

JETPACK COMPOSE III

2025

BEGOÑA RODRÍGUEZ FERRERAS
brodfer@gmail.com



Controles de botón

Vamos a cambiar en la clase el modelarlos como un booleano a que sea un string, no porque no vaya a funcionar. Sino porque así luego lo mostramos directamente.

```
// RadioButton para "Hombre"
RadioButton(
    selected = sexo == "Hombre",
    onClick = { sexo = "Hombre" } // Actualiza el estado al seleccionar "Hombre"
)
Text(text = "Hombre", modifier = Modifier.weight(0.25f))

// RadioButton para "Mujer"
RadioButton(
    selected = sexo == "Mujer",
    onClick = { sexo = "Mujer" } // Actualiza el estado al seleccionar "Mujer"
)
Text(text = "Mujer", modifier = Modifier.weight(0.25f))
```

Ciclo de vida de los elementos componibles

Voy a aprovechar el ejercicio que hicisteis ayer para explicar esto:

Descripción general del ciclo de vida

<https://developer.android.com/develop/ui/compose/lifecycle?hl=es-419>

Un elemento Composition describe la IU de tu app y se produce ejecutando elementos que admiten composición. Es una estructura de árbol de esos elementos que describe tu IU.

Cuando Jetpack Compose ejecute tus elementos componibles por primera vez, durante la **composición inicial**, mantendrá un registro de los elementos componibles a los que llamas para describir tu IU en un objeto Composition. Luego, cuando cambie el estado de la app, Jetpack Compose programará una **recomposición**. Este evento se genera cuando **Jetpack Compose vuelve a ejecutar los elementos componibles que pueden haberse modificado en respuesta a cambios de estado y, luego, actualiza la composición para reflejar los cambios**.

Un objeto Composition sólo puede producirse mediante una composición inicial y actualizarse mediante la recomposición. La única forma de modificar un objeto Composition es mediante la recomposición.

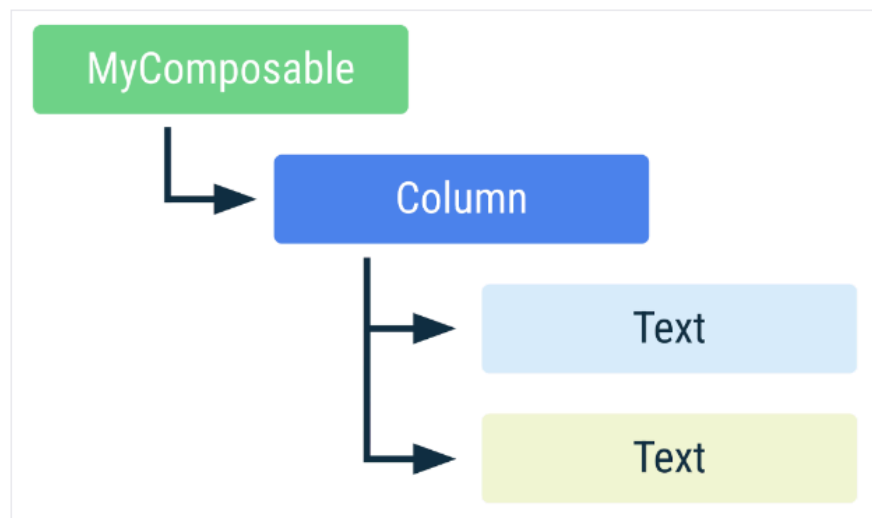
Punto clave: El ciclo de vida de un elemento componible se define mediante los siguientes eventos: ingresar el objeto Composition, volver a componer 0 o más veces, y dejar el objeto.

Por lo general, la recomposición se activa mediante un cambio en un objeto **State<T>**. Compose realiza un seguimiento de estas modificaciones y ejecuta todos los elementos componibles en el objeto Composition que lee ese **State<T>** determinado, y cualquier elemento componible que no se pueda omitir.

