Apuntes

Tema 4 . Fundamentos de Jetpack Compose

Descargar estos apuntes pdf o html

Índice

¿Qué es Jetpack Compose

Jetpack Compose es la nueva forma de programar interfaces de usuario propuesta por Google en 2019. Y en pleno funcionamiento en la actualidad.
Esta tecnología sigue la misma dirección que *Swift UI* y *Flutter* usando un paradigma declarativo.

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*.
-

¡Adiós *XML*!

La forma tradicional de trabajar con vistas *XML*, es decir, usando la programación imperativa, aumentaba la probabilidad de errores: olvidarse de actualizaciones de las vistas, estados ilegales…

En lugar de escribir *XML*, los desarrolladores pueden usar *Jetpack Compose* para crear interfaces de usuario utilizando una sintaxis declarativa y funciones composable.

Las funciones composable de *Jetpack Compose* se definen con la anotación `@Composable`.

La interfaz está totalmente interconectada con un estado, de tal forma que si un estado cambia, la interfaz se vuelve a pintar para representar ese nuevo estado.

Las funciones reciben datos por parámetros y emiten elementos de UI: `Text Button,…` que son a su vez funciones que emiten una vista.

Los elementos de la IU son jerárquicos, ya que unos contienen a otros. En *Compose*, compilas una jerarquía de la IU llamando a las funciones que admiten composición desde otras funciones del mismo tipo.

Este es un ejemplo sencillo que utiliza un `Text`:

```
@Composable
fun GreetingText(name: String) {
    Text(text = "$name!",
        modifier = Modifier
            .fillMaxWidth())
}
```

Cada interacción con la UI, por ejemplo un `onclick()` se actualiza o regenera la UI, a este efecto se le llama recomposición, y el sistema optimiza este proceso para que solamente se regeneren la vistas que cambian. Profundizaremos más tarde en esta optimización de las vistas.

Aquí te muestro un ejemplo que utiliza un `Text` y un `Button` (no codificar todavía este ejemplo en *Android Studio*):

```
    @Composable
2   fun MyScreen() {
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(16.dp),
7           horizontalAlignment = Alignment.CenterHorizontally,
8           verticalArrangement = Arrangement.Center
        ) {
10          Text(
                text = "¡Hola mundo!",
                style = MaterialTheme.typography.h3,
                modifier = Modifier
                    .padding(bottom = 16.dp)
                    .wrapContentWidth()
                    .clickable {
                        //Do something
                    }
19          )
20          Button(
                onClick = { /* acción al hacer clic */ },
                modifier = Modifier.fillMaxWidth()
            ) {
                Text(text = "Presionar")
25          }
        }
    }
```

> 📌 **NOTA**:
>
> - MyScreen es una función `@Composable` que define una columna de elementos (`Column`) que ocupa todo el espacio disponible (`fillMaxSize`) con un margen de *16dp* (`padding`). Se aplica también una alineamiento vertical y horizontal (líneas 7 y 8)

- Dentro de la columna, hay un `Text` que muestra el mensaje *"¡Hola mundo!"* con un estilo de fuente de encabezado 3 (`MaterialTheme.typography.h3`) y un margen inferior de *16dp* (`padding`). Con `.clickable` hemos tenido en cuenta que sobre ese texto podemos gestionar la pulsación, especificando las acciones que se tienen que ejecutar si se produce un *click*.
- También hay un `Button` que ocupa todo el ancho disponible (`fillMaxWidth`) y muestra el texto *"Presionar"*. Al hacer *clic* en el botón, se ejecutará la acción que se especifique en el parámetro `onClick` . Realmente es una función que se pasa como parámetro y que actúa como callback.

No preocuparos iremos viendo esto poco a poco.

