Tema 3.1 - Conceptos de Jetpack Compose

Descargar estos apuntes pdf o html

Índice

Introducción

Antes de la introducción de Compose, las aplicaciones de Android se construían completamente utilizando Android Studio junto con una colección de frameworks asociados que conforman el **Android Development Kit**.

Para ayudar en el diseño de las interfaces de usuario, Android Studio **incluye** una herramienta llamada **Layout Editor**, que permite crear **archivos XML** que contienen los componentes individuales que conforman una pantalla de la aplicación.

La disposición de la pantalla se diseña arrastrando componentes desde una paleta de widgets a la ubicación deseada en el lienzo de diseño. Esto es, usando una herramienta de diseño visual RAD (Rapid Application Development) que genera el código XML.

Finalmente, los componentes que necesitan responder a eventos del usuario, se conectan a métodos en el código fuente de la aplicación donde se maneja el evento.

> **Importante**
>
> A lo largo del proceso de desarrollo, es necesario compilar y ejecutar la aplicación en un simulador o dispositivo para probar que todo funciona como se espera.

En contraposición, **Compose** introduce una **sintaxis declarativa** que simplifica la creación de diseños de interfaz de usuario en Android. En lugar de diseñar manualmente los detalles de diseño, Compose permite **describir cómo debe verse la interfaz sin preocuparse por la complejidad de su construcción**. Se declaran componentes, se especifica el tipo de diseño y se aplican atributos mediante modificadores.

Ejemplo: Puedes declarar "*Quiero una lista con estos elementos en ella*" en lugar de "*Quiero un RecyclerView con un adaptador y un ViewHolder que se vea así y que se comporte así*".

Compose se encarga automáticamente de la disposición y renderización. Además, Android Studio ofrece **vista previa en tiempo real** y un **modo interactivo** para probar la aplicación sin necesidad de compilar y ejecutar en un dispositivo o simulador.

Compose es **data-driven** (*orientado a datos*), esto no significa que ya no debamos manejar eventos generados por el usuario en la interfaz. La característica **data-driven** se refiere, más bien, a cómo Compose gestiona la relación entre los datos de la aplicación y su interfaz y lógica.

Antes de Compose, las aplicaciones de Android requerían código para verificar constantemente los cambios en los datos y actualizar la interfaz de usuario en consecuencia. Esto complicaba el desarrollo cuando varios componentes de la aplicación dependían de los mismos datos. Compose simplifica esto al

basarse en el **state** (*estado*), lo que significa que los **cambios en el estado de los datos se reflejan automáticamente en la interfaz**. Los componentes de la interfaz que utilizan este estado se actualizan automáticamente cuando cambian los datos, en un proceso llamado **recomposition** (*recomposición*).

Otras ventajas principales de Jetpack Compose:

- Menos código para construir interfaces.
- Código mucho más intuitivo.
- Facilidad a la hora de reutilizar componentes.
- Programación de vistas en Kotlin.

> **Resumen**
>
> Jetpack Compose es una nueva herramienta de Android para crear interfaces de usuario. Con **Jetpack Compose** (a.k.a. Compose), puedes definir tu **interfaz de usuario (UI) de forma declarativa**, es decir, describiendo cómo debería ser la interfaz de usuario en lugar de escribir el código que la crea.
> Además, Compose es **orientado a datos** porque actualiza automáticamente la interfaz de usuario cuando cambian los datos.

- **Recursos adicionales**:
  - Documentación oficial: **El paradigma de programación declarativa**.

Conceptos básicos de Compose

Composable functions

En Compose, las interfaces de usuario se crean con **funciones anotadas con @Composable**. Estas funciones son funciones especiales de Kotlin que se utilizan para crear interfaces de usuario al trabajar con Compose.

Podemos decir que las funciones componibles transforman los datos en elementos de la interfaz de usuario.

Sintaxis de una función *composable*:

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

> **Importante**
>
> Fíjate que el identificador va en **PascalCasing**. En Jetpack Compose, los nombres de las funciones marcadas con `@Composable` que emiten UI deben usar **PascalCasing** (**No retornan nada**). Sin embargo, aquellas funciones marcadas con `@Composable` que no emiten UI deben usar **camelCase** (Retornan algún tipo).

Además tienen otra serie de características que las diferencian de las funciones '*normales*' de Kolin:

- **No retornan valores** en el sentido tradicional de la función de Kotlin, sino que **emiten elementos de la interfaz de usuario** al sistema de tiempo de ejecución de Compose para su representación.
- **No pueden ser llamadas desde funciones *normales***. Anunque las funciones '*composables*' pueden llamar a otras funciones '*composables*' para **crear una jerarquía de componentes** el proceso inverso no es posible.
- **Idempotencia**
  - Se comporta de la misma manera cuando se la llama varias veces con el mismo argumento y no usa otros valores como variables globales o llamadas a `random()`.
  - **No pueden tener efectos secundarios**. Esto significa que no pueden **modificar variables fuera de su alcance**. Por ejemplo, no pueden modificar variables globales, ni modificar el estado de un ViewModel, ni modificar el estado de un Fragment o Activity.
- **Pueden aceptar parámetros**, lo que permite que la lógica de la app describa la IU
- **Pueden ejecutar en cualquier orden**.

```
@Composable
fun ButtonRow() {
    MyFancyNavigation {
        StartScreen()
        MiddleScreen()
        EndScreen()
    }
}
```

Las llamadas a `StartScreen`, `MiddleScreen` y `EndScreen` pueden ocurrir en cualquier orden. Eso significa que no puedes, por ejemplo, hacer que StartScreen() establezca alguna variable global (un efecto secundario) y que MiddleScreen() aproveche ese cambio. Esto es porque Compose tiene la opción de reconocer que algunos elementos de la IU tienen mayor prioridad que otros, y los dibuja primero.