Si se llama varias veces a un elemento componible, se colocan varias instancias en el objeto Composition. Cada llamada tiene su propio ciclo de vida en Composition.

```
@Composable
fun MyComposable() {
    Column {
        Text("Hello")
        Text("World")
    }
}
```

[LifecycleSnippets.kt](#)



Anatomía de un elemento componible en Composition

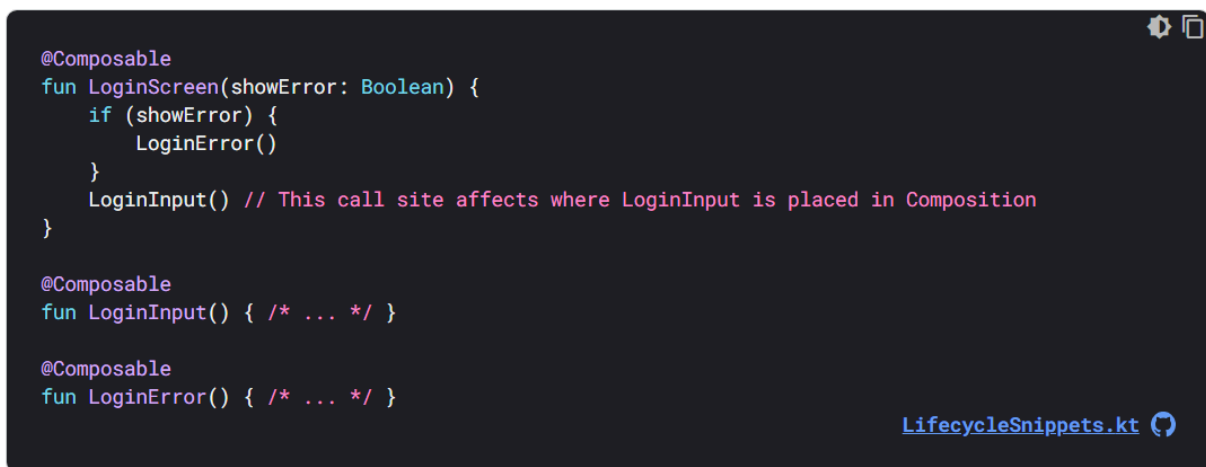
La instancia de un elemento componible en Composition se identifica mediante su **sitio de llamadas**. El compilador de Compose considera que cada sitio de llamadas es diferente. Invocar a elementos componibles desde varios sitios de llamadas creará varias instancias del elemento componible en Composition.

Término clave: El *sitio de llamadas* es la **ubicación del código fuente en la que se llama a un elemento componible**. Esto afecta su lugar en Composition y, por lo tanto, en el árbol de IU.

Si, durante una recomposición, un elemento componible llama a un elemento diferente del que invocó durante la composición anterior, Compose **identificará qué elementos componibles fueron llamados o no**. En el caso de los elementos llamados en ambas composiciones, Compose **evitará volver a componerlos si sus entradas no cambiaron**.

Preservar la identidad es fundamental para asociar efectos secundarios con su elemento componible a fin de que puedan completarse correctamente en lugar de reiniciarse para cada recomposición.


Consulta el siguiente ejemplo:

A screenshot of a code editor with a dark background. It displays three Kotlin Composable functions. The first function, @Composable fun LoginScreen(showError: Boolean) {, contains an if statement for showError and a call to LoginInput(). A comment explains that the call site affects the placement in Composition. The second function, @Composable fun LoginInput() { /* ... */ }, and the third, @Composable fun LoginError() { /* ... */ }, are both marked as Composables. In the bottom right corner, there is a link to 'LifecycleSnippets.kt' and a GitHub icon.

```
@Composable
fun LoginScreen(showError: Boolean) {
    if (showError) {
        LoginError()
    }
    LoginInput() // This call site affects where LoginInput is placed in Composition
}

@Composable
fun LoginInput() { /* ... */ }

@Composable
fun LoginError() { /* ... */ }
```