Personaliar componentes con Modifier

Todos los elementos *Composable* que ofrece el *SDK de Android* aceptan un parámetro llamado `Modifier` .
`Modifier` es una clase estática a la que se puede acceder sin necesidad de ser instanciada y desde cualquier lugar de nuestra aplicación. Tiene funciones para especificar parámetros como la anchura, altura, el tamaño total, padding, etc, de un componente.

```
@Composable
fun GreetingText(name: String) {
    Text(text = "Hello $name!",
      modifier = Modifier.width(80.dp))
}
```

Los métodos de `Modifier` implementan *method chaining pattern* de forma que permiten concatenar varias llamadas a métodos en la misma cadena pudiendo establecer varios parámetros en una única expresión.

```
@Composable
fun GreetingText(name: String) {
    Text(text = "Hello $name!",
      modifier = Modifier
            .width(80.dp)
            .height(240.dp))
}
```

Como alternativa, usando el método `size` podemos establecer valores para la anchura y para la altura de un componente.

```
@Composable
fun GreetingText(name: String) {
    Text(text = "Hello $name!",
      modifier = Modifier
            .size(width = 80.dp, height = 240.dp))
}
```

Si no se indican los parámetros de `width` y `height` el mismo valor será aplicado para ambos parámetros haciendo que el componente sea cuadrado.

```
@Composable
fun GreetingText(name: String) {
    Text(text = "Hello $name!",
      modifier = Modifier
            .size(80.dp))
}
```

`fillMaxSize` permite al componente ocupar todo el espacio que ocupa su componente padre.

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

`fillMaxHeight` permite al componente ocupar todo el espacio en altura que ocupa su componente padre. La anchura se mantiene como `wrap_content`.

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

`fillMaxWidth` permite al componente ocupar todo el espacio en anchura que ocupa su componente padre. La altura se mantiene como `wrap_content`.

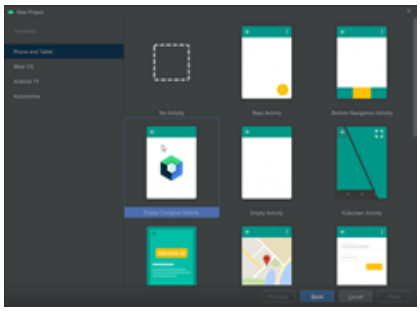
```
@Composable
fun GreetingText(name: String) {
    Text(text = "Hello $name!",
      modifier = Modifier
            .fillMaxWidth())
}
```

`fillMaxWidth` y `fillMaxHeight` aceptan como argumento fracciones (de 0 a 1) que indican el máximo espacio que queremos que ocupe nuestro componente dentro de su componente padre.

```
@Composable
fun GreetingText(name: String) {
    Text(text = "Hello $name!",
      modifier = Modifier
            .fillMaxWidth(0.5f))
}
```

Creando nuestro primer proyecto

Crear un proyecto en *Android Studio* usando *Jetpack Compose* es muy sencillo desde la versión estable de **Arctic Fox (2020.3.1 en adelante)**. Tan solo hay que crear un nuevo proyecto, y elegir la opción de *«Empty Compose Activity»*:

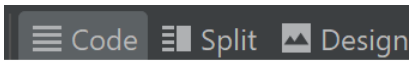


Introducimos el nombre *B4_PrimerProyecto* y nos genera entre otros archivos el *MainActivity.kt*:

```
package com.example.compose

import android.os.Bundle
...

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            B4_PrimerProyectoTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier=Modifier.fillMaxSize(),
                    color=MaterialTheme.colors.background
                ){
                     Greeting("World")
                }
            }
        }
    }
}

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

> 📌 **NOTA**:
>
> - La función **`setContent()`** que se usa es un poco especial, porque acepta un bloque de código. En este bloque es donde vamos a poder escribir nuestro código de *Compose*.
> - Podemos ver otra función **`ComposeTheme`** , que va a identificar el tema que va a usar nuestra aplicación (ya hablaremos sobre temas más adelante). Los temas en Jetpack Compose son un poco diferentes, porque están definidos también por funciones de *Compose* en vez de por *XMLs*.
> - A continuación vemos el primer componente de vista de *Compose*: **`Surface`** . Es uno de los distintos tipos de contenedores que ofrece *Jetpack Compose* para definir vistas.
> - Finalmente, la función **`Greeting()`** , que no es más que una función que visualiza un texto que se pasa por parámetro al invocar a la función **Text``**, que es una función *Compose* ofrecida por *Android*.

Existen tres modos de visualización



en *Android Studio* de forma que las funciones invocadas en la función `DefaultPreview()` puedan previsualizarse:

Code solo visualiza el código *Kotlin*, **Split** tanto el código como su vista, y **Design** solamente la vista que genera el código.

Si modificamos la llamada a Greeting, de la forma: `Greeting("Hola profe")` la vista se modificará cuando pulsemos el botón de refresco.

Hay que actualizar también la llamada a Greeting en `DefaultPreview()`.

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    B4_PrimerProyectoTheme {
        Greeting("World")
    }
}
```

Realizar las pruebas con `Modifier` del punto anterior.