- **Se pueden ejecutar en paralelo** y eso es una de las razones de no producir efectos secundarios que hemos mencionado.
- **No pueden lanzar excepciones**.
- **No pueden ser recursivas**.
- **No pueden ser privadas**.

Arquitectura de Compose

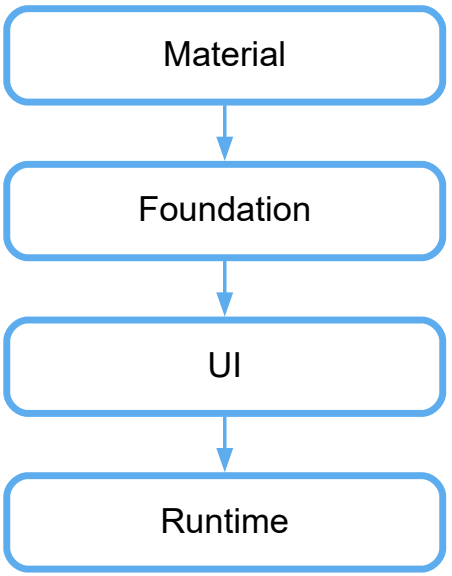
Extraído de Documentación oficial: **Arquitectura de Compose**.

Jetpack Compose no es un proyecto monolítico único, sino que se crea a partir de varios módulos que se ensamblan para formar un '***stack***'. Si comprendes los diferentes módulos que componen Jetpack Compose podrás:

- Usar el nivel adecuado de abstracción para compilar tu app o biblioteca.
- Comprender cuándo puedes "bajar" a un nivel inferior para tener más control o personalización.
- Minimizar tus dependencias.

Cada capa se compila sobre los niveles inferiores y se combinan las funcionalidades para crear componentes de nivel superior. Cada una de ellas toma como fundamento las API públicas de las capas inferiores para verificar los límites del módulo y permitirte reemplazar cualquiera de ellas si es necesario. Examinemos estas capas desde abajo hacia arriba.

1. **RunTime**: En este módulo se presentan los **aspectos básicos del entorno de ejecución de Compose**, por ejemplo, `remember`, `mutableStateOf`, la anotación `@Composable` y `SideEffect`.
2. **UI**: La capa de la IU consta de varios módulos (ui-text, ui-graphics, ui-tooling, etc.). Estos módulos implementan los aspectos básicos del **kit de herramientas de la IU**, como LayoutNode, `Modifier`, controladores de entrada, diseños personalizados y dibujos.
3. **Foundation**: En este módulo se proporcionan bloques de compilación agnóstica del sistema de diseño para la IU de Compose, como `Row` y `Column`, `LazyColumn`, el reconocimiento de gestos determinados, etc.
4. **Material**: En este módulo se proporciona una implementación del sistema **Material Design para la IU** de Compose y un **sistema de temas**, componentes de diseño, indicadores de ondas e íconos.



El estado y Jetpack Compose `State`

Recursos adicionales:

- Documentación oficial: **El estado en Jetpack Compose**.
- Android Developers: Vídeo **Jetpack Compose: State**.
- DevExperto: Vídeo **El estado en Jetpack Compose**.
- Martin Kiperszmid: Vídeo **¿Cómo funciona el Estado en Jetpack Compose?**.
- AristiDevs: Vídeo **Estados en Jetpack Compose**.
- Stvedza-San: Vídeo **States - Jetpack Compose**.

El estado es un **concepto fundamental** en Jetpack Compose. En lenguajes declarativos como Compose, el estado se refiere generalmente como ***"un valor que puede cambiar con el tiempo***".
Es por eso que el estado es la **fuente única de verdad** (*single source of truth*) de la interfaz de usuario. Cuando el estado cambia, la interfaz de usuario se actualiza automáticamente para reflejar el nuevo estado.

El estado **difiere de una variable estándar en dos formas significativas**.

1. El valor asignado a una variable de estado en una función *composable* debe ser **recordado** a la hora de redibujar. Esto es diferente de una variable estándar que se volvería a inicializar cada vez que se hace una llamada a la función en la que se declara por lo que tomaría siempre el valor inicial asignado al redibujar
2. La segunda diferencia clave es que un cambio en cualquier variable de estado tiene implicaciones de gran alcance para todo el árbol jerárquico de funciones *composables* que conforman una interfaz de usuario. Para entender por qué esto es así, ahora necesitamos hablar sobre la **recomposición**.

Recomposición

- Documentación oficial: **Recomposición en Jetpack Compose**.
- Stvedza-San: Vídeo **Recomposition - Jetpack Compose**.

