



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

Day 12

Jetpack Compose UI

Entering the World of Jetpack Compose UI (Why and How?) 🚀



Esmaeeil Moradi

@esmaeeil-moradi700

Day 12

Fundamentals of Modern UI Development: Entering the World of Jetpack Compose UI (Why and How?)

Greetings to all colleagues, engineers, and Android architects – both those who have wrestled with XML for years and those just starting their journey!

Today, on the twelfth day of our challenge, we're stepping into a new and exciting world in Android UI development: Jetpack Compose. If, like me, you've spent years grappling with the complexities of the traditional View System, lengthy XML files, the hassles of findViewById (or even ButterKnife and ViewBinding), frustrating State management, and Lifecycle issues in UI updates, then the name Compose likely sparks both curiosity and perhaps a bit of caution. As someone who has experienced all these eras and has now embraced Compose with open arms, I want to use this article to explain the fundamentals of this revolutionary toolkit in a way that clarifies "why" this shift is so important and "how" we can take our first solid steps into this declarative world. Ready for a fundamental mindset shift?



Esmaeel Moradi

@esmaeel-moradi700



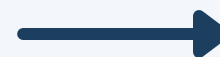
Introduction: "Why Compose?" – Farewell to the Past, Hello to the Future of Android UI.

Let's be honest with ourselves. Building UIs with the old View System, despite its power, sometimes felt like navigating a minefield. Voluminous XML files, the constant need to reference Views from Kotlin/Java code, complexities in creating custom components and reusing code, and most importantly, State management that could turn into a nightmare as applications grew.

Jetpack Compose is Google's modern, declarative, and Kotlin-first answer to all these challenges. But what does "declarative" actually mean? Simply put, in the old imperative approach, we told the system how to build and change the UI step-by-step (e.g., "Find a TextView, set its text to this, change its color to that"). In Compose, however, we simply describe what the UI should look like at any given moment for any given state, and Compose itself takes responsibility for efficiently implementing and updating it. This is a major paradigm shift that not only makes UI coding simpler and more readable but also significantly boosts our productivity and joy as developers. Compose is a true game-changer!



Esmaeel Moradi
@esmaeel-moradi700



Core Concepts of Compose (The New Building Blocks):

- **Composable Functions (@Composable):**

These are the fundamental building units of UI in Compose. Any function annotated with @Composable can describe a piece of your UI. Key point: These functions don't return View Objects; they provide a description of the UI. Ideally, Composable functions should be pure functions and idempotent (i.e., given the same input, they always produce the same output, and calling them multiple times doesn't change the result).

- **Declarative Thinking in Action:**

Again, I emphasize, you just say, "With this state, the UI should look like this." Compose figures out how to get from the previous state to the new one.



Esmaeel Moradi

@esmaeel-moradi700

• Basic Composables:

Let's meet some of the most common ones:

- `Text("Hello World!")`: For displaying text.
- `Button(onClick = { /* do something */ }) { Text("Click Me") }`: For buttons.
- `Image(painter = painterResource(id = R.drawable.my_image), contentDescription = "Image description")`: For displaying images.
- `Icon(imageVector = Icons.Filled.Favorite, contentDescription = "Favorite icon")`: For displaying vector icons.
- `Surface { Text("Text on a surface") }`: A surface with elevation, background color, etc. (akin to `CardView` or a `View`'s background).

```
// Simple Composable Example
@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}

@Composable
fun SimpleButton() {
    Button(onClick = { Log.d("Compose", "Button Clicked") }) {
        Text("A Simple Button")
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700

• **Modifier – The All-Purpose Spice:**

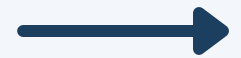
Almost every Composable accepts a modifier: Modifier parameter. Modifiers allow you to chain changes to a Composable's appearance and behavior: padding, size, background color, shape, click handling, scrolling, and dozens of other things. This is one of Compose's most powerful concepts

```
// Modifier Example
@Composable
fun StyledText(message: String) {
    Text(
        text = message,
        modifier = Modifier
            .padding(16.dp) // Padding around
            .background(Color.LightGray) // Background color
            .clickable { /* do something */ } // Clickable
    )
}
```



Esmaeel Moradi

@esmaeel-moradi700



- **Basic Layout Composables:**

For arranging Composables:

- Column { ... }: Arranges children vertically.
- Row { ... }: Arranges children horizontally.
- Box { ... }: Stacks children on top of each other (like `FrameLayout`).
- Within Column and Row, you can control child arrangement and alignment using parameters like `verticalArrangement`, `horizontalArrangement`, `verticalAlignment`, and `horizontalAlignment`.
- `Spacer(modifier = Modifier.height(8.dp))`: For creating space between items.

```
// Basic Layout Example
@Composable
fun MyLayout() {
    Column(
        modifier = Modifier.padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp) // Space between items
    ) {
        Text("Title")
        Row(horizontalArrangement = Arrangement.SpaceAround) {
            Button(onClick = {}) { Text("Button 1") }
            Button(onClick = {}) { Text("Button 2") }
        }
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700

Managing State in Compose (The "Aha!" Moment for Many):



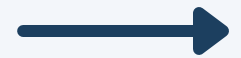
This is where Compose truly shines. In UI, State refers to any value that can change over time and affect how the UI is displayed (e.g., the text in a TextField, the status of a CheckBox, or a list of items).