> ✋ **Ejercicio resuelto**: Ejemplo de tres botones con distribución horizontal, el primero ocupa la mitad de la pantalla y los otros el resto del espacio de forma equitativa.

State en compose

El primer paradigma que debemos tener claro cuando se diseña vistas declarativas con *Compose* es el *State*.

Un componente *UI* es la combinación entre su representación gráfica(*View*) y su estado (*State*).

Toda aquella propiedad o dato que pueda cambiar en el componente *UI* puede ser representado como un estado.

El estado de una *app* es cualquier valor que puede cambiar con el paso del tiempo. Esta es una definición muy amplia y abarca desde una base de datos de *Room* hasta una variable de una clase.

Compose es declarativo y, por lo tanto, la única manera de actualizarlo es llamar al mismo elemento que admite composición con argumentos nuevos.

En consecuencia, elementos, como un `Text` que visualiza una variable que cambia su valor, no se actualizan automáticamente de la misma manera que en las vistas imperativas que se basan en *XML*. A un elemento que admite composición se le debe informar, de manera explícita, el estado nuevo para que se actualice según corresponda.

Vamos a explicar como funciona `State` mediante un ejemplo. Creamos un proyecto nuevo que llamaremos *CuentaSaludos*. Vamos a coger nuestra función que visualizaba `MyScreen()`, con una *Hola mundo* y el botón *Presionar*, de froma que visualizaremos en un `Text` la cantidad de veces que hemos pulsado el botón.

```
@Composable
fun MyScreen() {
    var cont:Int=0
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = "¡Hola mundo!",
            style = MaterialTheme.typography.h3,
            modifier = Modifier
                .padding(bottom = 16.dp)
                .wrapContentWidth()
        )
        Button(
            onClick = { cont++ },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text(text = "Presionar")
        }
        Text(
            text = "$cont",
            style = MaterialTheme.typography.h4,
            modifier = Modifier
                .padding(bottom = 16.dp)
                .wrapContentWidth()
        )
    }
}
```

> 📌 **NOTA**:
>
> - **Línea 3**: definimos la variable que se visualiza en el `Text()`
> - **Línea 19**: en el `onClick()` del `Button` incrementamos el valor de la variable.

- **Líneas 2 a 30**: incorporamos un nuevo elemento de vista `Text()` para la visualización de la variable `text = "$cont"`

Si ejecutamos nuestro programa verás que no sucede nada. `Text()` no se actualiza a sí mismo cuando cambia el valor de la variable *cont*. Tiene que ver con la manera en que funcionan la composición y la recomposición en *Compose*.

Para poder aplicar el `State` es necesario combinar dos conceptos: `remember` y `mutableStateOf`.

- `mutableStateOf(value)` crea un `MutableState`, que es un tipo observable en *Compose*. Cualquier cambio en su valor programará la recomposición de las funciones que admiten recomposición que lean ese valor.
- `remember` almacena objetos en la composición y los olvida cuando se quita de ella el elemento componible que llamó a remember.
- `rememberSaveable` conserva el estado en los cambios de configuración guardándolo en un `Bundle` permitiendo que en caso de rotación de la pantalla la información no se pierda..

Existen diferentes maneras de declarar los objetos `MutableState`.

Cuando usamos, por ejemplo: `val cont = remember{ mutableStateOf(0)}` significa que no solo estamos observando los datos sino que también persistiendo los datos a lo largo de la recomposición.