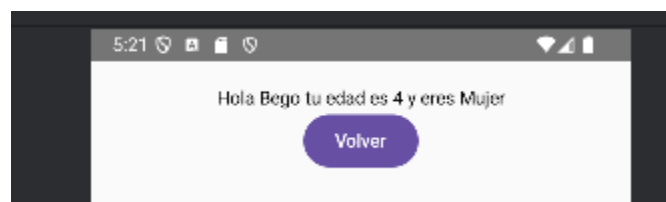
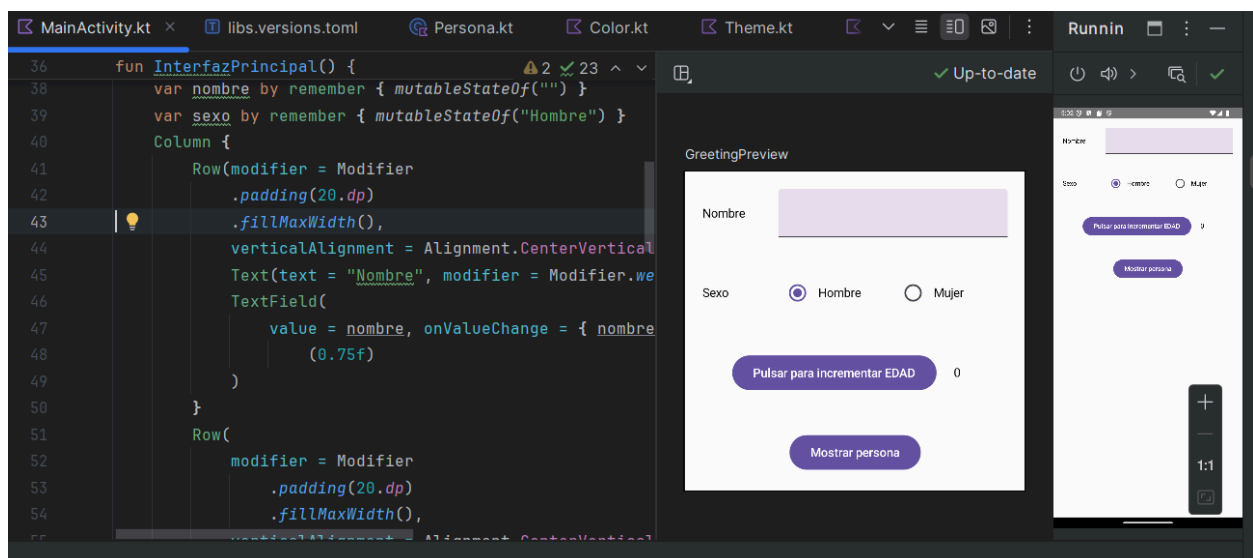
[LifecycleSnippets.kt](#) 

En el fragmento de código anterior, LoginScreen llamará condicionalmente al elemento componible LoginError y siempre llamará al elemento componible LoginInput. Cada llamada tiene un sitio de llamada y una posición en el código fuente únicos, que el compilador utilizará para identificarla de forma única.



Aunque LoginInput pasó de ser llamado en primer lugar al segundo, se conservará la instancia de LoginInput entre las recomposiciones. Además, debido a que LoginInput no tiene ningún parámetro que haya cambiado en la recomposición, Compose omitirá la llamada a LoginInput.

Vamos a verlo con el ejemplo que teníais hecho. Cuando se pulse en Mostrar persona, vamos a querer mostrar los datos que la persona haya introducido en el formulario. ¿Cómo podemos hacer eso de una forma sencilla?



Tendréis diferentes opciones para abordar esto. La más sencilla sería definir un booleano llamado **isSet** y modificar su valor según el usuario haya modificado los datos o quiera volver a introducirlos. Así de partida isSet sería false y mostraría los componentes del formulario original. Este booleano lo tenemos que definir donde tenemos nuestras variables y tiene que ser una variable cuyo estado sigamos.

```
@Composable
fun InterfazPrincipal() {
    var contadorEdad by remember { mutableStateOf(0) }
    var nombre by remember { mutableStateOf("") }
    var sexo by remember { mutableStateOf("Hombre") }
    var isSet by remember { mutableStateOf(false) }

    Column {
        if (!isSet) {
            Row(
                modifier = Modifier
                    .padding(20.dp)
                    .fillMaxWidth(),
                verticalAlignment = Alignment.CenterVertically
            ) {
                Text(text = "Nombre", modifier = Modifier.weight(0.25f))
                TextField(
                    value = nombre, onChange = { nombre = it }, modifier = Modifier
                        .weight(0.75f)
                )
            }
        }
    }
}
```

Entonces metemos todo el código original en un **isSet** negado y el nuevo código en el else:

```
    else {
        Column(modifier = Modifier
            .padding(20.dp)
            .fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally
        ){
            Text(text = "Hola " + persona.nombre + " tu edad es " + persona.edad +
            Button(onClick = {
                isSet = false
            }) {
                Text(text = "Volver")
            }
        }
    }
}
```

No va a funcionar directamente. Va a dar error. ¿Por qué da error? Por el scope de las variables. De momento en este ejemplo lo podemos solucionar. ¿Pero y si quisiéramos usar más funciones?

