *elevate*.

Making state hoistable avoids duplicating state and introducing bugs, helps reuse composables, and makes composables substantially easier to test.

In Compose **you don't hide UI elements**. Instead, you simply don't add them to the composition, so they're not added to the UI tree that Compose generates. You do this with simple conditional Kotlin logic.

Example:

@Composable  
fun MyApp(modifier: Modifier = Modifier) {  
  
    var shouldShowOnboarding by remember { mutableStateOf(true) }  
  
    Surface(modifier) {  
        if (shouldShowOnboarding) {  
            OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })  
        } else {  
            Greetings()  
        }  
    }  
}  
  
@Composable  
fun OnboardingScreen(  
    onContinueClicked: () -> Unit,  
    modifier: Modifier = Modifier  
) {  
  
  
    Column(  
        modifier = modifier.fillMaxSize(),  
        verticalArrangement = Arrangement.Center,  
        horizontalAlignment = Alignment.CenterHorizontally  
    ) {  
        Text("Welcome to the Basics Codelab!")  
        Button(  
            modifier = Modifier  
                .padding(vertical = 24.dp),  
            onClick = onContinueClicked  
        ) {  
            Text("Continue")  
        }  
    }  
  
}

In the MyApp composable, we define the shouldShowOnboarding variable as a mutableStateOf and using the by remember so this state can be changed later. Depending of the value of this variable, we show the composable OnboardingScreen or the Greetings.

We share the shouldShowOnboarding with the OnboardingScreen composable, but instead of passing its value directly, we use a callback. Callbacks are functions that are passed as arguments to other functions and get executed when the event occur.

By passing a function and not a state to OnboardingScreen we are making this composable more reusable and protecting the state from being mutated by other composables. In general, it keeps things simple.

### Lazy Lists

When you define a list like this one:

names: List<String> = List(1000) { "$it" }

The it represents the index.

If for every item you create a Composable, like the Greetings to show the value, you can use a LazyColumn instead of a Column to redners only the visible items on screen, allowing performance gains when rendering a big list.

**Note**: LazyColumn and LazyRow are equivalent to RecyclerView in Android Views.

import androidx.compose.foundation.lazy.LazyColumn  
import androidx.compose.foundation.lazy.items  
// ...  
  
@Composable  
private fun Greetings(  
    modifier: Modifier = Modifier,  
    names: List<String> = List(1000) { "$it" }   
) {  
    LazyColumn(modifier = modifier.padding(vertical = 4.dp)) {  
        items(items = names) { name ->  
            Greeting(name = name)  
        }  
    }  
}

**Note:**LazyColumn doesn't recycle its children like RecyclerView. It emits new Composables as you scroll through it and is still performant, as emitting Composables is relatively cheap compared to instantiating Android Views.

### Persisting State

If your app allows screen rotation, you may notice your state is lost. The remember function works **only as long as the composable is kept in the Composition**. When you rotate, the whole activity is restarted so all state is lost. This also happens with any configuration change and on process death.

Instead of using remember you can use rememberSaveable. This will save each state surviving configuration changes (such as rotations) and process death.

    import androidx.compose.runtime.saveable.rememberSaveable  
    // ...  
  
    var shouldShowOnboarding by rememberSaveable { mutableStateOf(true) }

You can also use rememberSaveable to remember a state after configuration changes.

   var expanded by rememberSaveable { mutableStateOf(false) }

### Animations

In Compose, there are multiple ways to animate your UI: from high-level APIs for simple animations to low-level methods for full control and complex transitions. You can read about them in the [documentation](https://developer.android.com/jetpack/compose/animation?authuser=1).

One example of a low level animation is the animateDpAsState composable.

animateDpAsState takes an optional animationSpec parameter that lets you customize the animation.

import androidx.compose.animation.core.Spring  
import androidx.compose.animation.core.spring  
  
  
@Composable  
private fun Greeting(name: String, modifier: Modifier = Modifier) {  
  
    var expanded by rememberSaveable { mutableStateOf(false) }  
  
    val extraPadding by animateDpAsState(  
        if (expanded) 48.dp else 0.dp,  
        animationSpec = spring(  
            dampingRatio = Spring.DampingRatioMediumBouncy,  
            stiffness = Spring.StiffnessLow  
        )  
    )  
  
    Surface(  
    // ...  
            Column(modifier = Modifier  
                .weight(1f)  
                .padding(bottom = extraPadding.coerceAtLeast(0.dp))  
  
    // ...  
  
    )  
}

### Styling and Theming

In the ui/theme/Theme.kt file, you see the definition of the project’s Theme, which uses MaterialTheme, a composable function that reflects the styling principles from the [Material design specification](https://m3.material.io/). That styling information cascades down to the components that are inside its content, which may read the information to style themselves.

From any descendant composable you can retrieve three properties of MaterialTheme: colorScheme, typography and shapes.

Example:

            Column(modifier = Modifier  
                .weight(1f)  
                .padding(bottom = extraPadding.coerceAtLeast(0.dp))  
            ) {  
                Text(text = "Hello, ")  
                Text(text = name, style = MaterialTheme.typography.headlineMedium)  
            }

The Text composable in the example above sets a new TextStyle. You can create your own TextStyle, or you can retrieve a theme-defined style by using MaterialTheme.typography, which is preferred. This construct gives you access to the Material-defined text styles, such as displayLarge, headlineMedium, titleSmall, bodyLarge, labelMedium etc

Sometimes you need to deviate slightly from the selection of colors and font styles. In those situations it's better to base your color or style on an existing one.

For this, you can modify a predefined style by using the copy function. Make the number extra bold

import androidx.compose.ui.text.font.FontWeight  
// ...  
Text(  
    text = name,  
    style = MaterialTheme.typography.headlineMedium.copy(  
        fontWeight = FontWeight.ExtraBold  
    )  
)

This way if you need to change the font family or any other attribute of headlineMedium, you don't have to worry about the small deviations.

Example to show a preview with the Dark Mode active:

import android.content.res.Configuration.UI\_MODE\_NIGHT\_YES  
  
  
@Preview(  
    showBackground = true,  
    widthDp = 320,  
    uiMode = UI\_MODE\_NIGHT\_YES,  
    name = "GreetingPreviewDark"  
)  
@Preview(showBackground = true, widthDp = 320)  
@Composable  
fun GreetingPreview() {  
    BasicsCodelabTheme {  
        Greetings()  
    }  
}

You can find everything related to the current theme in the files inside the ui/theme folder. For example, the default colors that we have been using so far are defined in Color.kt.

In Theme.kt, define for example, the palette for dark colors:

private val DarkColorScheme = darkColorScheme(  
    surface = Blue,  
    onSurface = Navy,  
    primary = Navy,  
    onPrimary = Chartreuse  
)

These colors can be defined in the Color.kt file:

val Navy = Color(0xFF073042)  
val Blue = Color(0xFF4285F4)  
val LightBlue = Color(0xFFD7EFFE)  
val Chartreuse = Color(0xFFEFF7CF)

Example:

private val DarkColorScheme = darkColorScheme(  
    surface = Blue,  
    onSurface = Navy,  
    primary = Navy,  
    onPrimary = Chartreuse  
)  
  
private val LightColorScheme = lightColorScheme(  
    surface = Blue,  
    onSurface = Color.White,  
    primary = LightBlue,  
    onPrimary = Navy  
)  
  
@Composable  
fun BasicsCodelabTheme(  
    darkTheme: Boolean = isSystemInDarkTheme(),  
    // Dynamic color is available on Android 12+  
    dynamicColor: Boolean = true,  
    content: @Composable () -> Unit  
) {  
    val colorScheme = when {  
        dynamicColor && Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.S -> {  
            val context = LocalContext.current  
            if (darkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(context)  
        }  
        darkTheme -> DarkColorScheme  
        else -> LightColorScheme  
    }  
    val view = LocalView.current  
    if (!view.isInEditMode) {  
        SideEffect {  
            (view.context as Activity).window.statusBarColor = colorScheme.primary.toArgb()  
            ViewCompat.getWindowInsetsController(view)?.isAppearanceLightStatusBars = darkTheme  
        }  
    }  
  
    MaterialTheme(  
        colorScheme = colorScheme,  
        typography = Typography,  
        content = content  
    )  
}

## The Compose UI toolkit

Jetpack Compose comes with a lot of UI elements out of the box.

Material Design is an adaptable system of guidelines, components and tools that supports the best practices of user interface design. Jetpack compose supports Material Design 2 and 3

With Material Design your app can be themed to match your brand by providing custom colors, typography and shapes.

You can define a Theme composable with all the customizations you want and this Theme function should be the outermost function we invoke. It is the first composable we invoke in the setContent

Scaffold is a fundamental material design component. It is a basic layout for arranging material components in common patterns, such as the screen with a small top app bar and a floating action button.

Another component is the Surface, which is where the content sits on.

There are 3 types of layouts as seen before: Rows (to order elements horizontally), Columns (to arrange elements vertically) and Boxes to show elements on top of anothers.

### Lazy Rows

A LazyRow allows you to implement a scrollable row composable. It only renders elements that are shown on the screen instead of all elements at the same time, which helps keep your app performance.

For the spacings between the elements we can make use of arrangements. There are several options to arrange the elements in a Row and in a column. In addition to the these arrangements, you can also use the Arrangement.spacedBy() to add a fixed space between each child composable.

import androidx.compose.foundation.layout.PaddingValues  
  
@Composable  
fun AlignYourBodyRow(  
   modifier: Modifier = Modifier  
) {  
   LazyRow(  
       horizontalArrangement = Arrangement.spacedBy(8.dp),  
       contentPadding = PaddingValues(horizontal = 16.dp),  
       modifier = modifier  
   ) {  
       items(alignYourBodyData) { item ->  
           AlignYourBodyElement(item.drawable, item.text)  
       }  
   }  
}

To maintain the same padding, but still scroll your content within the bounds of your parent list without clipping it, all lists provide a parameter to the LazyRow called contentPadding.

### Lazy Grids

The LazyHorizontalGrid provides a nice mapping from items to grid elements. In the following example you have a LazyHorizontalGrid with two fixed rows and some other measures like the height of the elements and the horizontal and vertical arrangement:

@Composable  
fun FavoriteCollectionsGrid(  
   modifier: Modifier = Modifier  
) {  
   LazyHorizontalGrid(  
       rows = GridCells.Fixed(2),  
       contentPadding = PaddingValues(horizontal = 16.dp),  
       horizontalArrangement = Arrangement.spacedBy(16.dp),  
       verticalArrangement = Arrangement.spacedBy(16.dp),  
       modifier = modifier.height(168.dp)  
   ) {  
       items(favoriteCollectionsData) { item ->  
           FavoriteCollectionCard(item.drawable, item.text, Modifier.height(80.dp))  
       }  
   }  
}

### Slot APIs

**Slot-based layouts** leave an empty space in the UI for the developer to fill as they wish. You can use them to create more flexible layouts.

In the following example:

@Composable  
fun HomeSection(  
   @StringRes title: Int,  
   modifier: Modifier = Modifier,  
   content: @Composable () -> Unit  
) {  
   Column(modifier) {  
       Text(stringResource(title))  
       content()  
   }  
}

You can use the content parameter for the composable's slot. This way, when you use the HomeSection composable, you can use a trailing lambda to fill the content slot. When a composable provides multiple slots to fill in, you can give them meaningful names that represent their function in the bigger composable container.

### Scrolling

You don’t always need a Lazy layout to automatically scroll a list of items. When a list has only a limited number of elements, you can instead choose to use a simple Column or Row and **add the scroll behavior manually**. To do so, you use the [verticalScroll](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary?authuser=1" \l "(androidx.compose.ui.Modifier).verticalScroll(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean)" \t "_blank) or [horizontalScroll](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary?authuser=1" \l "(androidx.compose.ui.Modifier).horizontalScroll(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean)" \t "_blank) modifiers. These require a [ScrollState](https://developer.android.com/reference/kotlin/androidx/compose/foundation/ScrollState?authuser=1" \t "_blank), which contains the current state of the scroll, used to modify the scroll state from outside. In this case, you're not looking to modify the scroll state, so you simply create a persistent ScrollState instance using [rememberScrollState](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary?authuser=1" \l "rememberScrollState(kotlin.Int)" \t "_blank).

Example:

import androidx.compose.foundation.layout.Spacer  
import androidx.compose.foundation.rememberScrollState  
import androidx.compose.foundation.verticalScroll  
  
@Composable  
fun HomeScreen(modifier: Modifier = Modifier) {  
   Column(  
       modifier  
           .verticalScroll(rememberScrollState())  
   ) {  
       Spacer(Modifier.height(16.dp))  
       SearchBar(Modifier.padding(horizontal = 16.dp))  
       HomeSection(title = R.string.align\_your\_body) {  
           AlignYourBodyRow()  
       }  
       HomeSection(title = R.string.favorite\_collections) {  
           FavoriteCollectionsGrid()  
       }  
       Spacer(Modifier.height(16.dp))  
   }  
}

### Scaffold

Material's [Scaffold](https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary?authuser=1#Scaffold(androidx.compose.ui.Modifier,kotlin.Function0,kotlin.Function0,kotlin.Function0,kotlin.Function0,androidx.compose.material3.FabPosition,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.foundation.layout.WindowInsets,kotlin.Function1)) composable. Scaffold gives you a **top-level configurable composable** for apps that implement Material design. It contains slots for various Material concepts, one of which is the bottom bar.

import androidx.compose.material3.Scaffold  
  
@Composable  
fun MySootheAppPortrait() {  
   MySootheTheme {  
       Scaffold(  
           bottomBar = { SootheBottomNavigation() }  
       ) { padding ->  
           HomeScreen(Modifier.padding(padding))  
       }  
   }  
}

### Navigation Rail

For the landscape mode, you can use a NavigationRail composable which is part of the Compose Material library and has a similar implementation to the [NavigationBar](https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary?authuser=1" \l "NavigationBar(androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.unit.Dp,androidx.compose.foundation.layout.WindowInsets,kotlin.Function1)" \t "_blank) that was used to create the bottom navigation bar.

import androidx.compose.foundation.layout.fillMaxHeight  
  
@Composable  
private fun SootheNavigationRail(modifier: Modifier = Modifier) {  
   NavigationRail(  
       modifier = modifier.padding(start = 8.dp, end = 8.dp),  
       containerColor = MaterialTheme.colorScheme.background,  
   ) {  
       Column(  
           modifier = modifier.fillMaxHeight(),  
           verticalArrangement = Arrangement.Center,  
           horizontalAlignment = Alignment.CenterHorizontally  
       ) {  
           NavigationRailItem(  
               icon = {  
                   Icon(  
                       imageVector = Icons.Default.Spa,  
                       contentDescription = null  
                   )  
               },  
               label = {  
                   Text(stringResource(R.string.bottom\_navigation\_home))  
               },  
               selected = true,  
               onClick = {}  
           )  
           Spacer(modifier = Modifier.height(8.dp))  
           NavigationRailItem(  
               icon = {  
                   Icon(  
                       imageVector = Icons.Default.AccountCircle,  
                       contentDescription = null  
                   )  
               },  
               label = {  
                   Text(stringResource(R.string.bottom\_navigation\_profile))  
               },  
               selected = false,  
               onClick = {}  
           )  
       }  
   }  
}

### Window Size

To show the landscape version when you turn the device, you need to use the calculateWindowSizeClass() function to see what configuration the phone is in.

