



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

Day 14

Complex Layouts in Compose

Mastering ConstraintLayout and Diving into Custom Layouts 🚀



Esmaeeil Moradi

@esmaeeil-moradi700

Day 14



Architecting Complex Layouts in Compose:

Mastering ConstraintLayout and Diving into Custom Layouts 🚀

Greetings again to all professional colleagues, Android engineers, and architects!

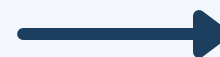
In the previous days, we became acquainted with the basics of Jetpack Compose, including fundamental layouts like Column, Row, and Box, as well as the principles of state management. These tools are sufficient and efficient for building many UIs. However, when we encounter more complex real-world designs – screens with numerous elements having intricate relationships, the need to optimize performance by reducing hierarchy depth, or even the desire to create a truly unique layout – we realize we need more powerful tools.

Today, we move beyond simple columns and rows to delve into architecting complex layouts in Compose. First, we'll tackle ConstraintLayout in Compose, and then, the most exciting part: we'll learn how to build our very own Custom Layouts! Think Column and Row are the end of the line? Prepare to discover the true power of Compose's layout system – a power that allows you to build layouts perhaps even the framework authors never envisioned!



Esmaeel Moradi

@esmaeeil-moradi700



Introduction: Beyond Columns and Rows – Architecting Complex UIs in Compose.

Building complex yet performant UIs has always been a core challenge in Android development. The old View System took a big step forward with its ConstraintLayout. The good news is that Compose not only offers a powerful implementation of ConstraintLayout but also opens the door wider for innovation and optimization by allowing us to build custom layouts. In this article, we will dissect both of these advanced approaches.



Esmaeel Moradi
@esmaeel-moradi700



Mastering Built-in Layouts (Advanced Techniques):

- **BoxWithConstraints**: Sometimes, layout decisions or even which elements to display need to be based on the available space. **BoxWithConstraints** is designed precisely for this. This Composable gives you access to the **Constraints** passed down from the parent, allowing you to make decisions based on **minWidth**, **maxWidth**, **minHeight**, and **maxHeight**.

```
@Composable
fun ResponsiveContent() {
    BoxWithConstraints {
        // 'this' is BoxWithConstraintsScope
        val boxWidth = this.maxWidth
        if (boxWidth < 400.dp) {
            // Show smaller content
            Text("Small content for width ${boxWidth}")
        } else {
            // Show larger content
            Text("Large content for width ${boxWidth}")
        }
    }
}
```



Esmaeel Moradi
@esmaeel-moradi700

- **ConstraintLayout in Compose (Deep Dive):** For managing complex relationships between many Composables or when you want a Flat UI Hierarchy (which is generally better for performance), ConstraintLayout in Compose is an incredibly powerful tool. —————>

1. **Why and When?** Pros: Easily manages complex relationships (relative to parent edges or other elements), creates flatter hierarchies, provides helper tools like Barriers and Guidelines.
2. **Cons:** Steeper initial learning curve for its specific syntax, slightly more verbose than basic layouts for simple arrangements.

- **Setup:**

Add the dependency to build.gradle:

```
// build.gradle (Module)
dependencies {
    implementation "androidx.constraintlayout:constraintlayout-compose:1"
}
```



Esmaeel Moradi
@esmaeel-moradi700



- **Key Concepts:**

1. **Refs:** Create references to your Composables using `createRefs()` (for multiple elements) or `createRefFor()` (for one).
2. **Modifier.constrainAs(ref) { ... }:** Define constraints for each element within this Modifier using `linkTo()`.
3. **linkTo():** Connects an element's edges (start, end, top, bottom) to the parent's edges (parent) or another element's edges (otherRef). You can also define bias.
4. **Barrier:** An invisible line whose position is determined by the edge of multiple other elements (e.g., a barrier below two Text elements of different heights).
5. **Guideline:** An invisible horizontal or vertical guide line at a specific percentage or distance from the parent's edges.
6. **Chain:** Arranges multiple elements horizontally or vertically with different styles (packed, spread, spread_inside, weighted).



Esmaeel Moradi
@esmaeel-moradi700

```
// ConstraintLayout Example (Simple Profile Header)
@Composable
fun ProfileHeader() {
    ConstraintLayout(modifier = Modifier.fillMaxWidth().padding(16.dp)) {
        // Create references for the composables
        val (avatar, name, bio, followButton) = createRefs()

        Image(
            /* painter = ..., contentDescription = ... */
            modifier = Modifier.constrainAs(avatar) {
                top.linkTo(parent.top)
                start.linkTo(parent.start)
                width = Dimension.value(48.dp)
                height = Dimension.ratio("1:1") // Aspect ratio
            }
        )
        Text("User Name", modifier = Modifier.constrainAs(name) {
            top.linkTo(avatar.top)
            start.linkTo(avatar.end, margin = 8.dp)
            end.linkTo(followButton.start, margin = 8.dp)
            width = Dimension.fillToConstraints // Fill available space
        })
        Text("User's short bio...", modifier = Modifier.constrainAs(bio) {
            top.linkTo(name.bottom, margin = 4.dp)
            start.linkTo(name.start)
            end.linkTo(name.end)
            width = Dimension.fillToConstraints
        })
        Button(onClick = { /*TODO*/ }, modifier = Modifier.constrainAs(followButton) {
            top.linkTo(avatar.top)
            end.linkTo(parent.end)
        }) { Text("Follow") }
    }
}
```