Al desarrollar con Compose, construimos aplicaciones creando **jerarquías de funciones *'composables'***. En la mayoría de los casos, los datos pasados de una función *composable* a otra se habrán declarado como una variable de estado en una función principal. Esto significa que cualquier cambio en el valor de estado en una función componible principal deberá reflejarse en cualquier función componible secundaria a la que se haya pasado el estado.

Compose aborda esto realizando una operación denominada recomposición y por tanto esta ocurre cada vez que cambia un valor de estado dentro de una jerarquía de funciones componibles.

La recomposición simplemente significa que la función *composable* que recibe el estado se llama de nuevo y se le pasa su nuevo valor que será recordado en futuras recomposiciones.

Volver a componer todo el árbol componible para una interfaz de usuario cada vez que cambia un valor de estado sería un enfoque altamente ineficiente para representar y actualizar una interfaz de usuario. Compose evita esta sobrecarga utilizando una técnica llamada recomposición inteligente que implica recomponer solo aquellas funciones directamente afectadas por el cambio de estado. En otras palabras, solo las funciones que leen el valor de estado se recompondrán cuando cambie el valor.

> **Resumen**
>
> La recomposición es el proceso de volver a ejecutar una función *composable* para actualizar la interfaz de usuario. Esta se produce cuando el estado cambia. Si esto sucede, el sistema de tiempo de ejecución de Compose vuelve a ejecutar la función *composable* que usa el estado. La función *composable* vuelve a calcular la interfaz de usuario y el sistema de tiempo de ejecución de Compose actualiza la interfaz de usuario para reflejar el nuevo estado.

El objeto `State<T>`

El **objeto estado** `State<T>` es un **objeto observable** que se puede observar desde el sistema de tiempo de ejecución de Compose. Como hemos comentado al explicar la recomposición, cuando el estado cambia, el sistema de tiempo de ejecución de Compose **vuelve a ejecutar la función *'composable'*** que lo usa. Si queremos que el valor del estado pueda cambiar usaremos `MutableState<T>`

> **Importante**
>
> Las funciones *'composables'* pueden usar la API de `remember` para almacenar un objeto en la memoria. Un valor calculado por remember se almacena en la composición durante la composición inicial, y **el valor almacenado se muestra durante la recomposición**.

Existen varias formas de declarar un objeto `MutableState<T>` en un elemento que admite composición pero nosotros básicamente usaremos dos:

```
val mutableState = remember { mutableStateOf(default) }
```

Ejemplo:

El siguiente código muestra un botón que cuenta el número de veces que se ha pulsado.

```
@Composable
fun Contador() {
    // cuenta es un MutableState<Int> con valor inicial 0
    val cuenta : MutableState<Int> = remember { mutableStateOf(0) }
    // Para acceder al valor de cuenta usamos la propiedad value
    Button(onClick = { cuenta.value++ }) {
        Text("Llevas $cuenta.value Clicks")
    }
}
```

La forma que vamos a usar en la mayoría de los casos es usando la sintaxis de delegación `by`:

```
var value by remember { mutableStateOf(default) }
```

Si reescribimos el ejemplo anterior quedaría así:

```
@Composable
fun Contador() {
    // cuenta ahora es un Int con valor inicial 0
    var cuentaState by remember { mutableStateOf(0) }
    // Ya no necesitamos acceder a value
    Button(onClick = { cuentaState++ }) {
        Text("Llevas $cuentaState Clicks")
    }
}
```

Aunque usemos un `Int` en el fondo es un estado que cuando estamos cambiando su valor se llama al delegado `setValue()` es por eso que hemos puesto el sufijo `State` para tener claro que es un estado aunque sea de tipo Int.

Cuidado el uso del by me obligará a tener los siguiente imports.