There are three window size class widths: Compact, Medium and Expanded. When the app is in portrait mode it is Compact width, when it is in landscape mode it is Expanded width.

In the following example, we define which composable to use depending on the window size class:

import androidx.compose.material3.windowsizeclass.WindowSizeClass  
import androidx.compose.material3.windowsizeclass.WindowWidthSizeClass  
@Composable  
fun MySootheApp(windowSize: WindowSizeClass) {  
   when (windowSize.widthSizeClass) {  
       WindowWidthSizeClass.Compact -> {  
           MySootheAppPortrait()  
       }  
       WindowWidthSizeClass.Expanded -> {  
           MySootheAppLandscape()  
       }  
   }  
}

When we call this function, we use this in the setContent for the MainActivity:

import androidx.compose.material3.windowsizeclass.ExperimentalMaterial3WindowSizeClassApi  
import androidx.compose.material3.windowsizeclass.calculateWindowSizeClass  
  
class MainActivity : ComponentActivity() {  
   @OptIn(ExperimentalMaterial3WindowSizeClassApi::class)  
   override fun onCreate(savedInstanceState: Bundle?) {  
       super.onCreate(savedInstanceState)  
       setContent {  
           val windowSizeClass = calculateWindowSizeClass(this)  
           MySootheApp(windowSizeClass)  
       }  
   }  
}

## State in Jetpack Compose

State is any value that changes over time. In Android apps, state is updated in response to events.

While the state of the app offers a description of what to display in the UI, events are the mechanism through which the state changes, resulting in changes to the UI.

### Re-composition

Compose apps transform data into UI by calling composable functions. We refer to ***the Composition*** as the description of the UI built by Compose when it executes composables. If a state change happens, Compose re-executes the affected composable functions with the new state, creating an updated UI—this is called ***recomposition***. Compose also looks at what data an individual composable needs, so that it only recomposes components whose data has changed and skips those that are not affected.

**The Composition:** a description of the UI built by Jetpack Compose when it executes composables.

**Initial composition:** creation of a Composition by running composables the first time.

**Recomposition:** re-running composables to update the Composition when data changes.

Compose need to know what state to track. Compose has a special state tracking system in place that schedules recompositions for any composables that read a particular state.

The State and MutableState types make state observable by Compose. Compose keeps track of each composable that reads State value properties and triggers a recomposition when its value changes. You can use the [mutableStateOf](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1" \l "mutableStateOf(kotlin.Any,androidx.compose.runtime.SnapshotMutationPolicy)" \t "_blank) function to create an observable MutableState. It receives an initial value as a parameter that is wrapped in a State object, which then makes its value observable

There are other variants of mutableStateOf like mutableIntStateOf, which are optimized for the primitive types.

### State drive UI

Compose is a declarative UI framework. Instead of removing UI components or changing their visibility when state changes, we describe how the UI *is* under specific conditions of state.

The output of the Composition is a tree-structure that describes the UI. You can inspect the app layout generated by Compose using [Android Studio's Layout inspector tool](https://developer.android.com/studio/debug/layout-inspector?authuser=1" \t "_blank).

@Composable  
fun WaterCounter(modifier: Modifier = Modifier) {  
   Column(modifier = modifier.padding(16.dp)) {  
       var count by remember { mutableStateOf(0) }  
  
       if (count > 0) {  
           // This text is present if the button has been clicked  
           // at least once; absent otherwise  
           Text("You've had $count glasses.")  
       }  
       Button(onClick = { count++ }, Modifier.padding(top = 8.dp)) {  
           Text("Add one")  
       }  
   }  
}

### Remember in Composition

[remember](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1#remember(kotlin.Function0)) stores objects in the Composition, and forgets the object if the source location where remember is called is not invoked again during a recomposition.

While remember helps you retain state across recompositions, it's **not retained across configuration changes**. For this, you must use [rememberSaveable](https://developer.android.com/reference/kotlin/androidx/compose/runtime/saveable/package-summary?authuser=1" \l "rememberSaveable(kotlin.Array,androidx.compose.runtime.saveable.Saver,kotlin.String,kotlin.Function0)" \t "_blank) instead of remember.

rememberSaveable automatically saves any value that can be saved in a [Bundle](https://developer.android.com/reference/android/os/Bundle?authuser=1). For other values, you can pass in a custom saver object.

Use rememberSaveable to restore your UI state after an Activity is recreated. Besides retaining state across recompositions, rememberSaveable also retains state across Activity recreation and system-initiated process death.

### State hoisting

Composables with internal state tend to be less reusable and harder to test. Composables that don't hold any state are called stateless composables. An easy way to create a stateless composable is by using state hoisting.

State hoisting in Compose is a pattern of moving state to a composable's caller to make a composable stateless. The general pattern for state hoisting in Jetpack Compose is to replace 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

A **stateless** composable is a composable that doesn't own any state, meaning it doesn't hold or define or modify new state.

A **stateful** composable is a composable that owns a piece of state that can change over time.

In real apps, having a 100% stateless composable can be difficult to achieve depending on the composable's responsibilities. You should design your composables in a way that they will own as little state as possible and allow the state to be hoisted, when it makes sense, by exposing it in the composable's API.

**Key Point:** When hoisting state, there are three rules to help you figure out where state should go:

1. State should be hoisted to at *least* the **lowest common parent** of all composables that use the state (read).
2. State should be hoisted to at *least* the **highest level it may be changed** (write).
3. If **two states change in response to the same events** they should be **hoisted to the same level.**

You can hoist the state higher than these rules require, but if you don't hoist the state high enough, it might be difficult or impossible to follow unidirectional data flow.

### Common patterns in Compose

The composable function rememberLazyListState creates an initial state for the list using rememberSaveable. When the Activity is recreated, the scroll state is maintained.

Lazy components, like LazyColumn or LazyRow, support this use case through hoisting events related to the list like scroll position or item layout changes.

Having a state parameter with a default value provided by a public rememberX function is a common pattern in built-in composable functions. Another example can be found in [BottomSheetScaffold](https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary?authuser=1" \l "BottomSheetScaffold(kotlin.Function1,androidx.compose.ui.Modifier,androidx.compose.material3.BottomSheetScaffoldState,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,androidx.compose.ui.graphics.Shape,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,kotlin.Function0,kotlin.Boolean,kotlin.Function0,kotlin.Function1,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,kotlin.Function1)" \t "_blank), which hoists state using [rememberBottomSheetScaffoldState](https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary?authuser=1" \l "rememberBottomSheetScaffoldState(androidx.compose.material3.SheetState,androidx.compose.material3.SnackbarHostState)" \t "_blank).

### Observable MutableList

Mutable objects like ArrayList<T> or mutableListOf won’t notify Compose that the items in the list have changed and schedule a recomposition of the UI. You need an instance f MutableList that is observable by Compose. The extension function [toMutableStateList()](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1" \l "(kotlin.collections.Collection).toMutableStateList()" \t "_blank) is the way to create an observable MutableList from an initial mutable or immutable Collection, such as List.

Alternatively, you could also use the factory method [mutableStateListOf](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1" \l "mutableStateListOf()" \t "_blank) to create the observable MutableList and then add the elements for your initial state.

The mutableStateOf function returns an object of type MutableState<T>.

The mutableStateListOf and toMutableStateList functions return an object of type [SnapshotStateList<T>](https://developer.android.com/reference/kotlin/androidx/compose/runtime/snapshots/SnapshotStateList?authuser=1" \t "_blank). In this section, the terms "observable MutableList" refer to this class.

You can use the mutableStateListOf API instead to create a list. If you just define the list and then add the tasks in a different operation it would result in duplicated items being added for **every** recomposition.

// Don't do this!

val list = remember { mutableStateListOf<WellnessTask>() }

list.addAll(getWellnessTasks())

Instead, create the list with its initial value in a single operation and then pass it to the remember function, like this:

// Do this instead. Don't need to copy

val list = remember {

mutableStateListOf<WellnessTask>().apply { addAll(getWellnessTasks()) }

}

### State in ViewModel

While the UI state describes what to show on the screen, the logic of an app describes how the app behaves and should react to state changes. There are two types of logic:

* The UI logic relates to *how to display* state changes on the screen (for example, the navigation logic or showing snackbars).
* The business logic is *what to do* with state changes (for example making a payment or storing user preferences). This logic is usually placed in the business or data layers, never in the UI layer.

[ViewModels](https://developer.android.com/topic/libraries/architecture/viewmodel?authuser=1) provide the UI state and access to the business logic located in other layers of the app. ViewModels survive configuration changes, so they have a longer lifetime than the Composition.

**Warning:** ViewModels are not part of the Composition. Therefore, you should not hold state created in composables (for example, a remembered value) because this could cause memory leaks.

viewModel() returns an existing ViewModel or creates a new one in the given scope. The ViewModel instance is retained as long as the scope is alive.

ViewModels are recommended to be used at screen-level composables, that is, close to a root composable called from an activity, fragment, or destination of a Navigation graph. ViewModels should never be passed down to other composables, instead you should pass only the data they need and functions that perform the required logic as parameters.

It is a bad practice to pass down ViewModel instances to other composables. You should pass only the data they need and functions that perform the required logic as parameters.

## Migrating to Jetpack Compose

The recommended [migration strategy](https://developer.android.com/jetpack/compose/interop/migration-strategy?authuser=1" \t "_blank) is this:

1. Build new screens with Compose
2. As you're building features, identify reusable elements and start to create a library of common UI components
3. Replace existing features one screen at a time

When migration an app to compose, if you are starting with a screen, specifically a part of the screen while leaving the overall structure of the screen intact, you need a host activity or Fragment to render the UI. You can use ComposeView: and Android View that can host Compose UI content using its setContent method.

Example:

<androidx.core.widget.NestedScrollView  
    android:id="@+id/plant\_detail\_scrollview"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"  
    android:clipToPadding="false"  
    android:paddingBottom="@dimen/fab\_bottom\_padding"  
    app:layout\_behavior="@string/appbar\_scrolling\_view\_behavior">  
  
    <!-- Step 2) Comment out ConstraintLayout and its children –->  
    <androidx.constraintlayout.widget.ConstraintLayout  
        android:layout\_width="match\_parent"  
        android:layout\_height="match\_parent"  
        android:layout\_margin="@dimen/margin\_normal">  
  
        <TextView  
            android:id="@+id/plant\_detail\_name"  
        ...  
      
    </androidx.constraintlayout.widget.ConstraintLayout>  
    <!-- End Step 2) Comment out until here –->  
  
    <!-- Step 3) Add a ComposeView to host Compose code –->  
    <androidx.compose.ui.platform.ComposeView  
        android:id="@+id/compose\_view"  
        android:layout\_width="match\_parent"  
        android:layout\_height="match\_parent"/>  
  
</androidx.core.widget.NestedScrollView>

**Note:** In a production app, a ViewModel should only be referenced by a screen-level composable. If child composables need data from a ViewModel, it is best practice to only pass data that child composables need rather than the whole ViewModel.

Composables don't have their own ViewModel instances, the same instance is shared between the composables and the lifecycle owner that hosts that Compose code (either Activity or Fragment).

Note: LiveData.observeAsState() starts observing the LiveData and represents its values as a State object. Every time there would be a new value posted into the LiveData the returned State will be updated causing recomposition of every State.value usage.

**Note:**AndroidView allows you to create a View programmatically. In case you want to inflate a View from an XML file, you can do it using [view binding](https://developer.android.com/topic/libraries/view-binding?authuser=1) with the AndroidViewBinding API from the androidx.compose.ui:ui-viewbinding library.

### Customize MaterialTheme

To create your own theme, open the Theme.kt file under the theme package. Theme.kt

If you have a custom theme, you can customize the colors, typhography and shapes. You can also use a parameter to determine if the user’s device is using the dark theme or not.

Example:

@Composable  
fun SunflowerTheme(  
    darkTheme: Boolean = isSystemInDarkTheme(),  
    content: @Composable () -> Unit  
) {  
    val lightColors  = lightColorScheme(  
        primary = colorResource(id = R.color.sunflower\_green\_500),  
        primaryContainer = colorResource(id = R.color.sunflower\_green\_700),  
        secondary = colorResource(id = R.color.sunflower\_yellow\_500),  
        background = colorResource(id = R.color.sunflower\_green\_500),  
        onPrimary = colorResource(id = R.color.sunflower\_black),  
        onSecondary = colorResource(id = R.color.sunflower\_black),  
    )  
    val darkColors  = darkColorScheme(  
        primary = colorResource(id = R.color.sunflower\_green\_100),  
        primaryContainer = colorResource(id = R.color.sunflower\_green\_200),  
        secondary = colorResource(id = R.color.sunflower\_yellow\_300),  
        onPrimary = colorResource(id = R.color.sunflower\_black),  
        onSecondary = colorResource(id = R.color.sunflower\_black),  
        onBackground = colorResource(id = R.color.sunflower\_black),  
        surface = colorResource(id = R.color.sunflower\_green\_100\_8pc\_over\_surface),  
        onSurface = colorResource(id = R.color.sunflower\_white),  
    )  
    val colors = if (darkTheme) darkColors else lightColors  
    MaterialTheme(  
        colorScheme = colors,  
        content = content  
    )  
}

You can use the custom theme in the setContent, for example:

class PlantDetailFragment : Fragment() {  
    ...  
    composeView.apply {  
        ...  
        setContent {  
            SunflowerTheme {  
                PlantDetailDescription(plantDetailViewModel)  
            }  
        }  
    }  
}

# Layouts, Theming and Animation

## Lazy Layouts in Compose

In Compose, you can achieve what the RecicleViews do using the Lazy Layouts. We have LazyColumns (vertical lists), LazyRows (horizontal lists) and LazyGrids (both). However, the amount of code you need to write for Lazy Layouts is smaller than the one you need for RecyclerViews.

You have two ways to insert items:

* You can describe one item using the item block.
* Insert multiple items with the items block.

You can also add an index to the elements.

The Lazy state allows you to store the scroll position and contains useful information of your list. To remember your state across compositions, hoist it by using rememberLazyListState. The state offers you the index of your first visible index and other info.