Esmaeel Moradi

@esmaeel-moradi700

- **Performance Considerations:**

For deep hierarchies, *ConstraintLayout* can improve performance by flattening the view tree. However, for simple linear layouts, *Column* and *Row* are generally more efficient.



- ***SubcomposeLayout* (A Glimpse of Hidden Power):** This is a lower-level layout primitive that allows you to compose a part of the UI during the measurement phase to make layout decisions based on its size. *BoxWithConstraints* uses *SubcomposeLayout* under the hood. It's a very powerful tool but typically used for advanced cases.
- **Intrinsic Measurements:** Sometimes you want to know a child's preferred size based on its content before the final measurement, to achieve better alignment. For instance, making all buttons in a *Row* the same height as the tallest button. You can use *Modifier.height(IntrinsicSize.Min)* on the *Row* and then *Modifier.fillMaxHeight()* on the children. *IntrinsicWidth* works similarly.



Esmaeel Moradi

@esmaeel-moradi700

Custom Layouts – Unleashing True Flexibility:



This is where you gain complete control!

- **"Why" Build Custom Layouts?**
 - When the desired layout cannot be easily or efficiently achieved by combining standard layouts.
 - To optimize performance for specific scenarios (e.g., reducing layers or measurement passes).
 - To create entirely unique and creative UI designs that set your app apart.
- **The Layout Process in Compose (Simplified but Accurate):**
 - a. Composition: The UI tree of Composables is built.
 - b. Measurement: The system traverses down the tree, asking each layout: "Given these Constraints, how much space do you need?" Each layout asks its children and determines its own size.
 - c. Placement: The system traverses down again, telling each layout: "Now that your size is determined, where will you place your children?" A Custom Layout allows you to write the logic for the Measurement and Placement phases yourself.
- The Layout Composable – The Core Building Block: The heart of custom layout creation is the Layout Composable function:



Esmaeel Moradi

@esmaeel-moradi700



```
@Composable
fun CustomLayout(
    modifier: Modifier = Modifier,
    content: @Composable () -> Unit // Children Composables
) {
    Layout(
        content = content,
        modifier = modifier,
        measurePolicy = // Your custom logic goes here!
    )
}
```

The most crucial part is the measurePolicy parameter, which takes an object implementing the MeasurePolicy interface.



Esmaeel Moradi
@esmaeeil-moradi700

• The MeasurePolicy Interface (The Heart of a Custom Layout):

This interface has one key function you need to implement: measure.

```
// Inside the 'measure' function of your MeasurePolicy
fun measure(
    measurables: List<Measurable>, // List of children to measure
    constraints: Constraints         // Constraints from the parent
): MeasureResult {
    // 1. Measure children (get Placeables)
    val placeables = measurables.map { measurable ->
        // Decide child constraints based on parent constraints & logic
        // E.g., measure child with unbounded height:
        // measurable.measure(constraints.copy(maxHeight = Constraints.Infinity)
        measurable.measure(childConstraints) // result is a Placeable
    }

    // 2. Determine the size of this custom layout
    val layoutWidth = /* Calculate based on placeables & logic */
    val layoutHeight = /* Calculate based on placeables & logic */

    // 3. Return the size and placement logic using the layout() function
    return layout(layoutWidth, layoutHeight) {
        // This lambda scope is for placing children
        // 4. Place children using placeable.placeRelative(x, y)
        placeables.forEach { placeable ->
            // Calculate x, y position for each placeable based on your logic
            val xPosition = /* ... */
            val yPosition = /* ... */
            placeable.placeRelative(xPosition, yPosition)
        }
    }
}
```



Esmaeel Moradi

@esmaeel-moradi700



- **Measurement Phase:**

1. You receive a list of *Measurables* (representing the children) and the parent's *Constraints*.
2. For each *Measurable*, you call its *measure()* method, passing the appropriate *Constraints* you decide for that child. The result is a *Placeable* which knows the child's measured size.
3. Based on the sizes of the *Placeables* and your layout logic (e.g., arranging in a row, wrapping to the next line), you calculate the overall size (width, height) of your custom layout.