Elevación del estado

Creo que el ejemplo anterior ya os puede dar que pensar sobre qué puede ocurrir cuando tenemos diferentes funciones componibles, dado que no podemos devolver argumentos.

Cuando tu app se vuelva más compleja y otros elementos componibles necesiten acceder al estado dentro del elemento componible, deberás considerar la elevación o extracción del estado fuera de la función de componibilidad.

Debes elevar el estado cuando necesites hacer lo siguiente:

- Compartir el estado con varias funciones de componibilidad

-
- Crear un elemento sin estado componible que se pueda volver a usar en tu app

Cuando extraes el estado de una función de componibilidad, la función de componibilidad resultante se considera *sin estado*. Es decir, las funciones de componibilidad pueden dejar de tener estado si se lo extrae de ellas.

Un elemento sin estado componible significa que no contiene, define ni modifica un estado nuevo. Por otro lado, un elemento con estado componible es aquel que posee una parte de estado que puede cambiar con el tiempo.

Nota: En apps reales, tener un elemento componible 100% sin estado puede ser difícil de alcanzar según las responsabilidades de ese elemento. Debes diseñar tus elementos componibles de modo que tengan la menor cantidad de estado posible y permitir que se eleve cuando convenga, exponiéndolos en la API del elemento componible.

La **elevación de estado** es un patrón que consiste en mover el estado hacia el llamador para hacer que el componente no tenga estado.

Cuando se aplica a los elementos componibles, esto suele implicar incorporar dos parámetros a este elemento:

- Un **parámetro value**: T, que es el valor actual que se mostrará.
- Una **lambda de devolución** de llamada onChange: (T) -> Unit, que se activa cuando cambia el valor para que el estado se pueda actualizar en otro lugar, como cuando un usuario ingresa texto en el cuadro de texto.

Por ejemplo si fuéramos a sacar los datos de nuestra función como un String:

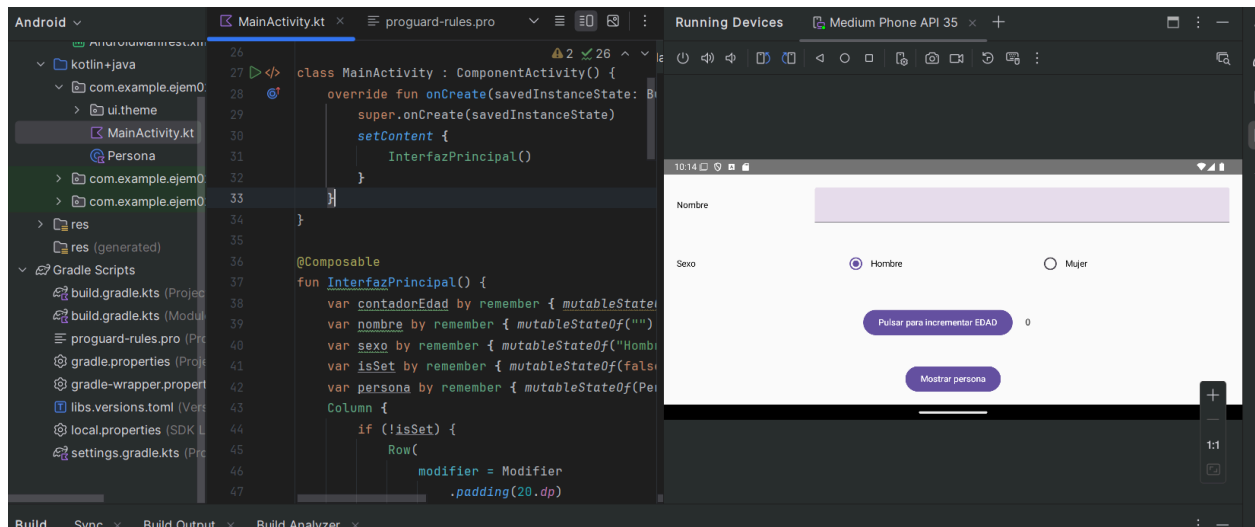
```
@Composable
fun EditNumberField(
    value: String,
    onValueChange: (String) -> Unit,
    modifier: Modifier = Modifier
) {
    //...
```

y obviamente tendríais que cambiar los parámetros en el textField.