Un enfoque más común y conciso para declarar el estado es usar delegados de propiedades de Kotlin a través de la palabra clave `by` de la siguiente manera (ten en cuenta que se deben importar dos bibliotecas adicionales al usar delegados de propiedad).

```
val cont by remember{ mutableStateOf(0)}
```

Utilizaremos siempre este segundo enfoque, entre otras cosas porque obtenemos una variable del tipo concreto del que queremos controlar el estado y esto facilita mucho la legibilidad del código sin necesidad de hacer referecia a variables `MutableState`.

> 📌 **NOTA**:
> Existe una tercera técnica para declarar el estado, es a través de una pareja valor y función *setter*.
> `val (cont,setCont) remember{ mutableStateOf(0)}` , tenedlo en cuenta por si se encuentra código que trate el estado de esta forma.

Incorporemos estos conceptos a nuestro código.

```
    @Composable
    fun MyScreen() {
3       var cont by remember{ mutableStateOf(0) }
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(16.dp),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        ) {
            Text(
                text = "¡Hola mundo!",
                style = MaterialTheme.typography.h3,
                modifier = Modifier
                    .padding(bottom = 16.dp)
                    .wrapContentWidth()
            )
            Button(
19              onClick = { cont++ },
                modifier = Modifier.fillMaxWidth()
            ) {
                Text(text = "Presionar")
            }
            Text(
25              text = cont.toString(),
                style = MaterialTheme.typography.h4,
                modifier = Modifier
                    .padding(bottom = 16.dp)
                    .wrapContentWidth()
            )
        }
    }
```

📌 **NOTA**:

- **Línea 3**: aplicamos *State* a la variable que actúa como contador y que se visualizará en el `Text()`
- **Línea 19**: incrementamos su valor.
- **Línea 25**: para su visualización convertimos a *String* la variable con el método `.toString()` .

Patrón State Hoisting

A las vistas que contienen o dependen de un estado se les denomina *Stateful Views* y aquellas vistas que carecen de estado se las conoce como *Stateless Views*.

Google, recomienda como buena práctica de diseño, utilizar vistas de tipo *stateless*, ya que son vistas que pueden ser reutilizads y permiten delegar el manejo de estado a otros componentes.

De acuerdo a dichas recomendaciones, es importante convertir las vistas de tipo *statefull* a vistas de tipo *stateless*.

¿Esto cómo se logra? aplicando el patrón *State Hoisting*.

El patrón State Hosting consiste precisamente en mover los estados al componente padre de tal forma que los hijos nunca tengan que manejarlos.

El principal objetivo es reemplazar la variable de estado por dos argumentos en cada función composable hija:

- `value: T`, el valor para mostrar.
- `onValueChange: (T) -> Unit`, evento *(lambda)* que dispara la modificación del `State`.

El patrón *State Hosting* ofrece las siguientes ventajas:

- Manejar los estados de forma única y centralizada.
- Solo las funciones que manejan estados pueden modificarlos.
- Funciones composable hijas no tienen que preocuparse por manejar estados, solo pintar información y elevar eventos.

Vamos a explicar como aplicar este patrón al ejemplo de la aplicación *Hola Mundo* con un `Button` y un `Text` que visualiza el número de veces que se ha presionado. Creamos un nuevo proyecto que se llame *CuentaSaludoSH*.

En primer lugar vamos a cambiar la función `onCreate` de la clase `MainActivity`, incorporando una llamada a la función composable `MyApp()` que actuará commo padre de `MyScreen()`. Lo que estamos haciendo es convertir `MyApp()` en una función *statefull* y a `MyScreen()` en una función tipo *stateless*.

Veamos el código.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeTheme {
                // A surface container using the 'background' color from the theme
                MyApp()
            }
        }
    }
}

@Composable
fun MyApp(){
    MyScreen()
}
```

Apliquemos el patrón *State Hoisting* tal y como lo hemos explicado anteriormente.