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
```

Stateful vs Stateless

- Documentación oficial: **stateful vs. stateless composables**.

Podemos resumir que un **composable stateful** es aquel que **almacena un estado** y un **composable stateless** es aquel que **no almacena un estado**. En el ejemplo del punto anterior el composable `Contador` es stateful porque almacena el estado de la cuenta.

> **Importante**
>
> Podemos resumir que salvo ciertos casos en los que necesitemos almacenar un estado, **la mayoría de los *'composables'* serán stateless**. Los componibles stateless son más fáciles de entender, probar y reutilizar. Los componibles stateful deben usarse con moderación y solo cuando sea necesario almacenar un estado.

Para conseguir que nuestras funciones composables sean stateless debemos usar el concepto de **elevación de estado**.

Elevación de estado (`state-hoisting`)

- Recursos adicionales:
  - Documentación oficial: **Introducción**.
  - Documentación oficial: **Elevación de estado en Jetpack Compose**.

La elevación de estado o state-hoisting es un patrón de diseño que consiste en **mover el estado de un *composable* a su padre**. Esto se hace para que el estado pueda ser compartido por varios '*composables*'.

Además de compartir el estado, el padre también se encarga de actualizar el estado y para ello debe pasar también una **función de actualización al hijo** o '*event handler*'. Esta función de actualización se ejecutará cuando el hijo necesite actualizar el estado y deberá declararse o definirse en el mismo nivel que el estado.

El estado elevado de esta manera tiene algunas propiedades importantes:

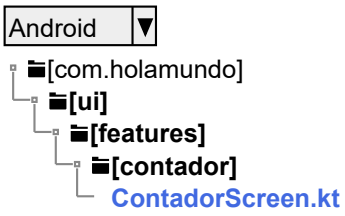
- **Fuente única de información**: Mover el estado en lugar de duplicarlo garantizará que exista solo una fuente de información. Eso ayuda a evitar errores.
- **Capacidad de compartir**: El estado elevado puede compartirse con varios elementos que admiten composición.
- **Capacidad de interceptar**: Los llamadores a los elementos componibles sin estado pueden decidir ignorar o modificar eventos antes de cambiar el estado.
- **Reutilización**: Los elementos componibles sin estado pueden reutilizarse en diferentes contextos sin tener que preocuparse por el estado. Estamos aplicando el principio de **bajo acoplamiento** donde un

elemento componible no necesita saber nada sobre donde es usado o en que jerarquía se está componiendo.

- **Pruebas**: Los elementos componibles sin estado son más **fáciles de testear** porque no tienen estado.

Ejemplo 1 de elevación de estado

Veamos un ejemplo sencillo de elevación de estado y el concepto de Stateful vs Stateless con nuestra aplicación que cuenta *'clicks'* de un botón.

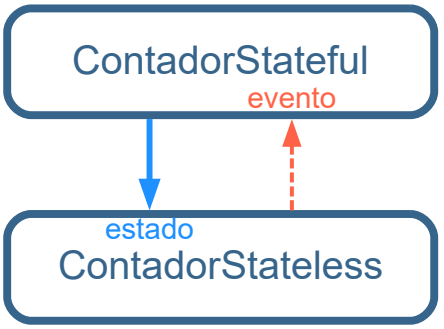


Para ello en nuestro proyecto `HolaMundo` hemos creado la jerarquía de paquetes propuesta en nuestra arquitectura y dento un fuente denominado **ContadorScreen.kt**.

EL código propuesto es el siguiente:

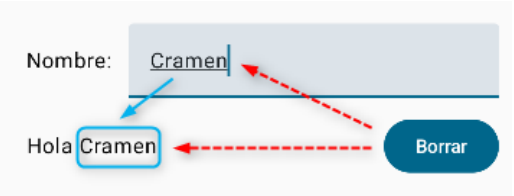
```
@Composable
fun ContadorStateful() {
    // Definimos el estado y el manejador del evento click
    var cuentaState by remember { mutableStateOf(0) }
    val onAumentarCuenta: () -> Unit = { cuentaState++ }