Cambios de configuración

Un **cambio de configuración** ocurre cuando cambia el estado del dispositivo de manera tan radical que la forma más simple de que el sistema resuelva el cambio es cerrar y volver a crear la actividad por completo. Por ejemplo, si el usuario cambia el idioma del dispositivo, es posible que todo el diseño deba cambiar para adaptarse a diferentes orientaciones y longitudes de texto. Si el usuario enchufa el dispositivo a un conector o agrega un teclado físico, es posible que el diseño de la app deba aprovechar otro diseño o tamaño de visualización. Además, si cambia la orientación del dispositivo (si el dispositivo rota del modo vertical al modo horizontal, o viceversa), es posible que el diseño deba modificarse a fin de que se ajuste a la nueva orientación.

Rotamos el dispositivo en el emulador y el texto desaparece:



Las **funciones de componibilidad** tienen su propio ciclo de vida, que es **independiente del ciclo de vida de la actividad**. Su ciclo de vida está compuesto por los siguientes eventos: ingresar a la composición, volver a componer 0 o más veces y, luego, salir de la composición.

Para que Compose realice el seguimiento y active una recomposición, **debe saber cuándo cambió el estado**. Para indicarle a Compose que debe seguir el estado de un objeto, este debe ser del tipo State o MutableState. El tipo State es inmutable y solo se puede leer. Un tipo MutableState es mutable y permite lecturas y escrituras.

Ahora, creo que habéis visto en el temario que el ciclo de vida de las actividades de Android es distinto. Para indicarle a Compose que conserve y vuelva a usar su valor durante las recomposiciones, debes declararlo con la API de remember.

```
var contadorEdad by remember { mutableStateOf(0) }
var nombre by remember { mutableStateOf("") }
var sexo by remember { mutableStateOf("Hombre") }
var isSet by remember { mutableStateOf(false) }
var persona by remember { mutableStateOf(Persona("", 0, "Hombre")) }
```

Si bien Compose recuerda el estado de los ingresos durante las recomposiciones, no retiene este estado durante un cambio de configuración. Para que lo haga, debes usar `rememberSaveable`.

Usa la función **`rememberSaveable`** a fin de guardar los valores que necesitarás si el SO Android destruye y vuelve a crear la actividad.

Para guardar valores durante las recomposiciones, debes usar `remember`. Usa **`rememberSaveable`** para guardar valores durante las recomposiciones Y los cambios de configuración.

Nota: En ocasiones, Android cierra todo el proceso de la app, lo que incluye todas las actividades asociadas con ella. Android realiza este tipo de cierre cuando el sistema está sobrecargado y corre el riesgo de tener un atraso visual, por lo que en este momento no se ejecutan devoluciones de llamadas ni código adicional. El proceso de tu app simplemente se cierra, de manera silenciosa, en segundo plano. Sin embargo, para el usuario, no parece que la app se haya cerrado. Cuando el usuario vuelve a la app que el sistema cerró, Android la reinicia. Asegúrate de que el usuario no experimente ninguna pérdida de datos cuando esto suceda.

