Tutorial Completo de Jetpack Compose

Guía integral para dominar Jetpack Compose - Desde conceptos básicos hasta implementaciones avanzadas en Android con Kotlin

Índice de Contenidos

1. Introducción a Jetpack Compose

Jetpack Compose es un framework moderno de Android para crear interfaces de usuario de forma declarativa. Este tutorial te guiará desde los conceptos básicos hasta implementaciones avanzadas.

Código Base - MainActivity

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HolaMundoTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HolaMundoTheme {
        Greeting("Android")
    }
}
```

Componentes Principales:

- **MainActivity**: Es la actividad principal de la app, hereda de `ComponentActivity`
- **onCreate()**: Método que se ejecuta cuando se crea la actividad
- **enableEdgeToEdge()**: Permite que la app use toda la pantalla, incluyendo las áreas del notch/barras de sistema
- **setContent {}**: Define el contenido visual usando Compose en lugar del tradicional XML

> ### Jetpack Compose
>
> Framework moderno de Android para crear interfaces de usuario

> ### ComponentActivity

> Clase base para actividades que usan Compose

> **@Composable**
>
> Anotación que marca funciones que pueden crear UI

> **Scaffold**
>
> Estructura de layout que implementa Material Design

2. Data Classes y MessageCard

¿Qué es una Data Class?

Una **data class** es un tipo especial de clase en Kotlin diseñada específicamente para **almacenar datos**.

```
data class Message(val author: String, val body: String)
```

Características clave:

- **data class**: Tipo especial de clase optimizada para almacenar datos
- **val author**: Propiedad inmutable que almacena el nombre del autor
- **val body**: Propiedad inmutable que contiene el contenido del mensaje

Implementación con MessageCard

```
@Composable
fun MessageCard(msg: Message) {
    Column {
        Text(text = msg.author)
        Text(text = msg.body)
    }
}
```

¿Qué genera automáticamente Kotlin?

- **equals() y hashCode()** - Compara contenido, no referencia
- **toString()** - Representación legible del objeto
- **copy()** - Crear copias modificadas del objeto
- **componentN()** - Destructuring de propiedades

Ejemplos Prácticos

```
// Crear instancias
val mensaje1 = Message("Android", "Jetpack Compose")
val mensaje2 = Message("Kotlin", "Es genial")

// Comparación automática
val msg1 = Message("Juan", "Hola")
val msg2 = Message("Juan", "Hola")
println(msg1 == msg2) // true - compara el contenido

// Copia con modificaciones
val original = Message("Carlos", "Mensaje original")
val copia = original.copy(body = "Mensaje modificado")
```

3. Contenedores de Layout

Los contenedores de layout son fundamentales para organizar elementos en la interfaz de usuario de Jetpack Compose.

Column - Layout Vertical

`Column` organiza sus elementos hijos **verticalmente**, uno debajo del otro.

```
@Composable
fun VerticalLayout() {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Primer elemento")
        Text("Segundo elemento")
        Button(onClick = {}) {
            Text("Botón")
        }
    }
}
```

Row - Layout Horizontal

`Row` organiza sus elementos hijos **horizontalmente**, uno al lado del otro.

```
@Composable
fun HorizontalLayout() {
    Row(
        modifier = Modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.SpaceBetween,
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text("Izquierda")
        Text("Centro")
        Text("Derecha")
    }
}
```

Box - Superposición de Elementos

`Box` permite **superponer** elementos unos encima de otros, como capas.

```
@Composable
fun OverlayLayout() {
    Box {
        Image(
            painter = painterResource(R.drawable.background),
            contentDescription = null
        )
        Text(
            text = "Texto superpuesto",
            color = Color.White,
            modifier = Modifier.align(Alignment.Center)
        )
    }
}
```

Column

Elementos apilados verticalmente, formularios, listas simples

Row

Elementos en línea horizontal, toolbars, headers

Box

Superposición de elementos, overlays, badges sobre iconos

4. LazyColumn y LazyRow

LazyColumn - Lista Vertical Eficiente

`LazyColumn` crea **listas verticales eficientes** que solo renderizan los elementos visibles en pantalla.