    // Pasamos el estado y el manejador al composable ContadorStateless
    ContadorStateless(
        cuentaState = cuentaState,
        onAumentarCuenta = onAumentarCuenta
    )
}
@Composable
fun ContadorStateless(
    cuentaState: Int,
    onAumentarCuenta: () -> Unit) =
    Column {
        // El botón llama al manejar del evento click
        // enviado por el padre
        Button(onClick = onAumentarCuenta) {
            Text("Cuenta Clicks")
        }
        // El texto muestra el estado enviado por el padre
        Text(text = "Llevas $cuentaState Clicks")
    }
```

Donde podemos ver que el `ContadorStateful` es el padre y el `ContadorStateless` es el hijo. El padre es stateful porque almacena el estado de la cuenta y el hijo es stateless porque no almacena el estado. Además, el hijo no puede modificar el estado, solo puede enviar un evento al padre para que este lo modifique.



Ejemplo 2 de elevación de estado

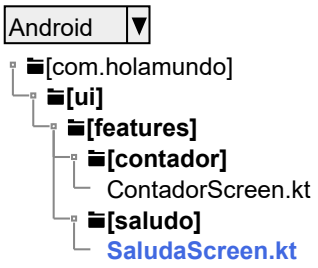
En este ejemplo vamos a crear una aplicación que nos permita introducir un nombre y nos salude. Además, vamos a añadir un botón para borrar el nombre introducido asignándole la cadena vacía. Esto es, mientras escribimos el nombre en el campo de texto, el texto del saludo se actualiza automáticamente, ya que ambos '*composables*' comparten el estado.

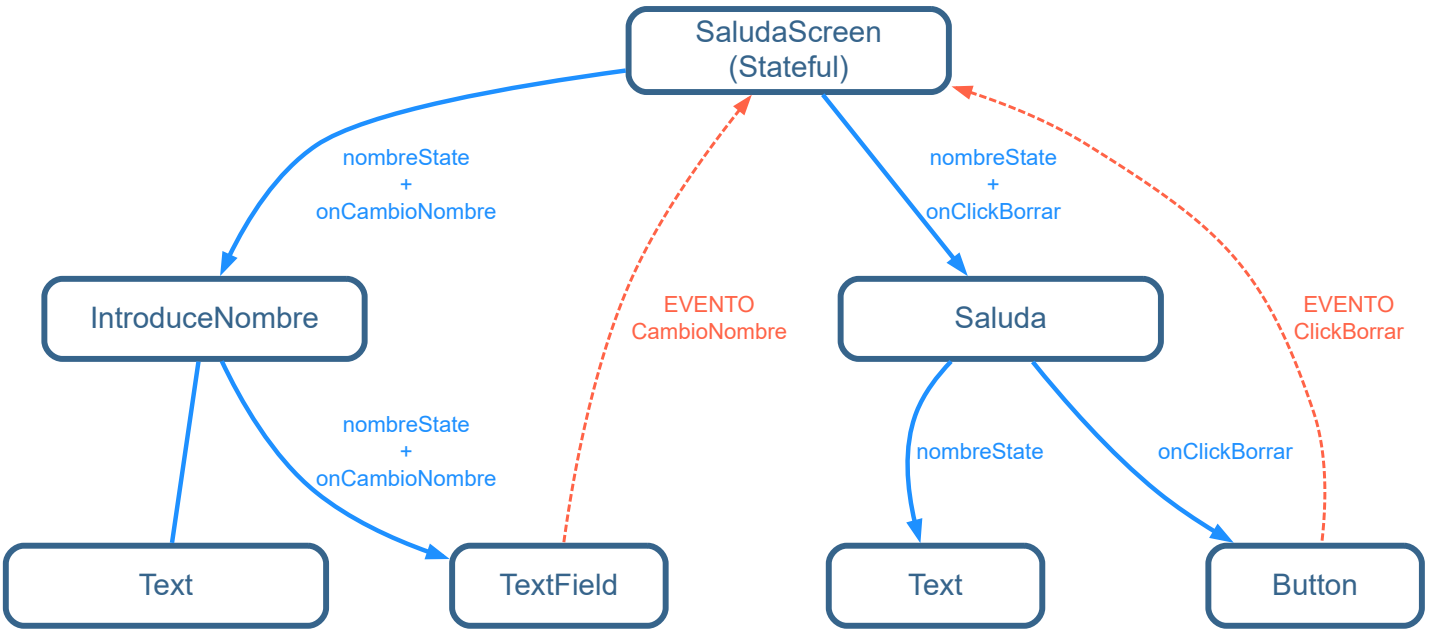


Para ello vamos a diseñar la una jerarquía de '*composables*' donde el estado lo tiene `SaludaScreen` y se va pasando a través de la jerarquía de composición.

Vamos a añadir a nuestro proyecto `HolaMundo` como en el ejemplo anterior un nuevo `feature` en el paquete `saludo` con el nombre **SaludaScreen.kt**.

Puedes ver la jerarquía de composición en el siguiente diagrama. De ella, deducimos que en cada recomposición se van a volver a redibujar la gran mayoría de los '*composables*' que la componen.





> **Nota**
>
> Con el fin de que el ejemplo sea algo más realista. En el código propuesto a continuación hemos usado layouts, modificadores y algunos componentes de Material Design que veremos en el siguiente tema. Nosotros **nos vamos a centrar en el concepto de elevación de estado**. Esto es, la jerarquía de componentes usada y como se va propagando el estado y los manejadores de eventos.

```
@Composable
fun SaludaScreen() {
    // Definimos el estado y los manejadores
    var nombreState by remember { mutableStateOf("") }
    val onCambioNombre = { nombre: String -> nombreState = nombre }
    val onClickBorrar = { nombreState = "" }

    Column(horizontalAlignment = Alignment.CenterHorizontally) {
        // Pasamos el estado y manejadores necesarios a los componentes
        IntroduceNombre(
            nombreState = nombreState,
            onCambioNombre = onCambioNombre
        )
        Saluda(
            nombreState = nombreState,
            onClickBorrar = onClickBorrar
        )
    }
}
@OptIn(ExperimentalLayoutApi::class)
@Composable
fun Saluda(
    nombreState: String,
    onClickBorrar: () -> Unit
) {
    FlowRow(
        Modifier.fillMaxWidth().padding(12.dp),
        horizontalArrangement = Arrangement.SpaceBetween
    ) {
        Text(
            modifier = Modifier.padding(12.dp),
            text = "Hola ${nombreState}"
        )
        // Elevamos el evento borrar
        Button(onClick = onClickBorrar) { Text(text = "Borrar") }
    }
}
```