You can use LazyVerticalGrids and LazyHorizontalGrids

Some tips:

* Don’t use 0 px sized items.
* Avoid nesting components scrollable in the same direction.
* Beware of putting multiple elements in one item.
* Consider using custom arrangements.

## Theming in Compose with Material 3

The foundation of a color scheme is the set of five key colors that each relate to a [tonal palette](https://m3.material.io/styles/color/the-color-system/key-colors-tones#a828e350-1551-45e5-8430-eb643e6a7713) of 13 tones which are used by Material 3 components.

Accent colors: Primary Key Color, Secondary Key Color, Tertiary Key Color

Neutral colors: Neutral Key Color, Neutral Variant Key Color.

Each accent color (primary, secondary, and tertiary) is then provided in four compatible colors of different tones for pairing, defining emphasis, and visual expression.

* Primary, On Primary, Primary Container and On Primary Container
* Secondary, On Secondary, Secondary Container and On Secondary Container
* Tertiary, On Tertiary, Tertiary Container and On Tertiary Container

Similarly, neutral colors are also divided into four compatible tones used for surfaces and background. These are also important to emphasize text icons when placed on any surface:

* Background, On Background, Surface, On Suface
* Surface Variant, On Surface Variant, Outline

While you can create a custom [ColorScheme](https://developer.android.com/reference/kotlin/androidx/compose/material3/ColorScheme?authuser=1" \t "_blank) manually, it's often easier to generate one using source colors from your brand. The [Material Theme Builder](https://m3.material.io/theme-builder#/custom) tool allows you to do this, and optionally export Compose theming code.

Color roles of primary, secondary, and tertiary colors.

**Primary** is the base color, which is used for the main components like prominent buttons and active states.

The **secondary** key color is used for less prominent components in the UI, such as filter chips.

The **tertiary** key color is used to provide contrasting accents, and neutral colors are used for the background and surfaces in the app.

Once you have created and imported the new theme, you need to apply it in the application. Wrap the main composable in the MainActivity.kt with the function AppTheme()

1. **Tonal and shadow elevations**

Material 3 represents elevation mainly using tonal color overlays. This is a new way to differentiate containers and surfaces from each other — increasing tonal elevation uses a more prominent tone — in addition to shadows.

### Surface

A Surface is a component that represents a physical layer or plane in the UI. Surfaces are used to create visual hierarchy and structure within an app. Here are some key aspects of Surfaces in Material 3:

1. **Elevation**: Each Surface has an elevation level, which determines its shadow and how it visually relates to other surfaces. Higher elevation surfaces appear to float above lower ones
2. **Tonal Variance**: Surfaces can have different tones or shades, which help in distinguishing different parts of the UI and enhancing visual appeal
3. **Interaction**: Surfaces can respond to user interactions, such as clicks or gestures, by changing their elevation or appearance to provide feedback
4. **Theming**: Surfaces are integral to theming in Material 3, allowing for dynamic color changes and personalization based on user preferences

AppTheme {

Surface(tonalElevation = 5.dp) {

ReplyApp(

replyHomeUIState = uiState,

// other parameters

)

}

}

### Dynamic Colors

Dynamic colors in Materil 3 is an algorithm that derives custom colors from a user's wallpaper to be applied to their apps and system UI.

It is available on Android 12+.  If dynamic color is available, you can set up a dynamic color scheme using dynamicDarkColorScheme() or dynamicLightColorScheme(). If not, you should fall back to using a default light or dark ColorScheme.

### Typography

Compose provides the M3 [Typography](https://developer.android.com/reference/kotlin/androidx/compose/material3/Typography?authuser=1) class — along with the existing [TextStyle](https://developer.android.com/reference/kotlin/androidx/compose/ui/text/TextStyle?authuser=1" \t "_blank) and [font-related](https://developer.android.com/reference/kotlin/androidx/compose/ui/text/font/package-summary?authuser=1) classes — to model the Material 3 type scale.

You can define the typography styles in the ui/theme/Type.kt and then add it to the theme, passing it to the MaterialTheme() composable inside the AppTheme.

@Composable

fun AppTheme(

useDarkTheme: Boolean = isSystemInDarkTheme(),

content: @Composable() () -> Unit

) {

// dynamic theming content

MaterialTheme(

colorScheme = colors,

typography = typography,

content = content

)

}

Just like colors, you'll access the typography style for the current theme using MaterialTheme.typography. This gives you the typography instance to use all defined typography in Type.kt.

### Shapes

Material surfaces can be displayed in different shapes. Shapes direct attention, identify components, communicate state, and express brand.

Compose provides the [Shapes](https://developer.android.com/reference/kotlin/androidx/compose/material3/Shapes?authuser=1) class with expanded parameters to implement new M3 shapes. The M3 shape scale, similar to [type scale](https://m3.material.io/styles/typography/), enables an expressive range of shapes across the UI.

There are different sizes of shapes in the shape scale:

* Extra small
* Small
* Medium
* Large
* Extra large

You can create a Shape.kt file under ui/theme and add something like this:

package com.example.reply.ui.theme  
  
import androidx.compose.foundation.shape.RoundedCornerShape  
import androidx.compose.material3.Shapes  
import androidx.compose.ui.unit.dp  
  
val shapes = Shapes(  
   extraSmall = RoundedCornerShape(4.dp),  
   small = RoundedCornerShape(8.dp),  
   medium = RoundedCornerShape(16.dp),  
   large = RoundedCornerShape(24.dp),  
   extraLarge = RoundedCornerShape(32.dp)  
)

Then, you will use it in the MaterialTheme in the Theme.kt:

@Composable  
fun AppTheme(  
   useDarkTheme: Boolean = isSystemInDarkTheme(),  
   content: @Composable() () -> Unit  
) {  
  // dynamic theming content  
  
   MaterialTheme(  
       colorScheme = colors,  
       typography = typography,  
       shapes = shapes,  
       content = content  
   )  
}

### Emphasis

Emphasis in the UI helps you to highlight some content over the other, such as when you want to differentiate the title from subtitles. Emphasis in M3 uses variations of color and its on-color combinations. You have two ways to add emphasis:

1. Using surface, surface-variant, and background alongside on-surface and on-surface-variants colors from the expanded M3 color system.
2. Using different font weights for text. As you saw in the typography section, you can provide custom weights to your type scale for providing different emphasis.

## Animating Elements

There are several kind of animtations in Compose. One of the simples is the animate\*AsState, that is used when State changes.

val backgroundColor by animateColorAsState(  
        targetValue = if (tabPage == TabPage.Home) Seashell else GreenLight,  
        label = "background color")

### Animated Visibility

In the following example, you have an animation for a Floating Action button, specifically for the text which will be extended or shrink depending of the visibility of the button.

AnimatedVisibility(extended) {  
    Text(  
        text = stringResource(R.string.edit),  
        modifier = Modifier  
            .padding(start = 8.dp, top = 3.dp)  
    )  
}

AnimatedVisibility runs its animation every time the specified Boolean value changes. By default, AnimatedVisibility shows the element by fading in and expanding it, and hides it by fading out and shrinking. This behavior works great for this example with FAB, but we can also customize the behavior.

### Animate content size

You can animate the change of the size of an element by adding the animateContentSize modifier:

Column(  
    modifier = Modifier  
        .fillMaxWidth()  
        .padding(16.dp)  
        .animateContentSize()  
) {  
    // ... the title and the body  
}

animateContentSize can be customized with a custom animationSpec too. We can provide options to change the type of animation from spring to tween etc.

### Animate Multiple Values

Using the Transition API allows us to track when all animations on a Transition are finished, which is not possible when using individual animate\*AsState APIs that we've seen previously. The Transition API also allows us to define different transitionSpec's when transitioning between different states.

To animate these multiple values simultaneously, we can use a Transition. A Transition can be created with the updateTransition function. Pass the index of the tab currently selected as the targetState parameter.

Each animating value can be declared with the animate\* extension functions of Transition.

val transition = updateTransition(tabPage, label = "Tab indicator")  
val indicatorLeft by transition.animateDp(label = "Indicator left") { page ->  
   tabPositions[page.ordinal].left  
}  
val indicatorRight by transition.animateDp(label = "Indicator right") { page ->  
   tabPositions[page.ordinal].right  
}  
val color by transition.animateColor(label = "Border color") { page ->  
   if (page == TabPage.Home) PaleDogwood else Green  
}

### Repeating animations

The Transition API and InfiniteTransition can be used to animate multiple values, while Transition animates values based on state changes, InfiniteTransition animates values indefinitely.

To create an InfiniteTransition, use the rememberInfiniteTransition function. Then, each animating value change can be declared with one of the animate\* extension functions of InfiniteTransition.

val infiniteTransition = rememberInfiniteTransition()  
val alpha by infiniteTransition.animateFloat(  
    initialValue = 0f,  
    targetValue = 1f,  
    animationSpec = infiniteRepeatable(  
        animation = keyframes {  
            durationMillis = 1000  
            0.7f at 500  
        },  
        repeatMode = RepeatMode.Reverse  
    ),  
    label = "alpha"  
)

The default repeatMode is RepeatMode.Restart . This transitions from initialValue to targetValue and starts again at the initialValue. By setting the repeatMode to RepeatMode.Reverse, the animation progresses from initialValue to targetValue and then from targetValue to initialValue. The animation progresses from 0 to 1 then 1 to 0.

### Gesture Animation

[SwipeToDismiss](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary?authuser=1#swipetodismiss) is a Composable in Material that can be used instead of needing to implement your own custom modifier. This section will teach you the thinking behind gesture-based animations.

## Advanced Layout Concepts

Compose transforms data into UI via its three phases:

A close-up of a red rectangle

AI-generated content may be incorrect.

What to show?

Where to place it?

How to render it?

A layout is an abstract term for any Compose UI Element.

A layout node is a visual representation of an element in the UI tree- result of Composition.

Layout Composable: core component of the Compose UI, creates a layout node in the UI tree.

Layout() function: Start of the placement in the Layout phase; places children in a parent layout.

.layout() Modifier: wraps a single layout node to size and place it individually.

During the layout phase, every element in the UI tree measures its children, if any, and decide its own size and finally place the children in the available 2D space.

## Constraints and Modifiers order

Modifiers in compose can be chained and the order in which we do it matters.

The modifiers can affect different phases of the data being transformed into UI, for example, the size and padding can affect the layout phase, while the clip modifier affects the Drawing phase.

Constraints help finding the right sizes for our nodes. They are the min and max bounds for a node’s size.

Constraints are passed down from parent to child in the UI tree during the layout phase.

# Architecture and State

## Compose Phases

To transform Data into UI, compose uses 3 phases:

1. **Composition**

What to show. Here, compose executes your composable functions and outputs a tree data structure, that represents the UI

1. **Layout**

Where to place.

Each element in the tree measures its children and places them in the available space.

The tree is reversed using the following three-step algorithm:

* + A node measures its children.
  + Decide its own size.
  + Places its children.

Each node is visited only once.

1. **Drawing**

How to render it.

Each node in a tree draws its pixels on the screen.

The tree is traversed again from top to bottom and each node draws itself on the screen in turn

The modifiers are like wrapper nodes for our layout nodes. When we chain multiple modifiers, each modifier node wraps the rest of the chain and the layout node with it.

## Architecting your Compose UI

In Compose the UI is immutable—there's no way to update it after it's been drawn. What you can control is the state of your UI. Every time the state of the UI changes, Compose [recreates the parts of the UI tree that have changed](https://developer.android.com/develop/ui/compose/mental-model?authuser=1#recomposition).

### Unidirectional Data Flow

A *unidirectional data flow* (UDF) is a design pattern where state flows down and events flow up. By following unidirectional data flow, you can decouple composables that display state in the UI from the parts of your app that store and change state.

The UI update loop for an app using unidirectional data flow looks like this:

* **Event**: Part of the UI generates an event and passes it upward, such as a button click passed to the ViewModel to handle; or an event is passed from other layers of your app, such as indicating that the user session has expired.
* **Update state**: An event handler might change the state.
* **Display state**: The state holder passes down the state, and the UI displays it.

**Keys**

**mutableStateOf(value)** creates a **MutableState**, which is an observable type in Compose. Any changes to its value will schedule recomposition of any composable functions that read that value.

**remember** stores objects in the composition, and forgets the object when the composable that called **remember** is removed from the composition.

**rememberSaveable** retains the state across configuration changes by saving it in a **Bundle**.

To encourage decoupling and reuse, each composable should hold the least amount of information possible.

### Events in Compose

Every input to your app should be represented as an event: taps, text changes, and even timers or other updates. As these events change the state of your UI, the ViewModel should be the one to handle them and update the UI state.

Prefer passing immutable values for state and event handler lambdas. This approach has the following benefits:

* You improve reusability.
* You ensure that your UI doesn't change the value of the state directly.
* You avoid concurrency issues because you make sure that the state isn't mutated from another thread.
* Often, you reduce code complexity.

### ViewModels, States and Events

By using ViewModel and mutableStateOf, you can also introduce unidirectional data flow in your app if one of the following is true:

* The state of your UI is exposed via observable state holders, like StateFlow or LiveData.
* The ViewModel handles events coming from the UI or other layers of your app and updates the state holder based on the events.

For example, when implementing a sign-in screen, tapping on a *Sign in* button should cause your app to display a progress spinner and a network call. If the login was successful, then your app navigates to a different screen; in case of an error the app shows a Snackbar. Here's how you would model the screen state and the event:

The screen has four states:

* **Signed out**: when the user hasn't signed in yet.
* **In progress**: when your app is currently trying to sign the user in by performing a network call.
* **Error**: when an error occurred while signing in.
* **Signed in**: when the user is signed in.

You can model these states as a sealed class. The ViewModel exposes the state as a State, sets the initial state, and updates the state as needed. The ViewModel also handles the sign-in event by exposing an onSignIn() method.

In addition to the mutableStateOf API, Compose [provides extensions](https://developer.android.com/develop/ui/compose/interop?authuser=1#streams) for LiveData, Flow, and Observable to register as a listener and represent the value as a state.

class MyViewModel : ViewModel() {

private val \_uiState = MutableLiveData<UiState>(UiState.SignedOut)

val uiState: LiveData<UiState>

get() = \_uiState

// ...

}

@Composable

fun MyComposable(viewModel: MyViewModel) {

val uiState = viewModel.uiState.observeAsState()

// ...

}

## State Changes

Compose has a special state-tracking system in place that schedules recompositions for any composables that read a particular state.

By using mutableStateOf you can keep track of a specific state.

State created in the composition needs to be remembered.

State hoisting is a pattern of moving private state out of a composable to make it less stateful, therefore, more reusable.

## Advanced State and Side Effects in Compose

### UI State Pipeline

A good architecture for an application is organized in layers to obey basic good system design practices, like separation of concerns and testability.

[UI State production](https://developer.android.com/topic/architecture/ui-layer/state-production?authuser=1) refers to the process in which the app accesses the data layer, applies business rules if needed, and exposes UI state to be consumed from the UI.

