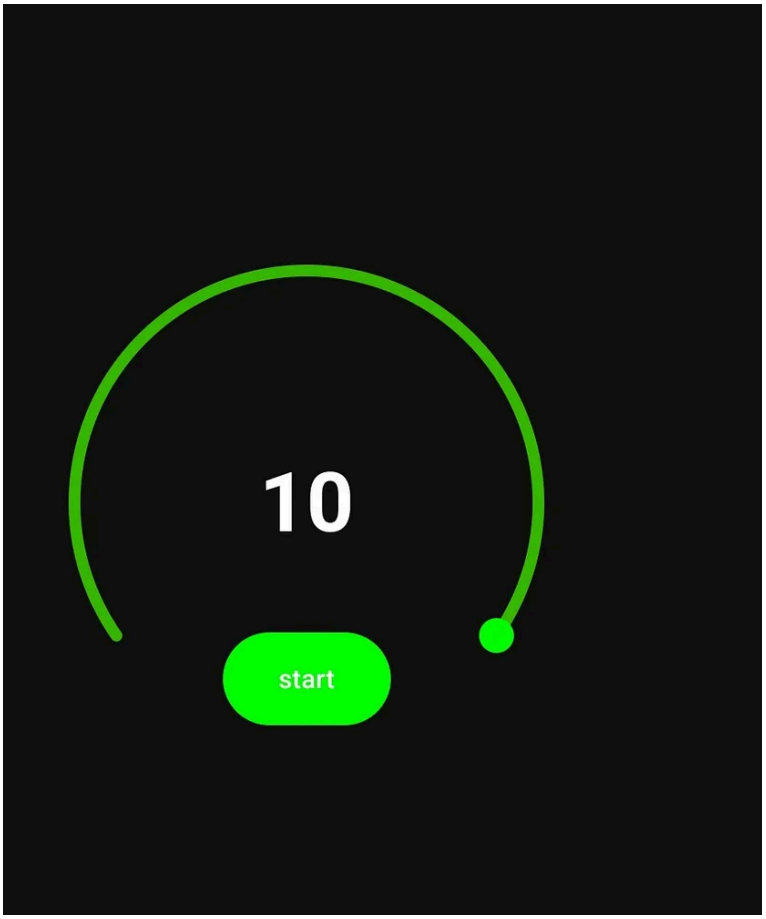
Craft a Captivating Animated Countdown Timer with Jetpack Compose