```
@Composable
fun IntroduceNombre(
    nombreState: String,
    onCambioNombre: (String) -> Unit
) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Text(
            modifier = Modifier.padding(12.dp),
            text = "Nombre:"
        )
        TextField(
            value = nombreState,
            // Elevamos el evento cambio en Nombre
            onValueChange = onCambioNombre
        )
    }
}
```

Restableciendo el estado en Compose

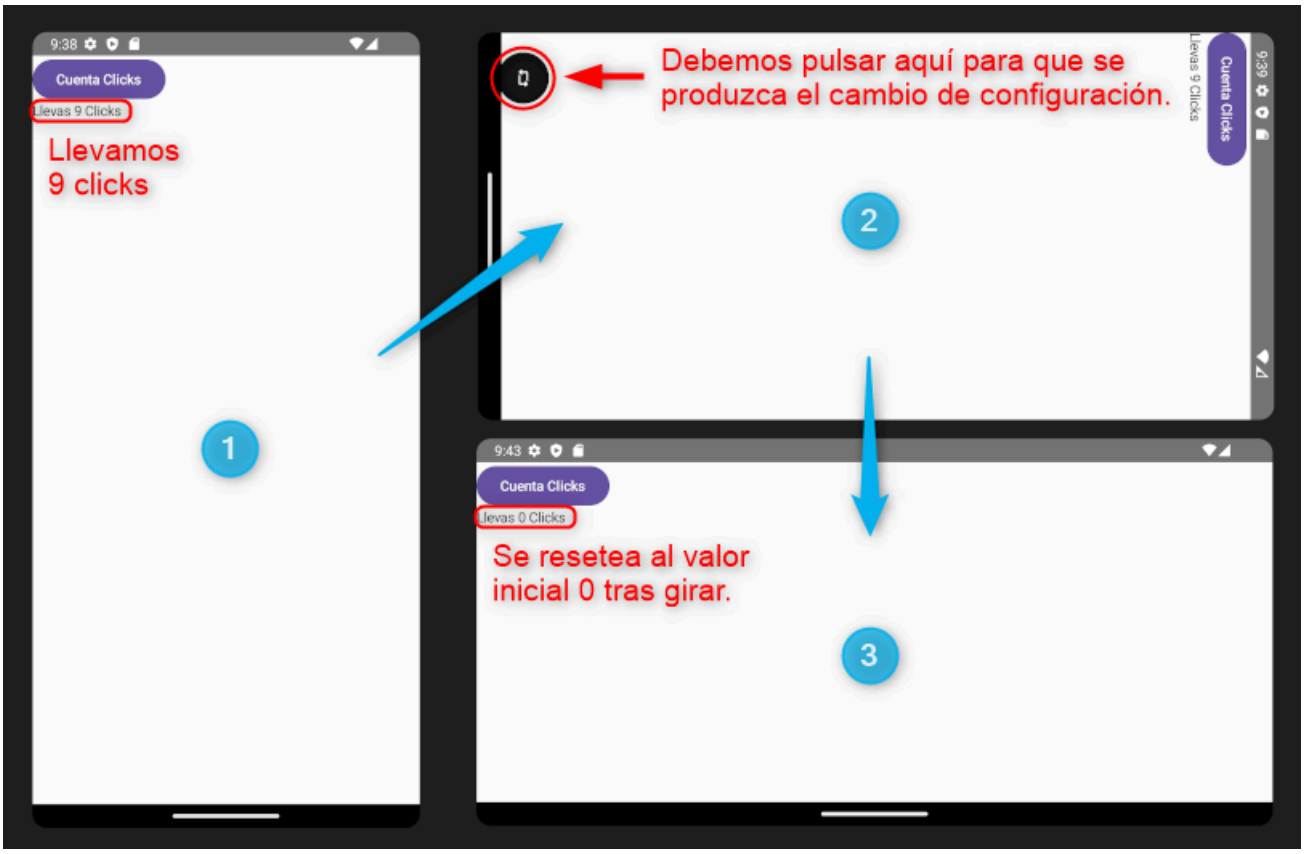
- Documentación oficial: **Cómo restablecer el estado en Compose**.

Aunque "remember" se puede utilizar para guardar valores de estado a través de recomposiciones. Sin embargo, esta técnica no retiene el estado entre **cambios de configuración**.

Cambiaremos la configuración cuando algún aspecto del dispositivo cambia de manera que altera la apariencia de una actividad (como rotar la orientación del dispositivo entre vertical y horizontal o cambiar la configuración de fuente en todo el sistema).

Estos cambios harán que toda la **actividad se destruya** y se vuelva a crear. El resultado es una actividad recién inicializada que ha perdido los valores de estado de la anterior configuración y estos volverán a tomar sus valores iniciales.

Por ejemplo, si ejecutemos en un dispositivo virtual nuestro programa del **contador** tendríamos una vista similar a la siguiente:



Si nos fijamos, al girar el dispositivo virtual, tras *'confirmar el giro'*, el contador se ha reiniciado a cero. Esto es porque se ha destruido la actividad y se ha vuelto a crear el árbol de composición.

Para evitar esto, podemos usar la API `rememberSaveable` que nos permite guardar el estado de un objeto en el sistema de tiempo de ejecución de Compose y restaurarlo cuando sea necesario.

Por tanto, reescribiremos nuestro ejemplo del contador usando `rememberSaveable`.

```
@Composable
fun ContadorStateful() {
    var cuentaState by rememberSaveable { mutableStateOf(0) }
    val onAumentarCuenta: () -> Unit = { cuentaState++ }