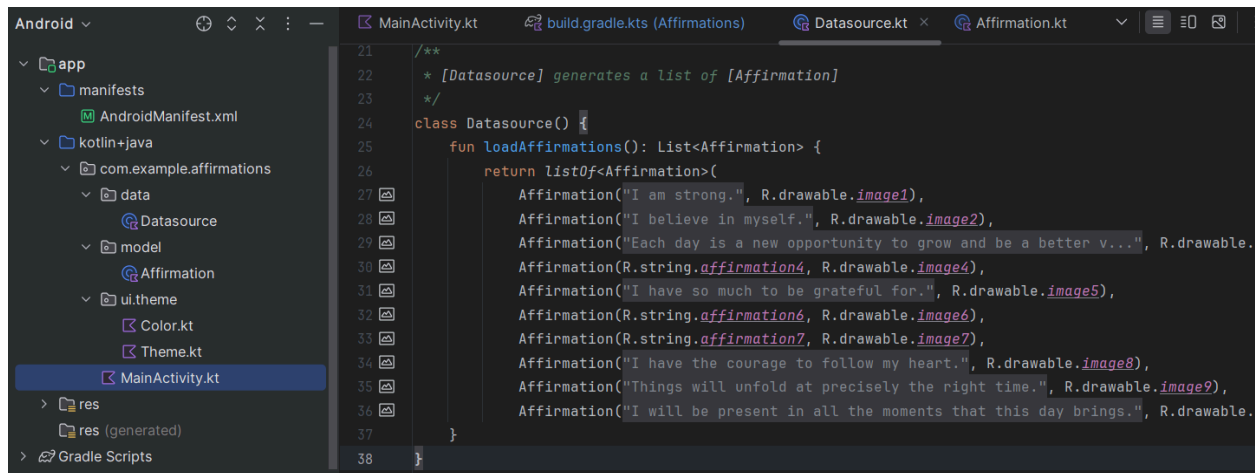
Lazy Column/Row y Cards

Existen también un componentes que nos permite navegar deslizando con el dedo hacia abajo o hacia un lado:

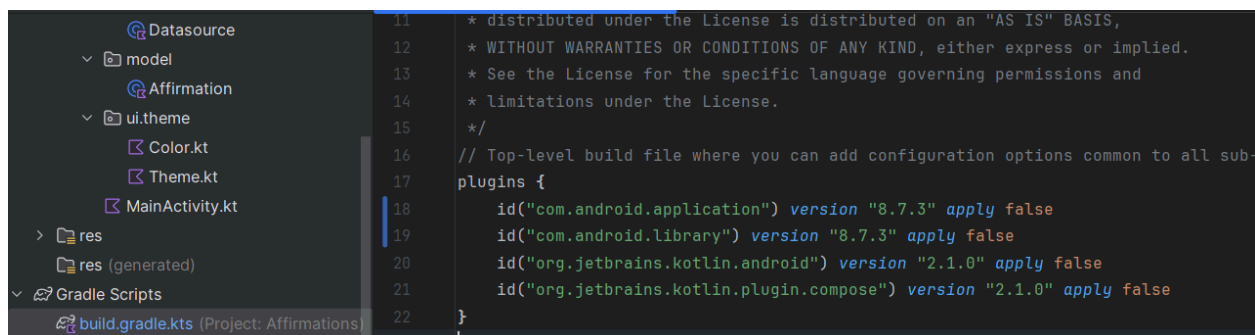
<https://developer.android.com/develop/ui/compose/lists?hl=es-419>

Para que veais esto en acción, clonad el siguiente repositorio:

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-affirmations/tree/starter>



Puede daros un problema con el Gradle, con cambiar la versión debería funcionar aunque igual tarda en descargarlo.



En ese proyecto veréis que la información está organizada en un nuevo elemento llamado **Card**.

```
Card(modifier = modifier) {  
    Column {  
        Image(  
            painter = painterResource(affirmation.imageResourceId),  
            contentDescription = stringResource(affirmation.stringResourceId),  
            modifier = Modifier  
                .fillMaxWidth()  
                .height(194.dp),  
            contentScale = ContentScale.Crop  
        )  
        Text(  
            text = LocalContext.current.getString(affirmation.stringResourceId),  
            modifier = Modifier.padding(16.dp),  
            style = MaterialTheme.typography.headlineSmall  
        )  
    }  
}
```

El elemento componible [Card](#) actúa como un contenedor de Material Design para tu IU. Por lo general, las tarjetas presentan **un solo elemento de contenido coherente**. Los siguientes son algunos ejemplos de cuándo podrías usar una tarjeta:

- Un producto en una app de compras
- Una noticia en una app de noticias
- Un mensaje en una app de comunicación.

Es el enfoque en representar una sola pieza de contenido lo que distingue a Card de otros contenedores. Por ejemplo, Scaffold proporciona la estructura general a toda una pantalla. Por lo general, la tarjeta es un elemento de la IU más pequeño dentro de un diseño más grande, mientras que un componente de diseño, como Column o Row, proporciona una API más simple y genérica.