```
    @Composable
    fun MyApp(){
3       var cont by remember { mutableStateOf(0)}
        MyScreen(cont){
            cont++
        }
    }
```

> 📌 **NOTA**:
>
> - **Línea 3**: aplicamos *State* a la variable contador, iniciándola con el valor *0*. Trasladamos su definición del hijo al padre.
> - Implementamos entre *{}* la función *lambda* que se pasa como parámetro y que recoge la funcionalidad del botón. Trasladamos la funcionalidad del hijo al padre.

La función `MyScreen()` ahora quedaría así:

```
@Composable
fun MyScreen(
    cont: Int,
    onButtonPresionar:()->Unit) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = "¡Hola mundo!",
            style = MaterialTheme.typography.h3,
            modifier = Modifier
                .padding(bottom = 16.dp)
                .wrapContentWidth()
        )
        Button(
            onClick = onButtonPresionar,
            modifier = Modifier.fillMaxWidth()
        ) {
            Text(text = "Presionar")
        }
        Text(
            text = cont.toString(),
            style = MaterialTheme.typography.h4,
            modifier = Modifier
                .padding(bottom = 16.dp)
                .wrapContentWidth()
        )
    }
}
```

> 📌 **NOTA**:
>
> - **Línea 3 y 4**: aplicamos patrón *State Hoisting* pasando como parámetros la variable que se ve afectada para la recomposición (contador) y la función que se ejecutará al pulsar el botón.
> - **Línea 20**: asignamos al `onClick` del botón la función que se pasa por parámetro.

No siempre es necesario aplicar este patrón. El estado puede ser interno en un elemento componible cuando ningún otro elemento necesita controlarlo.

Posteriormente, cuando expliquemos el elemento `Card`, veremos un ejemplo de esta situación.

Ejercicio Propuesto.
Renombrar el botón *Presionar* y colocarle la propiedad `text` del `Text()` como *Incrementar*.

Añadir a la vista un `Button` *Decrementar* y aplicar patrón *State Hoisting* para que actualizar convenientemente el contador.

Column, Row y Box

En *Compose* existen varios tipos de *layout* que podemos utilizar para construir nuestras interfaces de usuario. A continuación te menciono algunos de los más comunes:

- `Column` : es un *layout* que organiza los elementos de forma vertical, uno debajo del otro. Puede ser utilizado para crear listas de elementos, por ejemplo.
- `Row` : es similar a `Column` , pero organiza los elementos de forma horizontal, uno al lado del otro.
- `Box` : es un *layout* que se utiliza para agrupar elementos. Es útil cuando queremos aplicar un efecto a varios elementos a la vez, como por ejemplo aplicar un padding o una sombra.

> 😲 **NOTA**: Otros tipos de contenedores:
>
> - `Card` :permite crear una vista de tarjeta, como las que se suelen usar en listas de contenido en muchas aplicaciones móviles. Se utiliza para dar una sensación de profundidad y separar visualmente una sección del resto de la interfaz de usuario.
> - `Spacer` : es un *layout* que simplemente añade un espacio en blanco entre dos elementos.
> - `LazyColumn` : es un *layout* que se utiliza para manejar grandes listas de elementos de forma eficiente. Los elementos son creados de manera perezosa, es decir, sólo se crean los elementos que son visibles en pantalla.
> - `LazyRow` : es similar a `LazyColumn` , pero organiza los elementos de forma horizontal.
> - `ScrollableColumn` : es un layout que nos permite crear una columna que sea scrollable. Es útil cuando queremos mostrar una lista de elementos que ocupe toda la pantalla.
> - `ScrollableRow` : es similar a `ScrollableColumn` , pero organiza los elementos de forma horizontal.

Vamos a crear un proyecto nuevo que llamaremos *Bloque4Compose* donde iremos viendo los ejemplos.

Ejemplo de Column

Vamos a definir una función `@Composable` que llamaremos *ChefColumnScreen* que incluya la imagen de un cocinero, su nombre y su dirección de correo. Aplicaremos al diseño unos pequeños efectos para hacerlo más atractivo. El resultado que queremos es el siguiente:

Para empezar, y siguiendo la arquitectura de aplicaciones que hemos visto en el tema anterior, vamos a crear una estructura de carpetas *\ui\feature\chefcolumn*, donde almacenaremos la pantalla correspondiente a este *componente*.

Creamos el archivo *ChefColumnScreen.kt* e incluimos el siguiente código:

```
@Composable
fun ChefColumnScreen(
    foto: Bitmap?,
    name:String,
    mail:String){