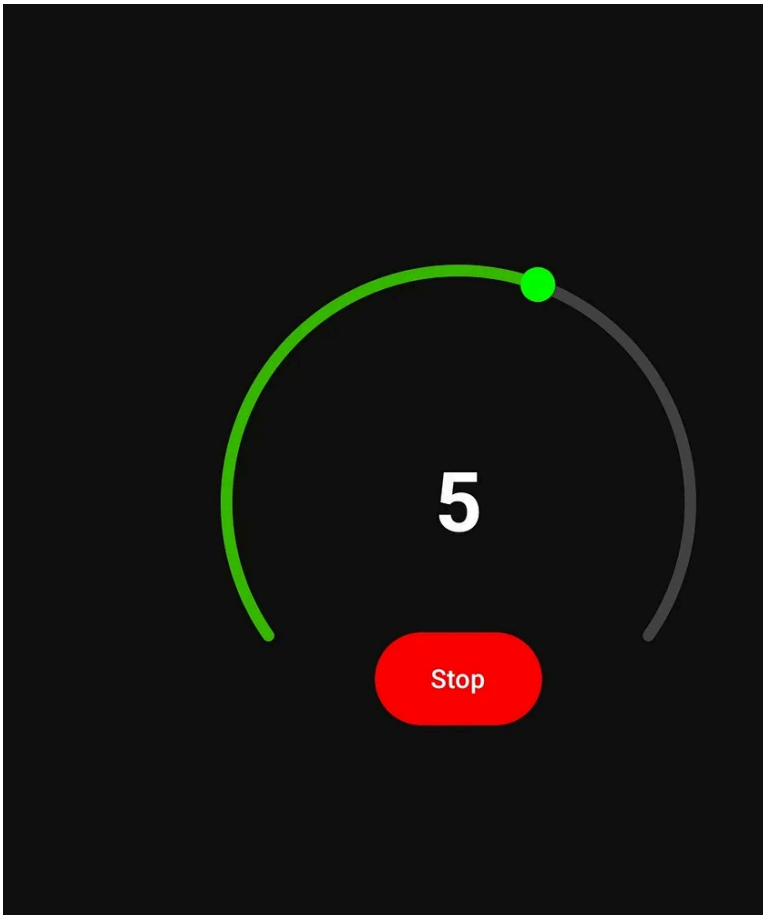
Iamfahad · Follow

5 min read · 6 days ago









Jetpack Compose empowers you to build modern and interactive UI elements for your Android apps. This guide takes you through the process of creating a customizable countdown timer using Jetpack Compose's powerful features.

What We'll Build:

A circular countdown timer with the following functionalities:

- **Customization:** Set the total duration, handle and bar colors.
- **Start/Stop:** Control the timer's running state.
- **Dynamic Button:** Button color and text change based on the timer state.
- **Visual Representation:** Displays the remaining time in seconds.

Prerequisites:

- Basic understanding of Jetpack Compose and Kotlin
- An Android Studio project set up

Step 1: Building the Timer Composable Function

1. Create a new composable function named `Timer`.
2. This function takes arguments for customization:

- `totalTime` : Total duration of the timer in milliseconds (e.g., 10000L for 10 seconds).
- `handleColor` : Color for the timer handle.
- `activeBarColor` : Color for the active portion of the timer bar.
- `inactiveBarColor` : Color for the inactive portion of the timer bar.
- Other optional arguments like `modifier` (to set size and position), `initialValue` (for initial progress), and `strokeWidth` (for the thickness of the timer bar).

```
@Composable
fun Timer(
    totalTime: Long,
    handleColor: Color,
    activeBarColor: Color,
    inactiveBarColor: Color,
    modifier: Modifier = Modifier,
    initialValue: Float = 1f,
    strokeWidth: Dp = 5.dp
) {
    // ... rest of the code
}
```

Step 2: Managing Internal State Variables

1. Inside the `Timer` composable, use `remember` to manage internal state variables:

- `size` : Tracks the size of the composable (`IntSize.Zero` initially).
- `value` : Represents the current progress of the timer (a float between 0 and 1).
- `currentTime` : Holds the remaining time in milliseconds (starts with `totalTime`).
- `isTimerRunning` : Indicates if the timer is currently running (initially `false`).

```
var size by remember {
        mutableStateOf(IntSize.Zero)
    }
    var value by remember {
        mutableFloatStateOf(initialValue)
    }
    var currentTime by remember {
        mutableLongStateOf(totalTime)
    }
    var isTimerRunning by remember {
```

```
        mutableStateOf(false)
    }
```

Step 3: Implementing Timer Logic with LaunchedEffect

1. A `LaunchedEffect` observes changes in `currentTime` and `isTimerRunning`.
2. If the timer is running (`isTimerRunning`) and there's remaining time (`currentTime` > 0), it decrements the `currentTime` every 100 milliseconds and updates the `value` accordingly (dividing by `totalTime` to get a 0-1 progress). This effectively tracks the timer's progress.

```
LaunchedEffect(key1 = currentTime, key2 = isTimerRunning) {
    if (currentTime > 0 && isTimerRunning) {
        delay(100L)
        currentTime -= 100L
        value = currentTime / totalTime.toFloat()
    }
}
```