There are a few APIs that can be used to produce UI state. The alternatives are summarized in the [Output types in state production pipelines](https://developer.android.com/topic/architecture/ui-layer/state-production?authuser=1#output-types) documentation. In general, it is a good practice to use Kotlin's [StateFlow](https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/" \t "_blank) to produce UI state.

### Consuming a Flow

The following code is an example of having a mutable varible of type MutableStateFlow and another immutable variableof type StateFlow that is public read-only and can be consumed from the UI. Exposing a read-only variable while using the mutable variable internally is a good practice. By doing this, you ensure the UI state cannot be modified unless it is through the ViewModel, which makes it the single source of truth. The extension function [asStateFlow](https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/as-state-flow.html" \t "_blank) converts the flow from mutable to immutable.

private val \_suggestedDestinations = MutableStateFlow<List<ExploreModel>>(emptyList())  
val suggestedDestinations: StateFlow<List<ExploreModel>> = \_suggestedDestinations.asStateFlow()

collectAsStateWithLifecycle() collects values from the StateFlow and represents the latest value via Compose's [State](https://developer.android.com/reference/kotlin/androidx/compose/runtime/State?authuser=1) API in a lifecycle-aware manner. This will make the Compose code that reads that state value recompose on new emissions.

Example of use of collectAsStateWithLifecycle:

import androidx.lifecycle.compose.collectAsStateWithLifecycle  
  
@Composable  
fun CraneHomeContent(  
    onExploreItemClicked: OnExploreItemClicked,  
    openDrawer: () -> Unit,  
    modifier: Modifier = Modifier,  
    viewModel: MainViewModel = viewModel(),  
) {  
    val suggestedDestinations by viewModel.suggestedDestinations.collectAsStateWithLifecycle()  
    // ...  
}

Compose also offers APIs for Android's most popular stream-based solutions:

* [LiveData.observeAsState()](https://developer.android.com/reference/kotlin/androidx/compose/runtime/livedata/package-summary?authuser=1#observeAsState(androidx.lifecycle.LiveData)) included in the androidx.compose.runtime:runtime-livedata:$composeVersion artifact.
* [Observable.subscribeAsState()](https://developer.android.com/reference/kotlin/androidx/compose/runtime/rxjava2/package-summary?authuser=1#subscribeAsState(io.reactivex.Observable,kotlin.Any)) included in the androidx.compose.runtime:runtime-rxjava2:$composeVersion or androidx.compose.runtime:runtime-rxjava3:$composeVersion artifact.

### LaunchedEffect and rememberUpdatedState

You can have a landing screen in your app that can be used to load all the data needed in the background.

Kotlin coroutines are the recommended way to perform asynchronous operations in Android. An app would usually use coroutines to load things in the background when it starts. Jetpack Compose offers APIs that make using coroutines safe within the UI layer

**A side-effect in Compose is a change to the state of the app that happens outside the scope of a composable function.** For example, opening a new screen when the user taps on a button, or showing a message when the app doesn't have Internet connection.

LaunchedEfect API triggers a coroutine-scoped side-effect in Compose, so you can call suspend functions safely from inside a composable.

When LaunchedEffect enters the Composition, it launches a coroutine with the block of code passed as a parameter. The coroutine will be canceled if LaunchedEffect leaves the composition

You should use rememberUpdatedState when a long-lived lambda or object expression references parameters or values computed during composition, which might be common when working with LaunchedEffect.

Example:

// home/LandingScreen.kt file  
  
import androidx.compose.runtime.getValue  
import androidx.compose.runtime.rememberUpdatedState  
import kotlinx.coroutines.delay  
  
@Composable  
fun LandingScreen(onTimeout: () -> Unit, modifier: Modifier = Modifier) {  
    Box(modifier = modifier.fillMaxSize(), contentAlignment = Alignment.Center) {  
        // This will always refer to the latest onTimeout function that  
        // LandingScreen was recomposed with  
        val currentOnTimeout by rememberUpdatedState(onTimeout)  
  
        // Create an effect that matches the lifecycle of LandingScreen.  
        // If LandingScreen recomposes or onTimeout changes,   
        // the delay shouldn't start again.  
        LaunchedEffect(Unit) {  
            delay(SplashWaitTime)  
            currentOnTimeout()  
        }  
  
        Image(painterResource(id = R.drawable.ic\_crane\_drawer), contentDescription = null)  
    }  
}

### rememberCoroutineScope

Suspend functions, in addition to being able to run asynchronous code, also help represent concepts that happen over time. As opening the drawer requires some time, movement, and potential animations, that's perfectly reflected with the suspend function, which will suspend the execution of the coroutine where it's been called until it finishes and resumes execution.

Using the [rememberCoroutineScope](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1" \l "rememberCoroutineScope(kotlin.Function0)" \t "_blank) API returns a CoroutineScope bound to the point in the Composition where you call it. The scope will be automatically canceled once it leaves the Composition. With that scope, you can start coroutines when you're not in the Composition, for example, in the openDrawer callback.

The scaffoldState.drawerState.open() must be called within a coroutine but you cannot simply call suspend function in it because openDrawer is not executed in the context of a coroutine. And you cannot use LaunchedEffect because we cannot call composables in openDrawer, we are not in the Composition.

// home/CraneHome.kt file  
  
import androidx.compose.runtime.rememberCoroutineScope  
import kotlinx.coroutines.launch  
  
@Composable  
fun CraneHome(  
    onExploreItemClicked: OnExploreItemClicked,  
    modifier: Modifier = Modifier,  
) {  
    val scaffoldState = rememberScaffoldState()  
    Scaffold(  
        scaffoldState = scaffoldState,  
        modifier = Modifier.statusBarsPadding(),  
        drawerContent = {  
            CraneDrawer()  
        }  
    ) {  
        val scope = rememberCoroutineScope()  
        CraneHomeContent(  
            modifier = modifier,  
            onExploreItemClicked = onExploreItemClicked,  
            openDrawer = {  
                scope.launch {  
                    scaffoldState.drawerState.open()  
                }  
            }  
        )  
    }  
}

1. **LaunchedEffect vs rememberCoroutineScope**

Using LaunchedEffect in this case wasn't possible because you needed to trigger the call to create a coroutine in a regular callback that was outside of the Composition.

LaunchedEffect guarantees that the side-effect will be executed when the call to that composable makes it into the Composition. If you use rememberCoroutineScope and scope.launch in the body of the LandingScreen, the coroutine will be executed every time LandingScreen is called by Compose regardless of whether that call makes it into the Composition or not. Therefore, you'll waste resources and you won't be executing this side-effect in a controlled environment.

### Create a State holder

By creating a state holder responsible for the internal state of a composable, you can centralize all state changes in one place. With this, it's more difficult for the state to be out of sync, and the related logic is all grouped together in a single class. Furthermore, this state can be easily hoisted up and can be consumed from callers of this composable.

State holders always need to be remembered in order to keep them in the Composition and not create a new one every time. It's a good practice to create a method in the same file that does this to remove boilerplate and avoid any mistakes that might occur.

### Custom Saver

A [Saver](https://developer.android.com/reference/kotlin/androidx/compose/runtime/saveable/package-summary?authuser=1" \l "Saver(kotlin.Function2,kotlin.Function1)" \t "_blank) describes how an object can be converted into something which is [Saveable](https://developer.android.com/reference/kotlin/androidx/compose/runtime/saveable/Saver?authuser=1" \t "_blank). Implementations of a [Saver](https://developer.android.com/reference/kotlin/androidx/compose/runtime/saveable/package-summary?authuser=1" \l "Saver(kotlin.Function2,kotlin.Function1)" \t "_blank) need to override two functions:

* save to convert the original value to a saveable one.
* restore to convert the restored value to an instance of the original class.

DisposableEffect

DisposableEffect is meant for side effects that need to be cleaned up after the keys change or the composable leaves the Composition.

Produce State

produceState allows you to convert non-Compose state into Compose [State](https://developer.android.com/reference/kotlin/androidx/compose/runtime/State?authuser=1). It launches a coroutine scoped to the Composition that can push values into the returned State using the value property. As with LaunchedEffect, produceState also takes keys to cancel and restart the computation.

derivedStateOf

Use derivedStateOf when you want a Compose State that's derived from another State. The derivedStateOf calculation block is executed every time the internal state changes, but the composable function only recomposes when the result of the calculation is different from the last one.

You learned about how to create state holders, side effect APIs such as [LaunchedEffect](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1#LaunchedEffect(kotlin.Any,kotlin.coroutines.SuspendFunction1)), [rememberUpdatedState](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1#rememberUpdatedState(kotlin.Any)), [DisposableEffect](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1#DisposableEffect(kotlin.Any,kotlin.Function1)), [produceState](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1#produceState(kotlin.Any,kotlin.coroutines.SuspendFunction1)), and [derivedStateOf](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary?authuser=1" \l "derivedStateOf(kotlin.Function0)" \t "_blank), and how to use coroutines in Jetpack Compose.

## Best Practices with Navigation Compose

* Define your screen as state in, events out.
* Split up your navigation graphs just like your split your screens.
* Build only the public APIs you need.
* User your module structure to guide how to combine your graphs together.
* Check out the updated resources

## Jetpack Compose Navigation

When you are migrating an app to Compose, the basic migration steps are:

1. Add the latest [Compose Navigation dependency](https://mvnrepository.com/artifact/androidx.navigation/navigation-compose" \t "_blank)
2. Set up the [NavController](https://developer.android.com/jetpack/compose/navigation?authuser=1" \l "getting-started" \t "_blank)
3. Add a [NavHost](https://developer.android.com/jetpack/compose/navigation?authuser=1" \l "create-navhost" \t "_blank) and create the navigation graph
4. Prepare routes for navigating between different app destinations
5. Replace the current navigation mechanism with Compose Navigation

### NavController

The [NavController](https://developer.android.com/jetpack/compose/navigation?authuser=1" \l "getting-started" \t "_blank) is the central component when using Navigation in Compose. It keeps track of back stack composable entries, moves the stack forward, enables back stack manipulation, and navigates between destination states. Because NavController is central to navigation, it has to be created as a first step in setting up Compose Navigation.

A NavController is obtained by calling the [rememberNavController()](https://developer.android.com/reference/kotlin/androidx/navigation/compose/package-summary?authuser=1" \l "rememberNavController(kotlin.Array)" \t "_blank) function. This creates and [remembers](https://developer.android.com/jetpack/compose/state?authuser=1#state-in-composables) a NavController which survives configuration changes (using [rememberSaveable](https://developer.android.com/reference/kotlin/androidx/compose/runtime/saveable/package-summary?authuser=1" \l "rememberSaveable(kotlin.Array,androidx.compose.runtime.saveable.Saver,kotlin.String,kotlin.Function0)" \t "_blank)).

You should always create and place the NavController at the top level in your composable hierarchy, usually within your App composable. Then, all composables that need to reference the NavController have access to it. This follows the principles of [state hoisting](https://developer.android.com/jetpack/compose/state?authuser=1#state-hoisting) and ensures the NavController is the main source of truth for navigating between composable screens and maintaining the back stack.

Example:

import androidx.navigation.compose.rememberNavController  
// ...  
  
@Composable  
fun RallyApp() {  
    RallyTheme {  
        var currentScreen: RallyDestination by remember { mutableStateOf(Overview) }  
        val navController = rememberNavController()  
        Scaffold(  
            // ...  
        ) {   
            // ...  
       }  
}

### Routes in Compose

You need to add the routes to the navigation graph, with a default route or start destination when the app is launched.

Each composable destination in your graph is associated with a route. Routes are represented as Strings that define the path to your composable and guide your navController to land on the right place. You can think of it as an implicit deep link that leads to a specific destination. **Each destination must have a unique route**.

### Call the NavHost composable

The 3 main parts of Navigation are the NavController, NavGraph, and NavHost. The NavController is always associated with a single NavHost composable. The NavHost acts as a container and is responsible for displaying the current destination of the graph. As you navigate between composables, the content of the NavHost is automatically [recomposed](https://developer.android.com/jetpack/compose/mental-model?authuser=1#recomposition). It also links the NavController with a navigation graph ( [NavGraph](https://developer.android.com/reference/androidx/navigation/NavGraph?authuser=1" \t "_blank)) that maps out the composable destinations to navigate between. It is essentially a collection of fetchable destinations.

The final parameter builder: NavGraphBuilder.() -> Unit is responsible for defining and building the navigation graph. It uses the lambda syntax from the [Navigation Kotlin DSL](https://developer.android.com/guide/navigation/navigation-kotlin-dsl?authuser=1#navgraphbuilder), so it can be passed as a trailing lambda inside the body of the function and pulled out of the parentheses:

import androidx.navigation.compose.composable  
// ...  
  
NavHost(  
    navController = navController,  
    startDestination = Overview.route,  
    modifier = Modifier.padding(innerPadding)  
) {   
    composable(route = Overview.route) {  
        Overview.screen()  
    }  
    composable(route = Accounts.route) {  
        Accounts.screen()  
    }  
    composable(route = Bills.route) {  
        Bills.screen()  
    }  
}

The navigate method from navControlles is used to navigate to a specific route, example:

@Composable  
fun RallyApp() {  
    RallyTheme {  
        var currentScreen: RallyDestination by remember { mutableStateOf(Overview) }  
        val navController = rememberNavController()  
        Scaffold(  
            topBar = {  
                RallyTabRow(  
                    allScreens = rallyTabRowScreens,  
                    // Pass the callback like this,  
                    // defining the navigation action when a tab is selected:  
                    onTabSelected = { newScreen ->  
                        navController.navigate(newScreen.route)  
                    },  
                    currentScreen = currentScreen,  
                )  
            }

You can use launchSingleTop flag in the navigate method to avoid the launch of multiple copies of the same destination.