    ContadorStateless(
        cuentaState = cuentaState,
        onAumentarCuenta = onAumentarCuenta
    )
}
```

Sin embargo, esto solo funciona para los tipos simples. ¿Qué sucedería si el estado un objeto compuesto?. En ese caso, debemos usar la API `rememberSaveable` con un `Saver` personalizado tal y como se indica en la **documentación oficial**.

> **Importante**
>
> De todas formas, esto no nos tiene que preocupar salvo en los casos en que tengamos un '*composable stateful*' usado por otro composable. Esto se va a dar en muy pocos casos puesto que el estado, como ya veremos más adelante, se almacena en los `ViewModels`. En caso que esto suceda, se tratará de un tipo simple o uno ya definido por las librerías de Compose que ya es compatible con este API de persistencia.

Depurando nuestra composición

En ocasiones nos va a interesar depurar nuestra composición. Para ello podemos usar la herramienta **Layout Inspector** de Android Studio.

Como sería un poco largo incluir aquí la explicación de como usarla, os dejo un **enlace a la documentación oficial** donde se explica como usarla.

Gestion e estados Avanzada

Función `remember` profundización

- Documentación oficial: **remember con claves**.
- Diferencia entre remember con claves y derevedStateOf: **Vídeo Stevdza-San**
- Articulo relacionado en Medium: **DerivedStateOf vs Remember with keys: The difference**.

Como ya hemos visto, la función `remember{}` se utiliza para, como su nombre indica, recordar el estado en la composición. Esto ayuda a la función componible a recordar el valor anterior cuando se recomponga. Sin embargo, la función `remember` también puede tomar un número de argumentos conocidos como **Claves** (**Keys**).

Una ***'clave'*** puede ser cualquier **valor que pueda cambiar con el tiempo**. La función puede tomar más de una clave como argumento. En Compose, usamos estados para representar valores que pueden cambiar con el tiempo. Cuando cualquiera de las claves pasadas al bloque `remember` cambia de valor, el lambda final del bloque `remember` se vuelve a ejecutar. Esto es muy útil si se necesita recordar un estado y luego recalcularlo solo cuando cambia otro estado.

Ejemplo:

Vamos rehacer el ejemplo del contador para controlar los múltiplos de 3. En una primera aproximación podríamos usar dos estados '*normales*' para el contador y el booleano que me indica si es múltiplo de tres o no.

Esta vez por simplificar no hemos utilizado *'state hoisting'*.

```
@Composable
fun ContadorMultiploTres() {
    var cuentaState by remember { mutableStateOf(0) }
    var esMultiploDeTresState by remember { mutableStateOf(false) }
    val onAumentarCuenta: () -> Unit = {
            cuentaState++
            esMultiploDeTresState = cuentaState % 3 == 0
        }
    Column {
        Button(onClick = onAumentarCuenta) {
            Text("Cuenta Clicks")
        }
        Text(text = "Llevas $cuentaState Clicks")
        Text(text = (if (esMultiploDeTresState) "Es" else "No es") + " múltiplo de 3")
    }
}
```

Aunque el código anterior es funcional y correcto. Además, el segundo `Text` no se recompone a no ser que cambie el valor de `esMultiploDeTresState`. En manejador de `onAumentarCuenta` hace algo más de la *'responsabilidad'* que realmente tiene. Por lo que sería más correcto hacer...

```
@Composable
fun ContadorMultiploTres() {
    var cuentaState by remember { mutableStateOf(0) }
    var esMultiploDeTresState by remember(key1 = cuentaState) {
        mutableStateOf(cuentaState % 3 == 0)
    }
    val onAumentarCuenta: () -> Unit = { cuentaState++ }
    ...
}
```

Sin embargo, el ejemplo anterior aunque es más correcto aún sigue teniendo un problema. Al estar cambiado constantemente el valor de `cuentaState` se va a estar recalculando el valor de `esMultiploDeTresState` aunque no cambie. Por jemplo, para los valores de la cunea 0, 1 y 2 el valor de `esMultiploDeTresState` siempre va a ser `false`. Sin embrago, el composable que muestra si es múltiplo de tres o no se va a recomponer porque el estado se ha recalculado al cambiar la clave. Para evitar esto, podemos usar la función `derivedStateOf`.

> **Importante**
>
> La función `derivedStateOf` reduce el número de recomposiciones innecesarias respecto a `remember with Keys` ya que, esta última, se recompondrá aunque la clave que cambió tenga el mismo valor que tenía anteriormente y `derivedStateOf` no.

Estado derivado de otros estados `derivedStateOf`

- Documentación oficial: **dereivedStateOf**.

La función `derivedStateOf` nos permite crear un estado derivado de otros estados. Esto es, un **estado calculado** que depende del valor de otros estados.

As mentioned before, it is used to perform another important fuctionality — reducing unnecessary recompositions. In the case of remember the lambda gets recomputed even if the key that changed was set to the same value it was set previously

Continuación ejemplo ...

Vamos rehacer el ejemplo del contador para controlar los múltiplos de 3. Si usamos `derivedStateOf` podemos reescribir el código anterior de la siguiente forma...