    //remember permite conservar y recordar el estado de un valor garantizando que el est
    var fotoDefectoState = rememberVectorPainter(image = Icons.Filled.Face)

10  Column(
        modifier= Modifier.padding(20.dp),
12      verticalArrangement = Arrangement.Center,
13      horizontalAlignment = Alignment.CenterHorizontally) {
14      Image(
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .size(75.dp)
18              .clip(CircleShape),
19          painter = foto?.asImageBitmap()?.let { BitmapPainter(it) }
                ?: fotoDefectoState,
            contentDescription = "Imagen de ejemplo")
        Text(
            modifier = Modifier
                .padding(5.dp),
            text = name,
            style = MaterialTheme.typography.titleSmall
        )
        Text(
            modifier = Modifier
                .padding(5.dp),
            text = mail,
            color= Color.Red,
            style = MaterialTheme.typography.bodySmall
        )
    }
}
```

📌 **NOTA**:

- **Línea 10**: definimos la columna con `Column` , dentro de ella se definen los elementos que queremos incorporar dentro de la misma: una `Image` y dos `Text`
- **línea 12**: Utilizamos `verticalArrangement = Arrangement.Center` para especificar que los hijos se distribuyan verticalmente de forma centrada.
- **línea 13**: Utilizamos `horizontalAlignment = Alignment.CenterHorizontally` para que todos los hijos tengan la misma disposición de alineación respecto al padre.
- **Línea 14**: definimos la imagen. La propiedad `contentScale` permite controlar cómo se escala y se posiciona el contenido de la imagen en su contenedor. `ContentScale.Crop` es

uno de los valores posibles para esta propiedad, y se usa para recortar la imagen en caso de que su tamaño no se ajuste perfectamente al contenedor.

- **Línea 18**: aplicamos un efecto de redondeo a la imagen.
- **Línea 19**: verifica si `foto` es nulo utilizando el operador de llamada segura `?` . y luego utiliza la función `let` para crear un `BitmapPainter` si `foto` no es nulo. Si `foto` es nulo, se asigna el valor de respaldo `fotoDefectoState` al painter.
- Incluimos los `Text` que visualizarán el nombre y el mail del chef.

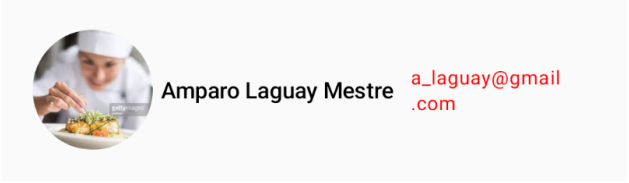
Ejemplo de Row

Vamos a definir el mismo ejemplo que antes pero con una fila, la función la llamaremos *ChefRowScreen*.

```
@Composable
fun ChefRowScreen(
    foto: Bitmap?,
    name:String,
    mail:String){

    var fotoDefectoState = rememberVectorPainter(image = Icons.Filled.Face)

    Row(
        modifier=Modifier.padding(20.dp),
        verticalAlignment = Alignment.CenterVertically) {
        Image(contentScale = ContentScale.Crop,
            modifier = Modifier
                .size(75.dp)
                .clip(CircleShape),
            painter = foto?.asImageBitmap()?.let { BitmapPainter(it) }
                ?: fotoDefectoState,
            contentDescription = "Imagen de ejemplo")
        Text(modifier = Modifier
            .padding(5.dp),
            text = name,
            style = MaterialTheme.typography.titleSmall
        )
        Text(modifier = Modifier
            .padding(5.dp),
            text = mail,
            color= Color.Red,
            style = MaterialTheme.typography.bodySmall
        )
    }
}
```

📌 **NOTA**: Solamente hemos modificado dentro de `Row` la alineación

El resultado puede no ser el esperado porque puede ser como sucede en este caso que no toda la fila quepa en la pantalla.



La solución puede ser cambiar el tamaño de la fuente, o plantearnos un cambio en el diseño de forma que el mail aparezca debajo del nombre del chef...

Esta solución implica combinar `Row` y `Column` tal y como indicamos a continuación en la función *MyChef*.