```
@Composable
fun VerticalList() {
    val items = (1..1000).map { "Item #$it" }

    LazyColumn {
        items(items) { item ->
            Text(
                text = item,
                modifier = Modifier.padding(16.dp)
            )
        }
    }
}
```

¿Cómo funciona LazyColumn?

LazyColumn utiliza **virtualización** para renderizar solo los elementos visibles, lo que permite manejar miles de elementos sin problemas de memoria.

LazyRow - Lista Horizontal Eficiente

`LazyRow` crea **listas horizontales eficientes** con scroll horizontal.

```
@Composable
fun HorizontalList() {
    val categories = listOf("Todos", "Deportes", "Música", "Arte", "Ciencia")

    LazyRow(
        contentPadding = PaddingValues(horizontal = 16.dp),
        horizontalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(categories) { category ->
            FilterChip(
                selected = false,
                onClick = { },
                label = { Text(category) }
            )
        }
    }
}
```

Casos de Uso Comunes

Componente	Mejor para	Ejemplos
LazyColumn	Listas verticales largas	Contactos, feeds de noticias, catálogos
LazyRow	Listas horizontales	Chips de categorías, carrusel de imágenes

5. Componentes Básicos de Jetpack Compose

Esta sección cubre los componentes más utilizados en Jetpack Compose, organizados por categorías.

Componentes de Texto e Imágenes

Text

```
Text(
    text = "Hola Mundo",
    style = MaterialTheme.typography.headlineMedium,
    color = Color.Blue
)
```

Image

```
Image(
    painter = painterResource(R.drawable.mi_imagen),
    contentDescription = "Descripción",
    modifier = Modifier.size(100.dp)
)
```

Botones e Interacciones

Button

```
Button(onClick = { /* acción */ }) {
    Text("Presionar")
}
```

OutlinedButton

```
OutlinedButton(onClick = { /* acción */ }) {
    Text("Cancelar")
}
```

Switch

```
var checked by remember { mutableStateOf(false) }
Switch(
    checked = checked,
    onCheckedChange = { checked = it }
)
```

Campos de Input

TextField

```
var text by remember { mutableStateOf("") }
TextField(
    value = text,
    onValueChange = { text = it },
    label = { Text("Nombre") }
)
```

> **Text**

Mostrar cualquier texto - títulos, párrafos, etiquetas

Button

Acciones principales, envío de formularios

TextField

Formularios, búsquedas, entrada de datos

Switch

Configuraciones booleanas, activar/desactivar

6. Componente Text - Guía Detallada

El componente `Text` es uno de los elementos más fundamentales en Jetpack Compose, responsable de mostrar texto con múltiples opciones de personalización.

Sintaxis Básica

```
Text(text = "Hola Mundo")
```

Estructura Completa

```
Text(
    text = "Mi texto",
    modifier = Modifier,
    color = Color.Unspecified,
    fontSize = TextUnit.Unspecified,
    fontStyle = null,
    fontWeight = null,
    fontFamily = null,
    letterSpacing = TextUnit.Unspecified,
    textDecoration = null,
    textAlign = null,
    lineHeight = TextUnit.Unspecified,
    overflow = TextOverflow.Clip,
    softWrap = true,
    maxLines = Int.MAX_VALUE,
    style = LocalTextStyle.current
)
```

Tipografía Material Design

```
@Composable
fun TypographyExamples() {
    Column(verticalArrangement = Arrangement.spacedBy(8.dp)) {
        Text("Display Large", style = MaterialTheme.typography.displayLarge)
        Text("Headline Medium", style = MaterialTheme.typography.headlineMedium)
        Text("Title Large", style = MaterialTheme.typography.titleLarge)
        Text("Body Large", style = MaterialTheme.typography.bodyLarge)
        Text("Body Medium", style = MaterialTheme.typography.bodyMedium)
        Text("Label Large", style = MaterialTheme.typography.labelLarge)
    }
}
```

Estilo	Uso recomendado
`displayLarge`	Títulos muy grandes, splash screens
`headlineMedium`	Subtítulos principales
`titleLarge`	Títulos de secciones
`bodyLarge`	Texto principal grande
`bodyMedium`	Texto principal estándar
`labelLarge`	Labels de botones grandes

Personalización Avanzada

Colores y Estilos

```
Text(
    text = "Texto personalizado",
    color = MaterialTheme.colorScheme.primary,
    fontSize = 24.sp,
    fontWeight = FontWeight.Bold,
    textDecoration = TextDecoration.Underline
)
```

Manejo de Overflow

```
Text(
    text = "Texto muy largo que se cortará con puntos suspensivos",
    maxLines = 1,
    overflow = TextOverflow.Ellipsis,
    modifier = Modifier.width(200.dp)
)
```

✅ Mejores Prácticas

- Usar estilos de tipografía de MaterialTheme
- Limitar maxLines para textos largos
- Usar overflow = TextOverflow.Ellipsis
- Mantener contraste adecuado
- Considerar tamaño mínimo de 16.sp

7. Unidades de Medida en Jetpack Compose

Elegir la unidad de medida correcta es crucial para crear interfaces responsivas y consistentes en diferentes dispositivos.

Dp (Density-independent pixels)

Definición: Píxeles independientes de densidad. Se escalan automáticamente según la densidad del dispositivo.