- **remember – Composables' Short-Term Memory:** When a Composable is called again due to its parent's state change (Recomposition), all its internal variables are lost. `remember { ... }` allows you to "remember" a value across different Recompositions.
- **mutableStateOf – Observable and Mutable State:** For Compose to know when to update the UI, your state needs to be "observable." `mutableStateOf(initialValue)` returns a `MutableState` object. Whenever its `.value` changes, Compose automatically Recomposes all Composables that read this state.
- **Recomposition – The Magic of Compose:** This is the intelligent UI update process. When a `MutableState` changes, Compose only re-invokes (Recomposes) the Composables that directly read that State (or depend on it). This is done very efficiently and, contrary to initial fears, doesn't lead to performance issues if used correctly.



Esmaeel Moradi

@esmaeel-moradi700



```
// Simple State Example: A Counter
@Composable
fun Counter() {
    // 'count' is a state that Compose can observe
    // 'remember' ensures 'count' survives recompositions
    var count by remember { mutableStateOf(0) }

    Column(horizontalAlignment = Alignment.CenterHorizontally) {
        Text("Counter: $count")
        Button(onClick = { count++ }) { // Changing count triggers recomposition
            Text("Increment")
        }
    }
}
```

- **State Hoisting:**

A very important pattern in Compose. Instead of each Composable having its own internal state, it's better to move (or "hoist") the state to their common parent and pass it down to children via parameters, and notify changes from children to parent via callbacks (lambda functions). This makes child Composables stateless, and thus much more reusable and testable. (This concept might be slightly advanced for basics, but mentioning it as an early best practice is important).



Esmaeel Moradi
@esmaeel-moradi700

Lifecycle in the World of Compose (A Simple Look at the Basics):



The lifecycle in Compose is different from Activities/Fragments. Instead of onCreate, onStart, etc., we deal with:

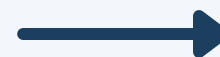
- **Entering Composition:** When a Composable is first displayed.
- **Recomposition:** When a Composable's dependent state changes and it needs to be updated.
- **Leaving Composition:** When a Composable is no longer displayed (e.g., due to an if condition or removal from a list).

To run code that needs to accompany a Composable's lifecycle (e.g., loading data from a network when it appears, or releasing resources when it's gone), we use Composables like LaunchedEffect, DisposableEffect, etc. For basics, a mention of LaunchedEffect for an initial task suffices:



Esmaeel Moradi

@esmaeel-moradi700

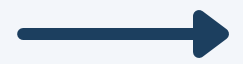


```
// LaunchedEffect Example
@Composable
fun MyScreen(itemId: String) {
    // This block runs when MyScreen enters composition
    // and when itemId changes.
    LaunchedEffect(key1 = itemId) {
        Log.d("MyScreen", "Loading data for item: $itemId")
        // viewModel.loadData(itemId)
    }
    Text("Information for item $itemId")
}
```



Esmaeel Moradi
@esmaeel-moradi700

Setting up Compose in a Project:



- Dependencies in build.gradle:

```
// build.gradle (Module)
// ...
android {
    buildFeatures {
        compose true // Enable Compose
    }
    composeOptions {
        kotlinCompilerExtensionVersion compose_compiler_version // e.g., "1.5.1"
    }
}

dependencies {
    def compose_bom_version = "2024.05.00" // Check for latest
    implementation platform("androidx.compose:compose-bom:$compose_bom_version")
    implementation "androidx.compose.ui:ui"
    implementation "androidx.compose.ui:ui-graphics"
    implementation "androidx.compose.ui:ui-tooling-preview"
    implementation "androidx.compose.material3:material3"
    // ... other compose dependencies
}
```

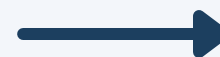
- Usage in Activity:

```
// MainActivity.kt
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent { // This is where your Compose UI goes
            MyApplicationTheme { // Your app's theme
                Surface(modifier = Modifier.fillMaxSize()) {
                    Greeting("Android with Compose")
                }
            }
        }
    }
}
```



Esmaeel Moradi

@esmaeeil-moradi700



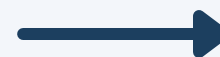
"Thinking in Compose" (A Mindset Shift for Veterans):

For those of us who've worked with XML and the View System for years, migrating to Compose requires a mental shift:

- **From Imperative to Declarative:** We no longer command the UI; we describe it.
- **Embrace Recomposition:** Don't fear Composable functions being called multiple times; it's the nature of Compose and it's optimized.
- **Immutable Data:** Work with immutable data as much as possible and manage state changes controllably.
- **Unidirectional Data Flow (UDF):** State flows down, events flow up. (A concept hinted at in State Hoisting).



Esmaeel Moradi
@esmaeel-moradi700



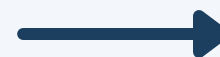
Benefits of Compose (From an Experienced Specialist's POV):

- **Less Code, More Readability:** Goodbye to verbose XML and findViewById.
- **Kotlin Power:** Full leverage of Kotlin language features (top-level functions, lambdas, Coroutines).
- **Amazing Tooling:** Instant @Preview and Live Edit in Android Studio revolutionize the development experience.
- **Improved Testability:** Declarative UIs and stateless Composables are easier to test.
- **Unification:** Compose isn't just for mobile (Compose Multiplatform for Desktop, Web, and iOS).



Esmaeel Moradi

@esmaeel-moradi700



Conclusion: The Future is Declarative!

Jetpack Compose isn't just a new UI library; it's a fundamental shift in how we build user interfaces in Android. It might have a learning curve initially, but its long-term benefits in terms of development speed, code quality, and programming enjoyment are well worth the investment. As someone who has witnessed several generations of Android UI tools, I say with confidence: the future of UI in Android is declarative and with Compose. Embrace this change and enjoy the learning journey!



Esmaeel Moradi

@esmaeel-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



LEAVE A COMMENT BELOW



Esmaeeil Moradi

@esmaeeil-moradi700