You can create a helper extension instead of copy pasting this flag to all your navigate() calls:

import androidx.navigation.NavHostController  
// ...  
  
fun NavHostController.navigateSingleTopTo(route: String) =  
    this.navigate(route) { launchSingleTop = true }

@Composable  
fun RallyApp() {  
    RallyTheme {  
        var currentScreen: RallyDestination by remember { mutableStateOf(Overview) }  
        val navController = rememberNavController()  
        Scaffold(  
            topBar = {  
                RallyTabRow(  
                    allScreens = rallyTabRowScreens,  
                    onTabSelected = { newScreen ->  
                        navController  
                            .navigateSingleTopTo(newScreen.route)  
                    },  
                    currentScreen = currentScreen,  
                )  
            }

There are additional options you can use within the same navigateSinelTopTo extension function:

* **launchSingleTop = true** - as mentioned, this makes sure there will be at most one copy of a given destination on the top of the back stack In Rally app, this would mean that re-tapping the same tab multiple times doesn't launch multiple copies of the same destination
* **popUpTo(startDestination) { saveState = true }** - pop up to the start destination of the graph to avoid building up a large stack of destinations on the back stack as you select tabs

In Rally, this would mean that pressing the back arrow from any destination would pop the entire back stack to Overview

* **restoreState = true** - determines whether this navigation action should restore any state previously saved by PopUpToBuilder.saveState or the popUpToSaveState attribute. Note that, if no state was previously saved with the destination ID being navigated to, **this has no effect**

In Rally, this would mean that, re-tapping the same tab would keep the previous data and user state on the screen without reloading it again

To get real time updates on your current destination from the back stack in a form of [State](https://developer.android.com/reference/kotlin/androidx/compose/runtime/State?authuser=1), you can use [navController.currentBackStackEntryAsState()](https://developer.android.com/reference/kotlin/androidx/navigation/compose/package-summary?authuser=1" \l "(androidx.navigation.NavController).currentBackStackEntryAsState()" \t "_blank) and then grab its current destination:

import androidx.navigation.compose.currentBackStackEntryAsState  
import androidx.compose.runtime.getValue  
// ...  
  
@Composable  
fun RallyApp() {  
    RallyTheme {  
        val navController = rememberNavController()  
  
        val currentBackStack by navController.currentBackStackEntryAsState()  
        val currentDestination = currentBackStack?.destination  
  
        // Change the variable to this and use Overview as a backup screen if this returns null  
        val currentScreen = rallyTabRowScreens.find { it.route == currentDestination?.route } ?: Accounts  
        // ...  
    }  
}

To handle the click events of the screens, we usually will receive the click handlers in the composables we will open and from the NavHost, we will send the callbacks for those events. For example, the OverviewScreen accepts several functions as callbacks to set as click events, so we define the actions these buttons will execute by sending the callbacks like this:

OverviewScreen(  
    onClickSeeAllAccounts = {  
        navController.navigateSingleTopTo(Accounts.route)   
    },  
    onClickSeeAllBills = {   
        navController.navigateSingleTopTo(Bills.route)   
    }  
)

### Navigating to a screen with arguments

Arguments are a very powerful tool that make navigation routing dynamic by passing one or more arguments to a route. It enables displaying different information based on the different arguments provided.

**Note:** A named argument is defined by appending it to routes inside curly braces, like this: {argument}. It is a syntax that looks similar to Kotlin's String template syntax, using the dollar sign, if necessary, to escape variable names, like: {${argument}}

To pass the argument alongside your route when navigating, you need to append them together, following a pattern: "route/{argument}". In your case, that would look like this: "${SingleAccount.route}/{${SingleAccount.accountTypeArg}}". Remember that $ sign is used to escape variables:

import androidx.navigation.NavType  
import androidx.navigation.compose.navArgument  
// ...  
  
composable(  
    route =  
        "${SingleAccount.route}/{${SingleAccount.accountTypeArg}}"  
) {   
    SingleAccountScreen()  
}

The composable needs to be aware that it should accept arguments. You do that by defining its arguments parameter. You could define as many arguments as you need and define its type.

import androidx.navigation.NavType  
import androidx.navigation.compose.navArgument  
// ...  
  
composable(  
    route =  
        "${SingleAccount.route}/{${SingleAccount.accountTypeArg}}",  
    arguments = listOf(  
        navArgument(SingleAccount.accountTypeArg) { type = NavType.StringType }  
    )  
) {   
    SingleAccountScreen()  
}

The screen or composable you are sending the parameters, should also expect them. In this example, SingleAccount composable is waiting for an accountType as part of its parameters.

fun SingleAccountScreen(  
    accountType: String? = UserData.accounts.first().name  
) {   
   // ...   
}

To retrieve the passed argument value, you can access the current NavBackStackEntry (a class that holds the information on the current route and passed arguments of an entry in the back stack). You can use the navBackStackEntry to get the arugments that you need to pass down to the SingleAccountScreen

NavHost(...) {  
    // ...  
    composable(  
        route =  
          "${SingleAccount.route}/{${SingleAccount.accountTypeArg}}",  
        arguments = SingleAccount.arguments  
    ) { navBackStackEntry ->  
        // Retrieve the passed argument  
        val accountType =  
            navBackStackEntry.arguments?.getString(SingleAccount.accountTypeArg)  
  
        // Pass accountType to SingleAccountScreen  
        SingleAccountScreen(accountType)  
    }  
}

### Deep Link Support

In addition to adding arguments, you can also add [deep links](https://developer.android.com/jetpack/compose/navigation?authuser=1#deeplinks) to associate a specific URL, action, and/or mime type with a composable. In Android, a deep link is a link that takes you directly to a specific destination within an app.

Since exposing deep links to external apps isn't enabled by default , you must also add <intent-filter> elements to your app's manifest.xml file, so this will be your first step.

In this example, you create a new intent filter via <intent-filter> inside of the <activity>, with the action VIEW and categories BROWSABLE and DEFAULT.

<activity  
    android:name=".RallyActivity"  
    android:windowSoftInputMode="adjustResize"  
    android:label="@string/app\_name"  
    android:exported="true">  
    <intent-filter>  
        <action android:name="android.intent.action.MAIN" />  
        <category android:name="android.intent.category.LAUNCHER" />  
    </intent-filter>  
    <intent-filter>  
        <action android:name="android.intent.action.VIEW" />  
        <category android:name="android.intent.category.DEFAULT" />  
        <category android:name="android.intent.category.BROWSABLE" />  
        <data android:scheme="rally" android:host="single\_account" />  
    </intent-filter>  
</activity>

In the composable in the NavHost, add the deepLinks parameter:

import androidx.navigation.navDeepLink  
// ...  
  
composable(  
    route = SingleAccount.routeWithArgs,  
    // ...  
    deepLinks = listOf(navDeepLink {  
        uriPattern = "rally://${SingleAccount.route}/{${SingleAccount.accountTypeArg}}"  
    })  
)

# Accessibility, Testing and Performance

To enable Compose Tests in a project, you need some dependencies in the build.gradle file:

androidTestImplementation "androidx.compose.ui:ui-test-junit4:$version"  
  
debugImplementation "androidx.compose.ui:ui-test-manifest:$rootProject.composeVersion"

## Create a simple UI Test

Create a test folder if is not already created. You will add it under the src folder, just right click the src folder and choose Directory, it will allow you to create an androidTest/java folder. Inside this folder you can create a package with the same package name as the app.

Then, create a new file in that folder, for example: TopAppBarTest.kt under app/src/androidTest/com/example/compose/rally

Compose comes with a ComposeTestRule that you can obtain by calling createComposeRule(). This rule lets you set the Compose content under test and interact with it.

package com.example.compose.rally  
  
import androidx.compose.ui.test.junit4.createComposeRule  
import org.junit.Rule  
  
class TopAppBarTest {  
  
    @get:Rule  
    val composeTestRule = createComposeRule()  
      
    // TODO: Add tests  
}

### Testing in Isolation

In Compose you can choose what Compose UI content to use in the test. This is done with the setContent method of the ComposeTestRule, and you can call it anywhere but just once:

    @Test  
    fun rallyTopAppBarTest() {  
        val allScreens = RallyScreen.values().toList()  
        composeTestRule.setContent {   
            RallyTopAppBar(  
                allScreens = allScreens,  
                onTabSelected = { },  
                currentScreen = RallyScreen.Accounts  
            )  
        }  
        Thread.sleep(5000)  
    }

### Finding UI elements

Finding UI elements, checking their properties and performing actions is done through the test rule, following this pattern:

composeTestRule{.finder}{.assertion}{.action}

A good way to understand what tools are available is using the [Compose Testing Cheat Sheet](https://developer.android.com/jetpack/compose/testing-cheatsheet?authuser=1) or the [test package reference documentation](https://developer.android.com/reference/kotlin/androidx/compose/ui/test/package-summary?authuser=1). Look for finders and assertions that might help in our situation. For example: onNodeWithText, onNodeWithContentDescription, isSelected, hasContentDescription, assertIsSelected...

Every UI element is considered as a Node, so we can use th onNodeWithContentDescription to look for an element with the text we want:

The assertIsSelected is what do the job. In this case, this method checks that the Tab with that text is selected in the bar.

import androidx.compose.ui.test.assertIsSelected  
import androidx.compose.ui.test.onNodeWithContentDescription  
...  
  
@Test  
fun rallyTopAppBarTest\_currentTabSelected() {  
    val allScreens = RallyScreen.values().toList()  
    composeTestRule.setContent {  
        RallyTopAppBar(  
            allScreens = allScreens,  
            onTabSelected = { },  
            currentScreen = RallyScreen.Accounts  
        )  
    }  
  
    composeTestRule  
        .onNodeWithContentDescription(RallyScreen.Accounts.name)  
        .assertIsSelected()  
}

### Semantics Tree

Compose tests use a structure called the [**semantics tree**](https://developer.android.com/jetpack/compose/testing?authuser=1#semantics) to look for elements on the screen and read their properties. This is the structure that accessibility services use as well, as they're meant to be read by a service such as [TalkBack](https://support.google.com/accessibility/android/answer/6283677?authuser=1" \t "_blank).

You can print the Semantics tree using the printToLog function on a node.

When you see the property MergeDescendats = true in the log, that means that this node has descendants but they have been merged into it. In order to verify whether the Text inside the tab is displayed or not, we can query the **unmerged** Semantics tree passing useUnmergedTree = true to the onRoot finder.

@Test  
fun rallyTopAppBarTest\_currentLabelExists() {  
    val allScreens = RallyScreen.values().toList()  
    composeTestRule.setContent {  
        RallyTopAppBar(  
            allScreens = allScreens,  
            onTabSelected = { },  
            currentScreen = RallyScreen.Accounts  
        )  
    }  
  
    composeTestRule.onRoot(useUnmergedTree = true).printToLog("currentLabelExists")  
  
  
}

import androidx.compose.ui.test.hasParent  
import androidx.compose.ui.test.hasText  
...  
  
@Test  
fun rallyTopAppBarTest\_currentLabelExists() {  
    val allScreens = RallyScreen.values().toList()  
    composeTestRule.setContent {  
        RallyTopAppBar(  
            allScreens = allScreens,  
            onTabSelected = { },  
            currentScreen = RallyScreen.Accounts  
        )  
    }  
  
    composeTestRule  
        .onNode(  
            hasText(RallyScreen.Accounts.name.uppercase()) and  
            hasParent(  
                hasContentDescription(RallyScreen.Accounts.name)  
            ),  
            useUnmergedTree = true  
        )  
        .assertExists()  
}

### Sychronization

Any test that you write must be properly synchronized with the subject under test. For example, when you use a finder such as onNodeWithText, the test waits until the app is *idle* before querying the semantics tree. Without synchronization, tests could look for elements before they're displayed or they could wait unnecessarily.

In the next example, the Overview screen has an animation that never ends. One approach to fix this test would be to disable animations in developer options. It's one of the widely accepted ways of dealing with it in the View world.

In Compose, the animation APIs were designed with testability in mind, so the problem can be fixed by using the correct API. Instead of restarting the [animateDpAsState](https://developer.android.com/jetpack/compose/animation?authuser=1" \l "animate-as-state" \t "_blank) animation, we can use [infinite animations](https://developer.android.com/jetpack/compose/animation?authuser=1#rememberinfinitetransition).

### Accessibility

To provide the best experience for users with accessibility needs, we can provide a specific description that explains what will happen when the user clicks this element.

For example, having a post card to read a full post, by default this will read out the content of the clickable element, followed by the text "Double tap to activate". Instead, we'd like to be more specific and use "Double tap to read article".

The clickable modifier includes a parameter that allows you to directly set this click label, example:

@Composable  
fun PostCardHistory(  
   // ...  
) {  
   Row(  
       Modifier.clickable(  
               // R.string.action\_read\_article = "read article"  
               onClickLabel = stringResource(R.string.action\_read\_article)  
           ) {  
               navigateToArticle(post.id)  
           }  
   ) {  
       // ...  
   }  
}

With composables like the Card that don’t allow you to directly set the click label, you can use the semantics modifier:

@Composable  
fun PostCardPopular(  
   post: Post,  
   navigateToArticle: (String) -> Unit,  
   modifier: Modifier = Modifier  
) {  
   val readArticleLabel = stringResource(id = R.string.action\_read\_article)  
   Card(  
       shape = MaterialTheme.shapes.medium,  
       modifier = modifier  
          .size(280.dp, 240.dp)  
          .semantics { onClick(label = readArticleLabel, action = null) },  
       onClick = { navigateToArticle(post.id) }  
   ) {  
       // ...  
   }  
}

### Visual elements

Visual composables like [Image](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary?authuser=1#Image(androidx.compose.ui.graphics.vector.ImageVector,kotlin.String,androidx.compose.ui.Modifier,androidx.compose.ui.Alignment,androidx.compose.ui.layout.ContentScale,kotlin.Float,androidx.compose.ui.graphics.ColorFilter)) and [Icon](https://developer.android.com/reference/androidx/wear/compose/material/IconKt?authuser=1#Icon(androidx.compose.ui.graphics.vector.ImageVector,kotlin.String,androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Color)) include a parameter contentDescription. Here you pass a **localized** description of that visual element, or null if the element is purely decorative.

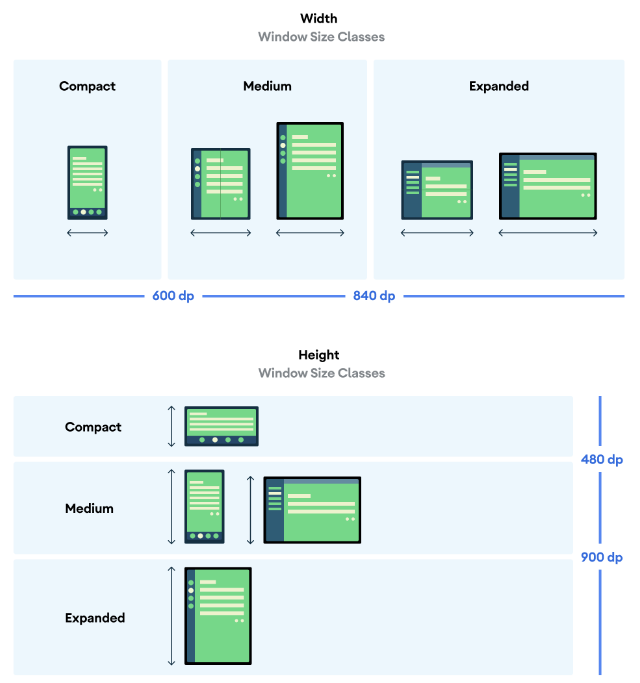
### Switches and Checkboxes

Toggleable elements like Switch and Checkbox read out loud their checked state as they are selected by TalkBack. Without context it can be hard to understand what these toggleable elements refer to though. We can include context for a toggleable element by lifting the toggleable state up, so a user can toggle the Switch or Checkbox by either pressing the composable itself, or the label that describes it.