Step 4: Drawing the Timer with Canvas

1. A `Canvas` composable allows you to draw custom shapes and visuals.
2. Within the `Canvas`:

- Draw two arcs:
- An outer arc using `inactiveBarColor` represents the inactive portion of the timer bar.
- An inner arc using `activeBarColor` represents the remaining time, with its size determined by the `value`.
- Draw the timer handle as a circle at the end of the active arc using `drawPoints` with `handleColor`.

```
Canvas(modifier = modifier){
    drawArc(
        color = inactiveBarColor,
        startAngle = -215f,
        sweepAngle = 250f,
        useCenter = false,
        size = Size(size.width.toFloat(), size.height.toFloat()),
        style = Stroke(strokeWidth.toPx(),cap= StrokeCap.Round)
    )
    drawArc(
        color = activeBarColor,
        startAngle = -215f,
        sweepAngle = 250f * value,
```

```
            useCenter = false,
            size = Size(size.width.toFloat(), size.height.toFloat()),
            style = Stroke(strokeWidth.toPx(),cap= StrokeCap.Round)
        )
        val center = Offset(size.width/2f,size.height/2f)
        val beta = (250f * value + 145f) * (PI / 180f).toFloat()
        val r = size.width/2f
        val a = cos(beta) * r
        val b = sin(beta) * r

        drawPoints(
            listOf(Offset(center.x + a,center.y + b)),
            pointMode = PointMode.Points,
            color = handleColor,
            strokeWidth = (strokeWidth * 3f).toPx(),
            cap = StrokeCap.Round
        )
    }
```

Step 5: Displaying Remaining Time and Button

1. A `Text` composable displays the remaining time in seconds (`currentTime / 1000L`).
2. A `Button` allows users to start, stop, or restart the timer:

- The button color changes based on the timer state using `ButtonDefaults.buttonColors`:

- Green for start or restart when the timer is not running or has reached zero.
- Red for stop when the timer is running.
- The button text changes depending on the timer state as well (using an if statement).

```
Text(
    text = (currentTime / 1000L).toString(),
    fontSize = 44.sp,
    fontWeight = FontWeight.Bold,
    color = Color.White
)
Button(
    onClick = {
        if(currentTime <= 0L) {
            currentTime = totalTime
            isTimerRunning = true
        } else {
            isTimerRunning = !isTimerRunning
        }
    },
    modifier = Modifier
        .align(Alignment.BottomCenter),
    colors = ButtonDefaults.buttonColors(containerColor =
           if (!isTimerRunning || currentTime <= 0L) {
            Color.Green
        }
           else {
```

```
                    Color.Red
                }
            )
        ) {
            Text(
                text =
                if (isTimerRunning && currentTime>0L) "Stop"
                else if(!isTimerRunning && currentTime>=0L) "start"
                else "restart"
            )
        }
```

Step 6: Combine All Together In — The Box Layout

1. The `Timer` composable combines these elements within a `Box` with `contentAlignment` set to `Alignment.Center`.
2. The `Canvas`, `Text`, and `Button` are positioned within the `Box` using modifiers and alignments

```
Box(
        contentAlignment = Alignment.Center,
        modifier = modifier.onSizeChanged {
            size = it
        }
    ){
```

```
// Box Scope
}
```

Step 7: Putting it All Together: Bringing the Timer to Life

Now that we've built the `Timer` composable function with its internal logic and visuals, it's time to integrate it into your Android application. This involves utilizing Jetpack Compose's `setContent` function, which defines the root of your composable hierarchy.

The `setContent` function serves as the entry point for your composable UI. It dictates the content displayed on the screen. Here's how we'll use it to showcase our countdown timer:

```
setContent {
    Surface(
        color = Color(0xFF101010),
        modifier = Modifier.fillMaxSize()
    ) {
        Box(
            contentAlignment = Alignment.Center
        ) {
            Timer(
                totalTime = 10L * 1000L,
```

```
                handleColor = Color.Green,
                inactiveBarColor = Color.DarkGray,
                activeBarColor = Color(0xFF37B900),
                modifier = Modifier.size(200.dp)
            )
        }
    }
}
```

The `DefaultPreviewTimer` Composable (Optional):

allows you to preview the timer composable in Android Studio's preview pane, providing a visual aid during development. Here's an example:

```
@Preview
@Composable
fun DefaultPreviewTimer() {
    Surface(
        color = Color(0xFF101010),
        modifier = Modifier.fillMaxSize()
    ) {
        Box(
            contentAlignment = Alignment.Center
        ) {
            Timer(
                totalTime = 10L * 1000L,
```

```
                handleColor = Color.Green,
                inactiveBarColor = Color.DarkGray,
                activeBarColor = Color(0xFF37B900),
                modifier = Modifier.size(200.dp)
            )
        }
    }
}
```

Conclusion:

By following this guide, you can create a custom timer with animation using Jetpack Compose in your Android app. Enhance your user experience by incorporating interactive and visually appealing timers into your application.

Happy coding!

Android | Android App Development | Kotlin | Jetpack Compose | Android Studio