```
// Ejemplos de uso con dp
Box(modifier = Modifier.size(48.dp))  // Tamaño táctil mínimo
Text(
    text = "Texto con padding",
    modifier = Modifier.padding(16.dp)
)
Card(elevation = CardDefaults.cardElevation(defaultElevation = 4.dp))
```

Sp (Scale-independent pixels)

Definición: Similar a dp pero también considera las preferencias de tamaño de fuente del usuario.

```
// Ejemplos de uso con sp
Text(text = "Título", fontSize = 24.sp)
Text(text = "Cuerpo", fontSize = 16.sp)
Text(
    text = "Texto con espaciado",
    fontSize = 16.sp,
    lineHeight = 24.sp
)
```

> ⚠️ **Cuándo usar cada unidad**
>
> - **Dp**: Dimensiones físicas, espaciado, tamaños de elementos UI
> - **Sp**: Tamaños de fuente, espaciado de líneas
> - **Px**: Solo en casos muy específicos (evitar)

Dimensiones Flexibles

fillMax* - Llenar espacio disponible

```
Column(modifier = Modifier.fillMaxSize()) {
    Box(
        modifier = Modifier
            .fillMaxWidth()      // Ancho completo
            .height(60.dp)
    )
    Box(
        modifier = Modifier
            .fillMaxWidth(0.7f)  // 70% del ancho
            .height(60.dp)
    )
}
```

weight - Distribución proporcional

```
Row(modifier = Modifier.fillMaxWidth()) {
    Box(modifier = Modifier.weight(1f))  // 1 parte
    Box(modifier = Modifier.weight(2f))  // 2 partes (doble)
    Box(modifier = Modifier.weight(1f))  // 1 parte
}
```

Unidad	Cuándo usar	Ejemplos
dp	Elementos UI, espaciado	Padding, tamaños de botones, elevación
sp	Texto y relacionado	Tamaños de fuente, line height
weight	Distribución proporcional	Layouts flexibles, columnas
fillMax*	Ocupar espacio disponible	Backgrounds, contenedores

Valores Recomendados Material Design

```
object MaterialSpacing {
    val xs = 4.dp      // Espaciado extra pequeño
    val small = 8.dp   // Espaciado pequeño
    val medium = 16.dp // Espaciado estándar
    val large = 24.dp  // Espaciado grande
    val xl = 32.dp     // Espaciado extra grande
}
```

8. Temas y Material Design

Los temas en Jetpack Compose proporcionan una forma consistente de aplicar colores, tipografía y formas a través de toda la aplicación.

Estructura Básica de MaterialTheme

```
@Composable
fun MyApp() {
    MaterialTheme(
        colorScheme = lightColorScheme(),
        typography = Typography(),
        shapes = Shapes()
    ) {
        // Tu contenido aquí
        MyAppContent()
    }
}
```

Color Scheme Personalizado

```
val CustomLightColorScheme = lightColorScheme(
    primary = Color(0xFF6750A4),
    onPrimary = Color(0xFFFFFFFF),
    primaryContainer = Color(0xFFEADDFF),
    onPrimaryContainer = Color(0xFF21005D),
    secondary = Color(0xFF625B71),
    onSecondary = Color(0xFFFFFFFF),
    background = Color(0xFFFFFBFE),
    onBackground = Color(0xFF1C1B1F),
    surface = Color(0xFFFFFBFE),
    onSurface = Color(0xFF1C1B1F)
)
```

Tipografía Personalizada

```
val CustomTypography = Typography(
    displayLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 57.sp,
        lineHeight = 64.sp
    ),
    headlineMedium = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 28.sp,
        lineHeight = 36.sp
    ),
    bodyLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp,
        lineHeight = 24.sp
    )
)
```

Tema Completo con Soporte Claro/Oscuro