# Form Factors

One of the important aspects when you need to make the app adaptive, is the Width. For that, you can use the Window Size Class. Window size class is always determined by the screen space available to the app, which may not be the entire physical screen for multitasking or other segmentations.

* A Compact size refers to a screen narrower than 600 dp (most of the smartphones in portrait mode).
* A Medium size refers to a screen width of 600 dp up 840 dp (tablets, foldable in portrait mode).
* An Expanded size refers to a screen width of 840 dp or more (tablets and foldables in landscape, desktop environments).



Both width and height are classified separately, so at any point in time, your app has two window size classes—one for width and one for height. Available width is usually more important than available height due to the ubiquity of vertical scrolling, so for this case you'll also use width size classes.

### Fold States

Foldable devices present yet more situations your app can adapt to because of their varying sizes and the presence of hinges. Hinges can obscure part of the display, making that area unsuitable to show content; they could also be separating, meaning there are two separate physical displays when the device is unfolded.

### Get adaptive information

The Material3 [adaptive](https://developer.android.com/jetpack/androidx/releases/compose-material3-adaptive?authuser=1) library provides convenient access to information about the window your app is running in.

1. Add entries for this artifact and its version to the version catalog file:

**gradle/libs.versions.toml**

[versions]  
material3Adaptive = "1.0.0"  
  
[libraries]  
androidx-material3-adaptive = { module = "androidx.compose.material3.adaptive:adaptive", version.ref = "material3Adaptive" }

1. In the build file of the app module, add the new library dependency and then perform a Gradle sync:

**app/build.gradle.kts**

dependencies {  
  
    implementation(libs.androidx.material3.adaptive)  
}

The following example, shows the use of currentWindowAdaptiveInfo and WindowAdaptiveInfo

override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    setContent {  
        ReplyTheme {  
            val uiState by viewModel.uiState.collectAsStateWithLifecycle()  
            ReplyApp(  
                replyHomeUIState = uiState,  
                onEmailClick = viewModel::setSelectedEmail  
            )  
  
            val adaptiveInfo = currentWindowAdaptiveInfo()  
            val sizeClassText =  
                "${adaptiveInfo.windowSizeClass.windowWidthSizeClass}\n" +  
                "${adaptiveInfo.windowSizeClass.windowHeightSizeClass}"  
            Text(  
                text = sizeClassText,  
                color = Color.Magenta,  
                modifier = Modifier.padding(  
                    WindowInsets.safeDrawing.asPaddingValues()  
                )  
            )  
        }  
    }  
}

## Dynamic Navigation

As you design your app and decide where to place interactive UI elements in your layout, consider the ergonomic implications of different regions of the screen.

* Which areas are comfortable to reach while holding the device?
* Which areas can be reached only by extending fingers, which may be inconvenient?
* Which areas are challenging to reach or are far away from where the user holds the device?

### Bottom Navigation

Bottom navigation is perfect for **compact** sizes, as we naturally hold the device where our thumb can easily reach all the bottom navigation touch points. Use it whenever you have a compact device size or a foldable in a compact folded state.

### Navigation rail

For a **medium** width window size, the navigation rail is ideal for reachability as our thumb naturally falls along the side of the device. You can also combine a navigation rail with a navigation drawer to show more information.

### Navigation drawer

The navigation drawer provides an easy way to see detailed information for navigation tabs, and is easily accessible when you're using **tablets or larger devices**. There are two kinds of navigation drawers available: a modal navigation drawer and a permanent navigation drawer.

### Implement Dynamic Navigation

You can use Material NavigationSuiteScaffold component to automatically switch between the different navigation components based on information like the current window size class.

You need to add the dependencies for Adaptive Nav Suite, in the libs.versions.toml and the build.gradle.kts.

You can use it like this:

@Composable  
private fun ReplyNavigationWrapperUI(  
    content: @Composable () -> Unit = {}  
) {  
    var selectedDestination: ReplyDestination by remember {  
        mutableStateOf(ReplyDestination.Inbox)  
    }

    val windowSize = with(LocalDensity.current) {  
        currentWindowSize().toSize().toDpSize()  
    }  
    val layoutType = if (windowSize.width >= 1200.dp) {  
        NavigationSuiteType.NavigationDrawer  
    } else {  
        NavigationSuiteScaffoldDefaults.calculateFromAdaptiveInfo(  
            currentWindowAdaptiveInfo()  
        )  
    }

    NavigationSuiteScaffold(  
        navigationSuiteItems = {  
            ReplyDestination.entries.forEach {  
                item(  
                    selected = it == selectedDestination,  
                    onClick = { /\*TODO update selection\*/ },  
                    icon = {  
                        Icon(  
                            imageVector = it.icon,  
                            contentDescription = stringResource(it.labelRes)  
                        )  
                    },  
                    label = {  
                        Text(text = stringResource(it.labelRes))  
                    },  
                )  
            }  
        }  
    ) {  
        content()  
    }  
}

The navigationSuiteItems argument is a block that lets you add items using the item() function, similar to adding items in a LazyColumn. Inside the trailing lambda, this code calls the content() passed as an argument to ReplyNavigationWrapperUI().

This code first gets the window size and converts it to DP units using currentWindowSize() and LocalDensity.current, and then compares the window width to decide the layout type of the navigation UI. If the window width is at least 1200.dp, it uses NavigationSuiteType.NavigationDrawer. Otherwise, it falls back to the default calculation.

## Screen Space Use

Material 3 defines three [canonical layouts](https://m3.material.io/foundations/layout/canonical-layouts/overview) which each have configurations for compact, medium, and expanded window size classes. The [List Detail](https://m3.material.io/foundations/layout/canonical-layouts/list-detail) canonical layout is perfect for this use case, and is available in compose as ListDetailPaneScaffold.

Example: @OptIn(ExperimentalMaterial3AdaptiveApi::class)  
@Composable  
fun ReplyAppContent(  
    replyHomeUIState: ReplyHomeUIState,  
    onEmailClick: (Email) -> Unit,  
) {  
    val navigator = rememberListDetailPaneScaffoldNavigator<Long>()  
  
    ListDetailPaneScaffold(  
        directive = navigator.scaffoldDirective,  
        value = navigator.scaffoldValue,  
        listPane = {  
            ReplyListPane(replyHomeUIState, onEmailClick)  
        },  
        detailPane = {  
            ReplyDetailPane(replyHomeUIState.emails.first())  
        }  
    )  
}

This code first creates a navigator using rememberListDetailPaneNavigator(). The navigator provides some control over which pane is displayed and what content should be represented in that pane.

ListDetailPaneScaffold will show two panes when the window width size class is expanded. Otherwise, it will show one pane or the other pane based on values provided for two parameters: the scaffold directive, and the scaffold value. To get the default behavior, this code uses the scaffold directive and the scaffold value provided by the navigator.

The remaining required parameters are composable lambdas for the panes. ReplyListPane() and ReplyDetailPane() (found in ReplyListContent.kt) are used to fill in the roles of the list and detail panes, respectively. ReplyDetailPane() expects an email argument, so for now this code uses the first email from the list of emails in ReplyHomeUIState.

Now, when the user taps on an email in the list pane, we want to show in the detail pane all the replies. To keep this information we need to add a property for the selected email in the ReplyHomeUIState:

data class ReplyHomeUIState(  
    val emails : List<Email> = emptyList(),  
    val selectedEmail: Email? = null,  
    val loading: Boolean = false,  
    val error: String? = null  
)

The setSelectedEmail() function in the View Model, that is called when the user taps a list item needs to copy the UI state and record the selected email:

fun setSelectedEmail(email: Email) {  
    \_uiState.update {  
        it.copy(selectedEmail = email)  
    }  
}

The observeEmails function, when the list of emails is loaded, if the previous UI state did not have a selected email, set it to the first item:

private fun observeEmails() {  
    viewModelScope.launch {  
        emailsRepository.getAllEmails()  
            .catch { ex ->  
                \_uiState.value = ReplyHomeUIState(error = ex.message)  
            }  
            .collect { emails ->  
                val currentSelection = \_uiState.value.selectedEmail  
                \_uiState.value = ReplyHomeUIState(  
                    emails = emails,  
                    selectedEmail = currentSelection ?: emails.first()  
                )  
            }  
    }  
}

Then, the ReaplyAppContent will use the selected email if it’s available to populate the detail pane content:

ListDetailPaneScaffold(  
    // ...  
    detailPane = {  
        if (replyHomeUIState.selectedEmail != null) {  
            ReplyDetailPane(replyHomeUIState.selectedEmail)  
        }  
    }  
)

# Android App Architecture

### **Separation of Concerns**

The most important principle to follow is [separation of concerns](https://en.wikipedia.org/wiki/Separation_of_concerns). It's a common mistake to write all your code in an [Activity](https://developer.android.com/reference/android/app/Activity?authuser=1) or a [Fragment](https://developer.android.com/reference/android/app/Fragment?authuser=1). These UI-based classes should only contain logic that handles UI and operating system interactions.

### **Drive UI from data models**

Another important principle is that you should drive your UI from data models, preferably persistent models. Data models represent the data of an app. Persistent models are ideal for the following reasons:

* Your users don't lose data if the Android OS destroys your app to free up resources.
* Your app continues to work in cases when a network connection is flaky or not available.

### **Single source of truth**

When a new data type is defined in your app, you should assign a Single Source of Truth (SSOT) to it. The SSOT is the *owner* of that data, and only the SSOT can modify or mutate it. To achieve this, the SSOT exposes the data using an immutable type, and to modify the data, the SSOT exposes functions or receive events that other types can call.

This pattern brings multiple benefits:

* It centralizes all the changes to a particular type of data in one place.
* It protects the data so that other types cannot tamper with it.
* It makes changes to the data more traceable. Thus, bugs are easier to spot.

The SSOF can be a ViewModel or even the UI.

### **Unidirectional Data Flow**

In UDF, **state** flows in only one direction. The **events** that modify the data flow in the opposite direction.

In Android, state or data usually flow from the higher-scoped types of the hierarchy to the lower-scoped ones. Events are usually triggered from the lower-scoped types until they reach the SSOT for the corresponding data type. For example, application data usually flows from data sources to the UI. User events such as button presses flow from the UI to the SSOT where the application data is modified and exposed in an immutable type.

## Recommended App Architecture

Considering the common architectural principles mentioned in the previous section, each application should have at least two layers:

* The *UI layer* that displays application data on the screen.
* The *data layer* that contains the business logic of your app and exposes application data.

You can add an additional layer called the *domain layer* to simplify and reuse the interactions between the UI and data layers.

### **Modern App Architecture**

This *Modern App Architecture* encourages using the following techniques, among others:

* A reactive and layered architecture.
* Unidirectional Data Flow (UDF) in all layers of the app.
* A UI layer with state holders to manage the complexity of the UI.
* Coroutines and flows.
* Dependency injection best practices.

### **UI layer**

The role of the UI layer (or *presentation layer*) is to display the application data on the screen. Whenever the data changes, either due to user interaction (such as pressing a button) or external input (such as a network response), the UI should update to reflect the changes.

The UI layer is made up of two things:

* UI elements that render the data on the screen. You build these elements using Views or [Jetpack Compose](https://developer.android.com/jetpack/compose?authuser=1) functions.
* State holders (such as [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel?authuser=1) classes) that hold data, expose it to the UI, and handle logic.

### **Data layer**

The data layer of an app contains the *business logic*. The business logic is what gives value to your app—it's made of rules that determine how your app creates, stores, and changes data.

The data layer is made of *repositories* that each can contain zero to many *data sources*. You should create a repository class for each different type of data you handle in your app. For example, you might create a MoviesRepository class for data related to movies

Repository classes are responsible for the following tasks:

* Exposing data to the rest of the app.
* Centralizing changes to the data.
* Resolving conflicts between multiple data sources.
* Abstracting sources of data from the rest of the app.
* Containing business logic.

Each data source class should work with only one source of data (a file, a network source or database).

### **Domain layer**

The domain layer is an optional layer that sits between the UI and data layers.

The domain layer is responsible for encapsulating complex business logic, or simple business logic that is reused by multiple ViewModels. This layer is optional because not all apps will have these requirements. You should use it only when needed—for example, to handle complexity or favor reusability.

Classes in this layer are commonly called use cases.

### Manage dependencies between components

You can use either of the following desing patterns to gather the dependencies of a particular class:

* [Dependency injection (DI)](https://developer.android.com/training/dependency-injection?authuser=1): Dependency injection allows classes to define their dependencies without constructing them. At runtime, another class is responsible for providing these dependencies.
* [Service locator](https://en.wikipedia.org/wiki/Service_locator_pattern): The service locator pattern provides a registry where classes can obtain their dependencies instead of constructing them.

These patterns allow you to quickly switch between test and production implementations.

**We recommend following dependency injection patterns and using the [Hilt library](https://developer.android.com/training/dependency-injection/hilt-android?authuser=1) in Android apps.** Hilt automatically constructs objects by walking the dependency tree, provides compile-time guarantees on dependencies, and creates dependency containers for Android framework classes.

### General Best Practices

* Don’t store data in app components: Avoid designating your app’s entry points – such as activities or services – as sources of data.
* Reduce dependencies on Android classes: Your app components should be the only classes that rely on Android framework SDK APIs such as [Context](https://developer.android.com/reference/android/content/Context?authuser=1), or [Toast](https://developer.android.com/guide/topics/ui/notifiers/toasts?authuser=1).
* Create well-defined boundaries of responsibility between various modules in your app: don’t spread the code that loads data from network across multiple classes or packages. Don’t define multiple unrelated responsibilities in the same class.
* Expose as little as possible from each module.
* Focus on the unique core of your app so it stands out from other apps.
* Consider how to make each part of your app testable in isolation.
* Types are responsible for their concurrency policy.
* Persist as much relevant and fresh data as possible.

### Benefits of Architecture

* Improves the maintainability, quality and robustness of the overall app.
* It allows the app to scale. More people and more teams can contribute to the same codebase with minimal code conflicts.
* It helps with onboarding. As Architecture brings consistency to your project, new members of the team can quickly get up to speed and be more efficient in less amount of time.
* It is easier to test. A good Architecture encourages simpler types which are generally easier to test.
* Bugs can be investigated methodically with well defined processes.

## UI Layer

The role of the UI is to display the application data on the screen and also to serve as the primary point of user interaction. Whenever the data changes, the UI should update to reflect those changes. T*he UI is a visual representation of the application state as retrieved from the data layer.*

The UI layer is the pipeline that converts application data changes to a form that the UI can present and then displays it.