```
@Composable
fun ContadorMultiploTres() {
    var cuentaState by remember { mutableStateOf(0) }
    val esMultiploDeTresState by remember {
        derivedStateOf { cuentaState % 3 == 0 }
    }
    val onAumentarCuenta: () -> Unit = { cuentaState++ }
    ...
}
```

de esta forma hasta que deriveStateOf no cambie su valor, no se va a recomponer el composable que usa el estado **`esMultiploDeTresState`**.

Previsualizar diseño de mis '*Composables*'

En Android Studio Disponemos de **diferentes herramientas** para asistirnos en el diseño de nuestros componentes de Compose. Una de ellas es la posibilidad de **previsualizar** el diseño de nuestros '*composables*' en diferentes dispositivos y orientaciones.

Es fácil de configurar a través de un dialogo de configuración y tienes la guía oficial de uso actualizada **aquí**.

Pero vamos hacer una pequeñas guías de uso a continuación...

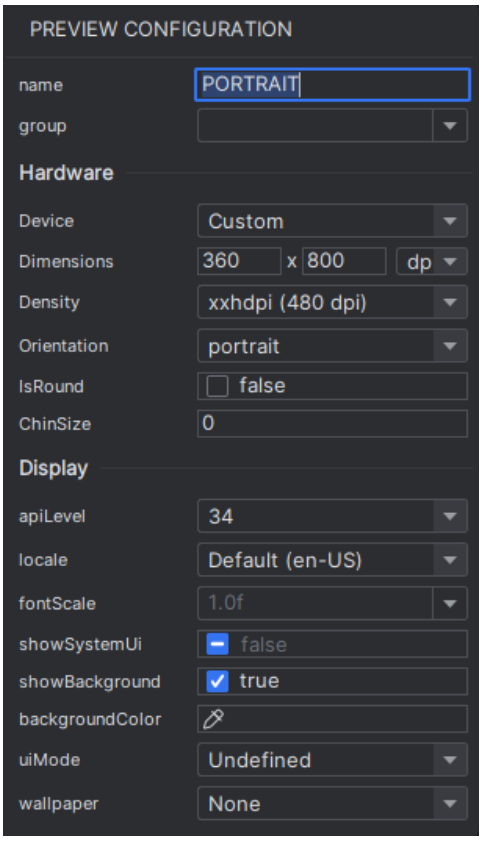
Vamos a partir del ejemplo **SaludaScreen.kt** que hemos usado en el ejemplo de elevación de estado.

1. La primera idea es que esta previsualización nos puede servir de ***'test'*** previo de nuestras vistas con **diferentes dispositivos** y **configuraciones de pantalla**.
   Añadiremos la anotación `@Preview` sobre el componente `composable` que queramos visualizar. Si es una **pantalla completa**, lo haremos sobre el componente que la contiene. En nuestro caso, sobre `SaludaScreen`.

   ```
   @Preview
   @Composable
   fun SaludaScreenPreview() {
       // Añadimos el Tema de la aplicación
       ProyectoBaseTheme {
           Surface {
               SaludaScreen()
           }
       }
   }
   ```

2. Al añadir la anotación nos aparecerá el símbolo ⚙ en la parte izquierda y al pulsarlo nos aparecerá un dialogo de configuración. Como el de la imagen a la derecha. En el que podemos seleccionar el dispositivo, su orientación, su densidad de pantalla, si queremos que se muestre el marco del mismo, se active el modo nocturno o incluso si queremos seleccionar un idioma para ver las traducciones.
   Podemos también en la opción `Device`, en lugar de `Custom (Personalizado)`, podemos elegir un dispositivo en concreto y nos configurará opciones de resolución y densidad de pantalla según sus características concretas.
   Por ejemplo, la configuración del diálogo me añadiría los siguientes parámetros a la anotación `@Preview`.

```
@Preview(
    name = "PORTRAIT",
    device = "spec:width=360dp,height=800dp,dpi=480",
    showBackground = true
)
```



3. Para futuros **test** del interfaz y su integración en MVVM es conveniente hace un `state hoisting` de todos mis estados hasta `SaludaScreenPreview` que debería ser el '*único*' componente stateful.

```
@Composable
fun SaludaScreen(
    nombreState: String,
    onCambioNombre: (String) -> Unit,
    onClickBorrar: () -> Unit) {
    ... // código omitido por abreviar
}

@Preview(
    name = "PORTRAIT",
    device = "spec:width=360dp,height=800dp,dpi=480",
    showBackground = true
)
@Composable
fun SaludaScreenPreview() {
    var nombreState by remember { mutableStateOf("") }
    val onCambioNombre = { nombre: String -> nombreState = nombre }
    val onClickBorrar = { nombreState = "" }

    ProyectoBaseTheme {
        Surface {
            SaludaScreen(
                nombreState = nombreState,
                onCambioNombre = onCambioNombre,
                onClickBorrar = onClickBorrar
            )
        }
    }
}
```

4. Podemos **añadir tantos '*previews*' como queramos**. Por ejemplo, podemos añadir un preview para la orientación **horizontal** o LANDSCAPE, **modo nocturno** e **idioma inglés**. De esta manera de una sola vez podremos ver todos los comportamientos de nuestra vista de una sola vez durante el diseño.

```
@Preview(
    name = "PORTRAIT",
    device = "spec:width=360dp,height=800dp,dpi=480",
    showBackground = true
)
@Preview(
    name = "LANDSCAPE",
    locale = "en",
    device = "spec:width=360dp,height=800dp,dpi=480,orientation=landscape",
    uiMode = Configuration.UI_MODE_NIGHT_YES,
    showBackground = true, fontScale = 1.0f
)
@Composable
fun SaludaScreenPreview() { ... }
```

5. Para evitar tener que añadir muchas anotaciones `@Preview` existen diferentes plantillas preview como `@PreviewScreenSizes`, `@PreviewFontScales`, `@PreviewLightDark`, y `@PreviewDynamicColors` dependiendo de la versión de la librería `androidx.compose.ui:ui-tooling-preview` que estemos usando.

```
@PreviewScreenSizes
@PreviewFontScale
@Composable
fun SaludaScreenPreview() { ... }
```

Decarga del código de ejemplo **SaludaScreenPreview.kt**.