- **Placement Phase:**

1. Inside the lambda passed to the *layout(width, height)* function, you have access to the list of *Placeables*.
2. For each *Placeable*, you call *placeable.placeRelative(x, y)* (or *placeAbsolute*) to determine its exact position within your custom layout's coordinates



Esmaeel Moradi

@esmaeel-moradi700



Step-by-Step Custom Layout Example (Simple FlowRow):

- Let's build a simple FlowRow that arranges items in a row and wraps to the next line if space runs out.

```
fun SimpleFlowRow(
    modifier: Modifier = Modifier,
    spacing: Dp = 8.dp, // Spacing between items
    content: @Composable () -> Unit
) {
    Layout(
        content = content,
        modifier = modifier
    ) { measurables, constraints ->
        val spacingPx = spacing.roundToPx()
        // Store placeable + x position per row
        val groupPositions =
            mutableListOf<MutableList<Pair<Placeable, Int>>>()
        var currentGroup = mutableListOf<Pair<Placeable, Int>>()
        var currentGroupWidth = 0
        var currentGroupMaxHeight = 0
```



Esmaeel Moradi

@esmaeeil-moradi700

```

// Measure children and group into rows
measurables.forEach { measurable ->
    // Measure with parent constraints first
    val placeable = measurable.measure(constraints)
    val placeableWidthWithSpacing =
        placeable.width + spacingPx

    if (currentGroup.isNotEmpty() &&
        currentGroupWidth + placeableWidthWithSpacing >
        constraints.maxWidth
    ) {
        // Start a new row
        groupPositions.add(currentGroup)
        currentGroup = mutableListOf()
        currentGroupWidth = 0
        currentGroupMaxHeight = 0
    }

    currentGroup.add(Pair(placeable, currentGroupWidth))
    currentGroupWidth += placeableWidthWithSpacing
    currentGroupMaxHeight =
        maxOf(currentGroupMaxHeight, placeable.height)
}
if (currentGroup.isNotEmpty()) {
    groupPositions.add(currentGroup) // Add the last group
}

// Calculate layout size
// Usually takes full width or max used width
val layoutWidth = constraints.maxWidth
val layoutHeight =
    groupPositions.foldIndexed(0) { index, acc, group ->
        val rowHeight =
            group.maxOfOrNull { it.first.height } ?: 0
        acc + rowHeight + if (index > 0) spacingPx else 0
    }

// Place children
layout(layoutWidth, layoutHeight) {
    var yPosition = 0
    groupPositions.forEach { group ->
        // Max height in the current row to correctly position the next
        var rowMaxHeight = 0
        group.forEach { (placeable, xPos) ->
            placeable.placeRelative(x = xPos, y = yPosition)
            rowMaxHeight =
                maxOf(rowMaxHeight, placeable.height)
        }
        yPosition += rowMaxHeight + spacingPx
    }
}
}
}
}

```



Esmaeel Moradi

@esmaeeil-moradi700



(This is a simplified example; a complete FlowLayout implementation can be more complex, handling vertical spacing, alignment within rows, etc.).

- **Handling Modifiers in Custom Layouts:** Modifiers applied to the custom Layout itself (like padding, size) affect the constraints passed into the measure function. Modifiers applied to the children are handled by the children themselves during their measure() call.
- **Performance Considerations:** Custom layouts can be highly performant if their measurement and placement logic is efficient. However, complex or inefficient implementations can degrade performance.



Esmaeel Moradi
@esmaeel-moradi700



Summary:

Choosing the Right Layout Strategy:

- **Column, Row, Box:** For simple, standard linear or stacked arrangements. Fastest and most readable for common cases.
- **ConstraintLayout:** For complex relationships between elements, creating flat hierarchies, and when you need helpers like Guidelines or Barriers.
- **Custom Layout:** For performance optimization in specific scenarios, implementing non-standard layout patterns (FlowLayout, StaggeredGrid, PieChartLayout), or achieving maximum flexibility and unique designs.

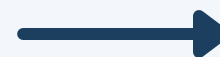
The Best Approach:

Start with the simplest tool (Column/Row/Box). If complexity increases or you need flattening, consider ConstraintLayout. If neither suffices or you need specific performance/design goals, venture into Custom Layouts. Balance readability, performance, and required flexibility.



Esmaeel Moradi

@esmaeel-moradi700



Conclusion:

Build Any UI Imaginable with Compose.

The Jetpack Compose layout system is remarkably powerful, flexible, and extensible. From basic building blocks to ConstraintLayout and the ultimate freedom of custom Layouts, you have the tools needed to implement any user interface imaginable – from the simplest to the most intricate and creative. Don't be afraid of the power of custom layouts; explore them, experiment, and use them to build unique and optimized user experiences. The future of UI design in Android is in your hands and your creativity with Compose!



Esmaeeil Moradi
@esmaeeil-moradi700

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



LEAVE A COMMENT BELOW



Esmaeel Moradi

@esmaeel-moradi700