The UI layer must perform the following steps:

1. Consume app data and transform it into data the UI can easily render.
2. Consume UI-renderable data and transform it into UI elements for presentation to the user.
3. Consume user input events from those assembled UI elements and reflect their effects in the UI data as needed.
4. Repeat steps 1 through 3 for as long as necessary.

### UI State

The information that the app presents to the user is the UI state. The UI is the visual representation of the UI State.

#### Immutability

The UI state is immutable. For example:

data class NewsUiState(

val isSignedIn: Boolean = false,

val isPremium: Boolean = false,

val newsItems: List<NewsItemUiState> = listOf(),

val userMessages: List<Message> = listOf()

)

The benefit is that immutable objects provide guarantees regarding the state of the application at an instant in time. You should never modify the UI state in the UI directly unless the UI itself is the sole source of its data.

#### Naming conventions in this guide

UI state classes are named based on the functionality of the screen or part of the screen they describe. The convention is as follows: *functionality* + *UiState*

### Manage State with Unidirectional Data Flow

#### State Holders

The classes that are responsible for the production of UI state and contain the necessary logic for that task are called *state holders.* The typical implementation is an instance of a ViewModel, although depending on the requirements of the application, a simple class might suffice.

The **[ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel?authuser=1)** type is the recommended implementation for the management of screen-level UI state with access to the data layer. Furthermore, it survives configuration changes automatically. **ViewModel** classes define the logic to be applied to events in the app and produce updated state as a result.

The pattern where the state flows down and the events flow up is called a unidirectional data flow (UDF). The implications of this pattern for app architecture are as follows:

* The ViewModel holds and exposes the state to be consumed by the UI. The UI state is application data transformed by the ViewModel.
* The UI notifies the ViewModel of user events.
* The ViewModel handles the user actions and updates the state.
* The updated state is fed back to the UI to render.
* The above is repeated for any event that causes a mutation of state.

For navigation destinations or screens, the ViewModel works with repositories or use case classes to get data and transform it into the UI state while incorporating the effects of events that may cause mutations of the state.

#### Types of Logic

* **Business logic** is the implementation of product requirements for app data. As mentioned already. Business logic is usually placed in the domain or data layers, but never in the UI layer.
* **UI behavior logic** or **UI logic** is *how* to display state changes on the screen. Examples include obtaining the right text to show on the screen using Android [Resources](https://developer.android.com/reference/android/content/res/Resources?authuser=1), navigating to a particular screen when the user clicks a button, or displaying a user message on the screen using a [toast](https://developer.android.com/guide/topics/ui/notifiers/toasts?authuser=1) or a [snackbar](https://developer.android.com/training/snackbar?authuser=1).

#### Why use UDF?

UDF allows for the following:

* **Data consistency.** There is a single source of truth for the UI.
* **Testability.** The source of state is isolated and therefore testable independent of the UI.
* **Maintainability.** Mutation of state follows a well-defined pattern where mutations are a result of both user events and the sources of data they pull from.

### Expose UI State

By using UDF to manage the production of state, you can consider the produced state to be a stream. As a result, you should expose the UI state in an observable data holder like LiveData or StateFlow. The reason for this is so that the UI can react to any changes made in the state without having to manually pull data directly from the ViewModel. These types also have the benefit of always having the latest version of the UI state cached, which is useful for quick state restoration after configuration changes.

When the data exposed to the UI is relatively simple, it's often worth wrapping the data in a UI state type. A common way of creating a stream of UiState is by exposing a backing mutable stream as an immutable stream from the ViewModel—for example, exposing a MutableStateFlow<UiState> as a StateFlow<UiState>.

class NewsViewModel(...) : ViewModel() {

var uiState by mutableStateOf(NewsUiState())

private set

...

}

The ViewModel can then expose methods that internally mutate the state, publishing updates for the UI to consume.

class NewsViewModel(

private val repository: NewsRepository,

...

) : ViewModel() {

var uiState by mutableStateOf(NewsUiState())

private set

private var fetchJob: Job? = null

fun fetchArticles(category: String) {

fetchJob?.cancel()

fetchJob = viewModelScope.launch {

try {

val newsItems = repository.newsItemsForCategory(category)

uiState = uiState.copy(newsItems = newsItems)

} catch (ioe: IOException) {

// Handle the error and notify the UI when appropriate.

val messages = getMessagesFromThrowable(ioe)

uiState = uiState.copy(userMessages = messages)

}

}

}

}

### Consume UI State

To consume the stream of UiState objects in the UI, you use the terminal operator for the observable data type that you're using. For example, for LiveData you use the observe() method, and for Kotlin flows you use the collect() method or its variations.

When consuming observable data holders in the UI, make sure you take the lifecycle of the UI into consideration. When using LiveData, the LifecycleOwner implicitly takes care of lifecycle concerns. When using flows, it's best to handle this with the appropriate coroutine scope and the repeatOnLifecycle API:

@Composable

fun LatestNewsScreen(

viewModel: NewsViewModel = viewModel()

) {

// Show UI elements based on the viewModel.uiState

}

#### Show in-progress operations

A simple way to represent loading states in a UiState class is with a boolean field:

data class NewsUiState(

val isFetchingArticles: Boolean = false,

...

)

This flag's value represents the presence or absence of a progress bar in the UI.

@Composable

fun LatestNewsScreen(

modifier: Modifier = Modifier,

viewModel: NewsViewModel = viewModel()

) {

Box(modifier.fillMaxSize()) {

if (viewModel.uiState.isFetchingArticles) {

CircularProgressIndicator(Modifier.align(Alignment.Center))

}

// Add other UI elements. For example, the list.

}

}

#### Show errors on the screen

This is similar to showing in-progress operations. While an in-progress operation is either loading or not loading, error states might need to be modeled with data classes that host the metadata appropriate for the context of the error.

data class Message(val id: Long, val message: String)

data class NewsUiState(

val userMessages: List<Message> = listOf(),

...

)

The error messages might then be presented to the user in the form of UI elements like [snackbars](https://material.io/components/snackbars/android).

### Threading and concurrency

Any work performed in a ViewModel should be *main-safe*—safe to call from the main thread. This is because the data and domain layers are responsible for moving work to a different thread.

If a ViewModel performs long-running operations, then it is also responsible for moving that logic to a background thread. Kotlin coroutines are a great way to manage concurrent operations, and the Jetpack Architecture Components provide built-in support for them.

### Navigation

Changes in app navigation are often driven by event-like emissions. For example, after a SignInViewModel class performs a sign-in, the UiState might have an isSignedIn field set to true

### Paging

The [Paging library](https://developer.android.com/topic/libraries/architecture/paging/v3-overview?authuser=1) is consumed in the UI with a type called PagingData. Because PagingData represents and contains items that can change over time—in other words, it is not an immutable type—it should not be represented in an immutable UI state. Instead, you should expose it from the ViewModel independently in its own stream

## Data Layer

The data layer contains *application data* and *business logic*. The business logic is what gives value to your app.

This separation of concerns allows the data layer to be used on multiple screens, share information between different parts of the app, and reproduce business logic outside of the UI for unit testing.

**The data exposed by this layer should be immutable** so that it cannot be tampered with by other classes, which would risk putting its values in an inconsistent state. Immutable data can also be safely handled by multiple threads.

Following [dependency injection](https://developer.android.com/training/dependency-injection?authuser=1) best practices, the repository takes data sources as dependencies in its constructor:

class ExampleRepository(

private val exampleRemoteDataSource: ExampleRemoteDataSource, // network

private val exampleLocalDataSource: ExampleLocalDataSource // database

) { /\* ... \*/ }

### Expose APIs

Classes in the data layer generally expose functions to perform one-shot Create, Read, Update and Delete (CRUD) calls or to be notified of data changes over time. The data layer should expose the following for each of these cases:

* **One-shot operations:** The data layer should expose suspend functions in Kotlin; and for the Java programming language, the data layer should expose functions that provide a callback to notify the result of the operation, or RxJava Single, Maybe, or Completable types.
* **To be notified of data changes over time:** The data layer should expose [flows](https://developer.android.com/kotlin/flow?authuser=1) in Kotlin; and for the Java programming language, the data layer should expose a callback that emits the new data, or the RxJava Observable or Flowable type.

class ExampleRepository(

private val exampleRemoteDataSource: ExampleRemoteDataSource, // network

private val exampleLocalDataSource: ExampleLocalDataSource // database

) {

val data: Flow<Example> = ...

suspend fun modifyData(example: Example) { ... }

}

### Naming Conventions

Repository classes are named after the data that they're responsible for. The convention is as follows:

*type of data* + *Repository*.

Data source classes are named after the data they're responsible for and the source they use. The convention is as follows:

*type of data* + *type of source* + *DataSource*.

Example: NewRemoteDataSource

### Source of Truth

The source of truth can be a data source—for example, the database—or even an in-memory cache that the repository might contain. Repositories combine different data sources and solve any potential conflicts between the data sources to update the single source of truth regularly or due to a user input event.

Different repositories in your app might have different sources of truth. For example, the LoginRepository class might use its cache as the source of truth and the PaymentsRepository class might use the network data source.

### Threading

Calling data sources and repositories should be *main-safe*—safe to call from the main thread. These classes are responsible for moving the execution of their logic to the appropriate thread when performing long-running blocking operations.

Note that most data sources already provide main-safe APIs like the suspend method calls provided by [Room](https://developer.android.com/training/data-storage/room?authuser=1), [Retrofit](https://square.github.io/retrofit/) or [Ktor](https://ktor.io/). Your repository can take advantage of these APIs when they are available.

### Lifecycle

Instances of classes in the data layer remain in memory as long as they are reachable from a garbage collection root.

If a class contains in-memory data—for example, a cache—you might want to reuse the same instance of that class for a specific period of time. This is also referred to as the *lifecycle* of the class instance.

## Architecture Recomendations

Our recommended [layered architecture](https://developer.android.com/topic/architecture?gclid=CjwKCAjw6raYBhB7EiwABge5Klm_5PN8nJF0Jrb_ymrPP0JAEsbmemmGv_nsn0nBQKQtQMCBuvjehRoC7qcQAvD_BwE&%3Bgclsrc=aw.ds&authuser=1" \l "recommended-app-arch) favors separation of concerns. It drives UI from data models, complies with the single source of truth principle, and follows [unidirectional data flow](https://developer.android.com/topic/architecture?gclid=CjwKCAjw6raYBhB7EiwABge5Klm_5PN8nJF0Jrb_ymrPP0JAEsbmemmGv_nsn0nBQKQtQMCBuvjehRoC7qcQAvD_BwE&%3Bgclsrc=aw.ds&authuser=1" \l "unidirectional-data-flow) principles. Here are some best practices for layered architecture:

| **Recommendation** | **Description** |
| --- | --- |
| Use a clearly defined [data layer](https://developer.android.com/jetpack/guide/data-layer?authuser=1).  Strongly recommended | The [data layer](https://developer.android.com/jetpack/guide/data-layer?authuser=1) exposes application data to the rest of the app and contains the vast majority of business logic of your app.   * You should create [repositories](https://developer.android.com/topic/architecture/data-layer?authuser=1#architecture) even if they just contain a single data source. * In small apps, you can choose to place data layer types in a data package or module. |
| Use a clearly defined [UI layer](https://developer.android.com/jetpack/guide/ui-layer?authuser=1).  Strongly recommended | The [UI layer](https://developer.android.com/jetpack/guide/ui-layer?authuser=1) displays the application data on the screen and serves as the primary point of user interaction.   * In small apps, you can choose to place data layer types in a ui package or module.   [More UI layer best practices here](https://developer.android.com/topic/architecture/recommendations?authuser=1&continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-architecture%3Fauthuser%3D1%23article-https%3A%2F%2Fdeveloper.android.com%2Ftopic%2Farchitecture%2Frecommendations#ui-layer). |
| The [data layer](https://developer.android.com/jetpack/guide/data-layer?authuser=1) should expose application data using a repository.  Strongly recommended | Components in the UI layer such as composables, activities, or ViewModels shouldn't interact directly with a data source. Examples of data sources are:   * Databases, DataStore, SharedPreferences, Firebase APIs. * GPS location providers. * Bluetooth data providers. * Network connectivity status provider. |
| Use [coroutines and flows](https://developer.android.com/kotlin/coroutines?gclid=CjwKCAjwhNWZBhB_EiwAPzlhNtReVIBfrUFBUt6SqZz3YLezP9YEiGuBube4YSTrOF-0ovxzpNGNaRoCiYsQAvD_BwE&gclsrc=aw.ds&authuser=1).  Strongly recommended | Use [coroutines and flows](https://developer.android.com/kotlin/coroutines?gclid=CjwKCAjwhNWZBhB_EiwAPzlhNtReVIBfrUFBUt6SqZz3YLezP9YEiGuBube4YSTrOF-0ovxzpNGNaRoCiYsQAvD_BwE&gclsrc=aw.ds&authuser=1) to communicate between layers.  [More coroutines best practices here](https://developer.android.com/kotlin/coroutines/coroutines-best-practices?authuser=1). |
| Use a [domain layer](https://developer.android.com/jetpack/guide/domain-layer?authuser=1).  Recommended in big apps | Use a [domain layer](https://developer.android.com/jetpack/guide/domain-layer?authuser=1), use cases, if you need to reuse business logic that interacts with the data layer across multiple ViewModels, or you want to simplify the business logic complexity of a particular ViewModel |