```
@Composable
fun MyCustomTheme(
    useDarkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colorScheme = if (useDarkTheme) {
        CustomDarkColorScheme
    } else {
        CustomLightColorScheme
    }

    MaterialTheme(
        colorScheme = colorScheme,
        typography = CustomTypography,
        shapes = CustomShapes,
        content = content
    )
}
```

Material You y Colores Dinámicos

En Android 12+ puedes usar colores dinámicos que se adaptan al wallpaper del usuario:

```
val dynamicColor = Build.VERSION.SDK_INT >= Build.VERSION_CODES.S
val colorScheme = when {
    dynamicColor && useDarkTheme -> dynamicDarkColorScheme(context)
    dynamicColor && !useDarkTheme -> dynamicLightColorScheme(context)
    useDarkTheme -> CustomDarkColorScheme
    else -> CustomLightColorScheme
}
```

✅ Mejores Prácticas para Temas

- Usar MaterialTheme como base para consistencia
- Definir colores semánticos (primary, secondary, error)
- Soportar tema claro y oscuro siempre
- Crear constantes reutilizables
- Seguir las guías de Material Design
- Considerar accesibilidad (contraste, tamaños)

9. Generación Segura de Iniciales

La generación de iniciales es un caso común en aplicaciones para crear avatares personalizados. Aquí aprenderemos a hacerlo de forma segura.

Código Principal

```
val iniciales = "${nombre.firstOrNull() ?: ""}${apellido.firstOrNull() ?: ""}"
        .uppercase()
```

> Esta línea implementa una forma **robusta y segura** de extraer iniciales, manejando automáticamente casos donde el nombre o apellido puedan estar vacíos.

Análisis Detallado

1. Función firstOrNull()

```
nombre.firstOrNull()  // Devuelve el primer carácter o null
```

- ✅ `firstOrNull()` → Devuelve null de forma segura
- ❌ `first()` → Lanza excepción si está vacío

2. Operador Elvis (?:)

```
nombre.firstOrNull() ?: ""  // Si es null, usa string vacío
```

3. Template Strings

```
"${expresión1}${expresión2}"  // Concatena las iniciales
```

4. Función uppercase()

```
.uppercase()  // Convierte a mayúsculas
```

Casos de Uso

Nombre	Apellido	Iniciales	Explicación
"Ana"	"García"	"AG"	Caso normal
""	"García"	"G"	Nombre vacío
"Ana"	""	"A"	Apellido vacío
""	""	""	Ambos vacíos

Función Reutilizable Mejorada

```
fun generarIniciales(nombre: String?, apellido: String?): String {
    val n = nombre?.trim()?.firstOrNull() ?: ""
    val a = apellido?.trim()?.firstOrNull() ?: ""
    return "$n$a".uppercase().ifEmpty { "?" }
}
```

Implementación en Compose - Avatar

```
@Composable
fun Avatar(
    nombre: String,
    apellido: String,
    backgroundColor: Color = Color(0xFF3498DB)
) {
    val iniciales = "${nombre.trim().firstOrNull() ?: ""}${apellido.trim().firstOrNull() ?: ""}"
        .uppercase()
        .ifEmpty { "?" }

    Box(
        modifier = Modifier
            .size(80.dp)
            .clip(CircleShape)
            .background(backgroundColor),
        contentAlignment = Alignment.Center
    ) {
        Text(
            text = iniciales,
            color = Color.White,
            fontWeight = FontWeight.Bold,
            fontSize = 28.sp
        )
    }
}
```

❌ Enfoque Peligroso (Evitar)

```
// PELIGROSO - Puede causar crash
val iniciales = "${nombre[0]}${apellido[0]}".uppercase()
// IndexOutOfBoundsException si está vacío
```

✅ Puntos Clave

- Siempre usar funciones seguras (`firstOrNull()`)
- Usar operador Elvis para valores por defecto
- Considerar casos edge como strings vacíos
- Escribir código que sea imposible que crashee

🎉 Conclusión

Este tutorial ha cubierto los aspectos fundamentales y avanzados de **Jetpack Compose**. Con estos conocimientos, puedes crear aplicaciones Android modernas, eficientes y visualmente atractivas.

Próximos Pasos:

- Practicar implementando cada concepto en proyectos reales
- Explorar animaciones avanzadas en Compose
- Integrar con Architecture Components (ViewModel, Navigation)
- Optimizar performance y testing