1. **UI layer**

The role of the [UI layer](https://developer.android.com/topic/architecture/ui-layer?authuser=1) is to display the application data on the screen and serve as the primary point of user interaction. Here are some best practices for the UI layer:

| **Recommendation** | **Description** |
| --- | --- |
| Follow [Unidirectional Data Flow (UDF)](https://developer.android.com/jetpack/compose/architecture?authuser=1#udf).  Strongly recommended | Follow [Unidirectional Data Flow (UDF)](https://developer.android.com/jetpack/compose/architecture?authuser=1#udf) principles, where ViewModels expose UI state using the observer pattern and receive actions from the UI through method calls. |
| Use [AAC ViewModels](https://developer.android.com/topic/libraries/architecture/viewmodel?authuser=1) if their benefits apply to your app.  Strongly recommended | Use [AAC ViewModels](https://developer.android.com/topic/libraries/architecture/viewmodel?authuser=1) to [handle business logic](https://developer.android.com/jetpack/guide/ui-layer?authuser=1#logic-types), and fetch application data to expose UI state to the UI (Compose or Android Views).  See more [ViewModel best practices here.](https://developer.android.com/topic/architecture/recommendations?authuser=1&continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-architecture%3Fauthuser%3D1%23article-https%3A%2F%2Fdeveloper.android.com%2Ftopic%2Farchitecture%2Frecommendations#viewmodel)  See the [benefits of ViewModels here.](https://developer.android.com/topic/architecture/ui-layer/stateholders?authuser=1#viewmodel-as) |
| Use lifecycle-aware UI state collection.  Strongly recommended | Collect UI state from the UI using the appropriate lifecycle-aware coroutine builder: [repeatOnLifecycle](https://developer.android.com/reference/kotlin/androidx/lifecycle/package-summary?authuser=1" \l "(androidx.lifecycle.Lifecycle).repeatOnLifecycle(androidx.lifecycle.Lifecycle.State,kotlin.coroutines.SuspendFunction1)) in the View system and [collectAsStateWithLifecycle](https://developer.android.com/reference/kotlin/androidx/lifecycle/compose/package-summary?authuser=1#(kotlinx.coroutines.flow.StateFlow).collectAsStateWithLifecycle(androidx.lifecycle.LifecycleOwner,androidx.lifecycle.Lifecycle.State,kotlin.coroutines.CoroutineContext)) in Jetpack Compose.  Read more about[repeatOnLifecycle](https://medium.com/androiddevelopers/a-safer-way-to-collect-flows-from-android-uis-23080b1f8bda).  Read more about about[collectAsStateWithLifecycle](https://medium.com/androiddevelopers/consuming-flows-safely-in-jetpack-compose-cde014d0d5a3). |
| Do not send events from the ViewModel to the UI.  Strongly recommended | Process the event immediately in the ViewModel and cause a state update with the result of handling the event. More about [UI events here](https://developer.android.com/topic/architecture/ui-layer/events?authuser=1#handle-viewmodel-events). |
| Use a single-activity application.  Recommended | Use [Navigation Fragments](https://developer.android.com/guide/navigation?authuser=1) or [Navigation Compose](https://developer.android.com/jetpack/compose/navigation?authuser=1) to navigate between screens and deep link to your app if your app has more than one screen. |
| Use [Jetpack Compose](https://developer.android.com/jetpack/compose?authuser=1).  Recommended | Use [Jetpack Compose](https://developer.android.com/jetpack/compose?authuser=1) to build new apps for phones, tablets and foldables and Wear OS. |

1. **ViewModel**

[ViewModels](https://developer.android.com/topic/architecture/ui-layer/stateholders?authuser=1#business-logic) are responsible for providing the UI state and access to the data layer. Here are some best practices for ViewModels:

| **Recommendation** | **Description** |
| --- | --- |
| ViewModels should be agnostic of the Android lifecycle.  Strongly recommended | ViewModels shouldn't hold a reference to any Lifecycle-related type. Don't pass Activity, Fragment, Context or Resources as a dependency. If something needs a Context in the ViewModel, you should strongly evaluate if that is in the right layer. |
| Use [coroutines and flows](https://developer.android.com/kotlin/coroutines?gclid=CjwKCAjwhNWZBhB_EiwAPzlhNtReVIBfrUFBUt6SqZz3YLezP9YEiGuBube4YSTrOF-0ovxzpNGNaRoCiYsQAvD_BwE&gclsrc=aw.ds&authuser=1).  Strongly recommended | The ViewModel interacts with the data or domain layers using:   * Kotlin flows for receiving application data, * suspend functions to perform actions using [viewModelScope](https://developer.android.com/topic/libraries/architecture/coroutines?authuser=1" \l "viewmodelscope). |
| Use ViewModels at screen level.  Strongly recommended | Do not use ViewModels in reusable pieces of UI. You should use ViewModels in:   * Screen-level composables, * Activities/Fragments in Views, * Destinations or graphs when using [Jetpack Navigation](https://developer.android.com/guide/navigation?authuser=1). |
| Use [plain state holder classes](https://developer.android.com/topic/architecture/ui-layer/stateholders?authuser=1#ui-logic) in reusable UI components.  Strongly recommended | Use [plain state holder classes](https://developer.android.com/topic/architecture/ui-layer/stateholders?authuser=1#ui-logic) for handling complexity in reusable UI components. By doing this, the state can be hoisted and controlled externally. |
| Do not use [AndroidViewModel](https://developer.android.com/reference/androidx/lifecycle/AndroidViewModel?authuser=1).  Recommended | Use the [ViewModel](https://developer.android.com/reference/androidx/lifecycle/ViewModel?authuser=1) class, not [AndroidViewModel](https://developer.android.com/reference/androidx/lifecycle/AndroidViewModel?authuser=1). The Application class shouldn't be used in the ViewModel. Instead, move the dependency to the UI or the data layer. |
| Expose a UI state.  Recommended | ViewModels should expose data to the UI through a single property called uiState. If the UI shows multiple, unrelated pieces of data, the VM can [expose multiple UI state properties](https://developer.android.com/jetpack/guide/ui-layer?authuser=1#additional-considerations).   * You should make uiState a StateFlow. * You should create the uiState using the [stateIn](https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/state-in.html) operator with the [WhileSubscribed(5000)](https://medium.com/androiddevelopers/migrating-from-livedata-to-kotlins-flow-379292f419fb) policy [(example)](https://github.com/android/compose-samples/blob/main/JetNews/app/src/main/java/com/example/jetnews/ui/interests/InterestsViewModel.kt#L56) if the data comes as a stream of data from other layers of the hierarchy. * For simpler cases with no streams of data coming from the data layer, it's acceptable to use a MutableStateFlow exposed as an immutable StateFlow [(example)](https://github.com/android/compose-samples/blob/main/Jetcaster/app/src/main/java/com/example/jetcaster/ui/home/category/PodcastCategoryViewModel.kt#L37). * You can choose to have the ${Screen}UiState as a data class that can contain data, errors and loading signals. This class could also be a sealed class if the different states are exclusive. |

1. **Lifecycle**

The following are some best practices for working with the [Android lifecycle](https://developer.android.com/guide/components/activities/activity-lifecycle?authuser=1):

| **Recommendation** | **Description** |
| --- | --- |
| Do not override lifecycle methods in Activities or Fragments.  Strongly recommended | Do not override lifecycle methods such as onResume in Activities or Fragments. Use [LifecycleObserver](https://developer.android.com/reference/androidx/lifecycle/LifecycleObserver?authuser=1) instead. If the app needs to perform work when the lifecycle reaches a certain Lifecycle.State, use the [repeatOnLifecycle](https://developer.android.com/reference/kotlin/androidx/lifecycle/package-summary?authuser=1" \l "(androidx.lifecycle.Lifecycle).repeatOnLifecycle(androidx.lifecycle.Lifecycle.State,kotlin.coroutines.SuspendFunction1)) API. |

**Handle dependencies**

There are several best practices you should observe when managing dependencies between components:

| **Recommendation** | **Description** |
| --- | --- |
| Use [dependency injection](https://developer.android.com/training/dependency-injection?authuser=1).  Strongly recommended | Use [dependency injection](https://developer.android.com/training/dependency-injection?authuser=1) best practices, mainly [constructor injection](https://developer.android.com/training/dependency-injection?authuser=1#what-is-di) when possible. |
| Scope to a component when necessary.  Strongly recommended | Scope to a [dependency container](https://developer.android.com/training/dependency-injection/manual?authuser=1#dependencies-container) when the type contains mutable data that needs to be shared or the type is expensive to initialize and is widely used in the app. |
| Use [Hilt](https://developer.android.com/training/dependency-injection/hilt-android?authuser=1).  Recommended | Use [Hilt](https://developer.android.com/training/dependency-injection/hilt-android?authuser=1) or [manual dependency injection](https://developer.android.com/training/dependency-injection/manual?authuser=1) in simple apps. Use [Hilt](https://developer.android.com/training/dependency-injection/hilt-android?authuser=1) if your project is complex enough. For example, if you have:   * Multiple screens with ViewModels—integration * WorkManager usage—integration * Advance usage of Navigation, such as ViewModels scoped to the nav graph—integration. |

**Testing**

The following are some best practices for [testing](https://developer.android.com/training/testing/fundamentals?authuser=1):

| **Recommendation** | **Description** |
| --- | --- |
| [Know what to test](https://developer.android.com/training/testing/fundamentals/what-to-test?authuser=1).  Strongly recommended | Unless the project is roughly as simple as a hello world app, you should test it, at minimum with:   * Unit test ViewModels, including Flows. * Unit test data layer entities. That is, repositories and data sources. * UI navigation tests that are useful as regression tests in CI. |
| Prefer fakes to mocks.  Strongly recommended | Read more in the [Use test doubles in Android documentation](https://developer.android.com/training/testing/fundamentals/test-doubles?authuser=1). |
| Test StateFlows.  Strongly recommended | When testing StateFlow:   * [Assert on the value property](https://developer.android.com/kotlin/flow/test?authuser=1#stateflows) whenever possible * You should [create a collectJob](https://developer.android.com/kotlin/flow/test?authuser=1#statein) if using WhileSubscribed |

**Models**

You should observe these best practices when developing models in your apps:

| **Recommendation** | **Description** |
| --- | --- |
| Create a model per layer in complex apps.  Recommended | In complex apps, create new models in different layers or components when it makes sense. Consider the following examples:   * A remote data source can map the model that it receives through the network to a simpler class with just the data the app needs * Repositories can map DAO models to simpler data classes with just the information the UI layer needs. * ViewModel can include data layer models in UiState classes. |

1. **Naming conventions**

When naming your codebase, you should be aware of the following best practices:

| **Recommendation** | **Description** |
| --- | --- |
| Naming methods.  Optional | Methods should be a verb phrase. For example, makePayment(). |
| Naming properties.  Optional | Properties should be a noun phrase. For example, inProgressTopicSelection. |
| Naming streams of data.  Optional | When a class exposes a Flow stream, LiveData, or any other stream, the naming convention is get{model}Stream(). For example, getAuthorStream(): Flow<Author> If the function returns a list of models the model name should be in the plural: getAuthorsStream(): Flow<List<Author>> |
| Naming interfaces implementations.  Optional | Names for the implementations of interfaces should be meaningful. Have Default as the prefix if a better name cannot be found. For example, for a NewsRepository interface, you could have an OfflineFirstNewsRepository, or InMemoryNewsRepository. If you can find no good name, then use DefaultNewsRepository. Fake implementations should be prefixed with Fake, as in FakeAuthorsRepository. |

# Android Accessibility

Some advices to create apps with accessibility:

### Increase Text Visibility

We recommend the *color contrast*

* If the text is smaller than 18pt, or if the text is bold and smaller than 14pt, set the color contrast ratio to at least 4.5:1.
* For all other text, set the color contrast ratio to at least 3:1.

### Use large, simle controls

Your app's UI is easier to use if its controls are easier to see and tap. We recommend that each interactive UI element have a focusable area, or *touch target size*, of at least 48dpx48dp. Larger is even better.

For a given UI element to have a large enough touch target size, the following conditions should **both** be true:

* The sum of the values of [android:paddingLeft](https://developer.android.com/reference/android/view/View?authuser=1" \l "attr_android:paddingLeft), [android:minWidth](https://developer.android.com/reference/android/view/View?authuser=1" \l "attr_android:minWidth), and [android:paddingRight](https://developer.android.com/reference/android/view/View?authuser=1" \l "attr_android:paddingRight) is greater than or equal to 48dp.
* The sum of the values of [android:paddingTop](https://developer.android.com/reference/android/view/View?authuser=1" \l "attr_android:paddingTop), [android:minHeight](https://developer.android.com/reference/android/view/View?authuser=1" \l "attr_android:minHeight), and [android:paddingBottom](https://developer.android.com/reference/android/view/View?authuser=1" \l "attr_android:paddingBottom) is greater than or equal to 48dp.

### Describe each UI element

For each UI element in your app, include a description that describes the element's purpose. In most cases, you include this description in the element's contentDescription attribute, as shown in the following code snippet:

* Don't include the type of UI element in the content description. Screen readers automatically announce both the element's type and description.
* Each description must be unique.
* If your app's minSdkVersion is 16 or higher, you can set the [android:importantForAccessibility](https://developer.android.com/reference/android/view/View?authuser=1" \l "attr_android:importantForAccessibility) attribute to "no" for graphical elements that are only used for decorative effect.

## Test your App Accessibility

For the best results, use all of the approaches described in this document:

* **Manual testing:** interact with your app using Android accessibility services.
* **Testing using analysis tools:** use tools to discover opportunities to improve your app's accessibility.
* **Automated testing:** turn on accessibility testing in Espresso and Robolectric.
* **User testing:** get feedback from people who interact with your app.

**TalkBack**

TalkBack is Android's built-in screen reader. When TalkBack is on, users can interact with their Android-powered device without seeing the screen. Users with visual impairments might rely on TalkBack to use your app.

**Turn on TalkBack**

1. Open your device's Settings app.
2. Navigate to **Accessibility** and select **TalkBack**.
3. At the top of the TalkBack screen, press **On/Off** to turn on TalkBack.
4. In the confirmation dialog, select **OK** to confirm permissions.

**Explore your app with TalkBack**

Once TalkBack is on, there are two common ways to navigate:

* **Linear navigation:** quickly swipe right or left to navigate through screen elements in sequence. Double-tap anywhere to select the current screen element.
* **Explore by tapping:** drag your finger over the screen to hear what's under your finger. Double-tap anywhere to select the current element.

1. **Testing using analysis tools**

Analysis tools can uncover opportunities to improve accessibility that you might miss with manual testing.

**Compose UI Check**

Activate Compose UI Check mode  on a Compose Preview to enable Android Studio to automatically audit your Compose UI for accessibility issues. Android Studio checks that your UI works across different screen sizes by highlighting issues such as text stretched on large screens or low color contrast in the problems panel.

**Accessibility Scanner**

The [Accessibility Scanner](https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&authuser=1) app scans your screen and suggests ways to improve the accessibility of your app. Accessibility Scanner uses the [Accessibility Test Framework](https://github.com/google/Accessibility-Test-Framework-for-Android) and provides specific suggestions after looking at content labels, clickable items, contrast, and more.

The Android Accessibility Test Framework is integrated in Android Studio to help you find accessibility issues in your layouts. To launch the panel, click the error report button [!](https://developer.android.com/static/studio/images/buttons/toggle-issue-panel-button.png?authuser=1) in the Layout Editor.

**UI Automator Viewer**

The uiautomatorviewer tool provides a convenient GUI to scan and analyze the UI components currently displayed on an Android-powered device. You can use UI Automator to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device. This information lets you create more fine-grained tests, for example by creating a UI selector that matches a specific visible property. The tool is located in the tools directory of the Android SDK.

**Lint**

Android Studio shows lint warnings for various accessibility issues and provides links to the relevant places in your source code. In the following example, an image is missing a contentDescription attribute.

### **Automated testing**

The Android platform supports several testing frameworks, such as Espresso, which lets you create and run automated tests that evaluate the accessibility of your app.

**Espresso**

[Espresso](https://developer.android.com/training/testing/espresso?authuser=1) is an Android testing library designed to make UI testing fast and easy. It lets you interact with UI components under test in your app and assert that certain behaviors occur or that specific conditions are met.