```
@Composable
fun MyChef(
    foto: Bitmap?,
    name: String,
    mail: String) {

    var fotoDefectoState = rememberVectorPainter(image = Icons.Filled.Face)

9   Row(modifier = Modifier.padding(20.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.spacedBy(20.dp)
    ) {
13      Image(
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .size(75.dp)
                .clip(CircleShape)
                .align(Alignment.CenterVertically),
            painter = foto?.asImageBitmap()?.let { BitmapPainter(it) }
                ?: fotoDefectoState,
            contentDescription = "Chef Picture"
        )
23      Column(
            horizontalAlignment = Alignment.CenterHorizontally){
25          Text(
                modifier = Modifier
                    .padding(5.dp),
                text = name,
                style = MaterialTheme.typography.titleSmall,
                 color = Color.Black
            )
32          Text(
                modifier = Modifier
                    .padding(5.dp),
                text = mail,
                color = Color.Red,
                style = MaterialTheme.typography.bodySmall
            )
        }
    }
}
```

> 📌 **NOTA**:
> En la **línea 9** definimos la fila, que contiene una imagen (**línea 3**) y una columna (**línea 23**).
> Dentro de esa columna especificamos los dos textos que queremos visualizar (**líneas 12** y **18**)

El resultado es el siguiente

Amparo Laguay Mestre

a_laguay@mail.com

Ejemplo de Box

Recordad que `Box` permite organizar distintos elementos alineándolos con la caja que los contiene.

Veamos el ejemplo anterior definido con `Box` de forma básica sin ningún tipo de alineción.

```
@Composable
fun ChefBoxScreen(
    foto: Bitmap?,
    name:String,
    mail:String){

    var fotoDefectoState = rememberVectorPainter(image = Icons.Filled.Face)

   Box(
        modifier= Modifier.padding(20.dp),
        ) {

        Image(
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .size(75.dp)
                .clip(CircleShape),
            painter = foto?.asImageBitmap()?.let { BitmapPainter(it) }
                ?: fotoDefectoState,
            contentDescription = "Imagen de ejemplo")

        Text(
            modifier = Modifier
                .padding(5.dp),
            text = name,
            style = MaterialTheme.typography.titleSmall
        )

        Text(
            modifier = Modifier
                .padding(5.dp),
            text = mail,
            color= Color.Red,
            style = MaterialTheme.typography.bodySmall
        )
    }
}
```

Como podemos observar los elementos se superponen según orden de creación.

El resultado es el siguiente:

Para llevar el mail del chef al final del box en su esquina derecha, colocaremos la siguiente alineación en su `modifer` `.align(Alignment.BottomEnd)`.

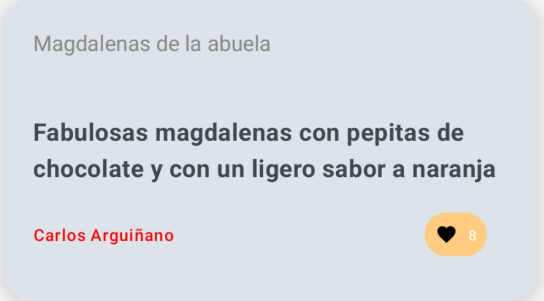
Para colocar la imagen centrada definiremos la siguiente alineación en el `modifier` de la misma: `.align(Alignment.Center)`.

Card

Un *Card* es un *composable* que se utiliza para representar un elemento visual con una apariencia similar a una tarjeta o una caja rectangular. El *Card* es un contenedor que se utiliza para envolver otros componibles y aplicarles un estilo específico.

El *Card* proporciona una superficie elevada con sombra y esquinas redondeadas por defecto, lo que le da una apariencia visual agradable y distintiva. Además, el *Card* también puede tener un color de fondo, bordes personalizados y otras propiedades de estilo.

Vamos a definir un nuevo componente que llamaremos *CardRecipe* con el siguiente diseño.



Para ello definiremos todos los elementos de la vista dentro de un `Box` para terminar personalizando la información de la vista colocando toda ella dentro de un `Card`.

¿Qué elementos configuran nuestro componente?

En primer lugar el nombre de la receta, junto con una descripción de la misma. También vemos el nombre del chef y un elemento más complejo que contiene un icono y el contador de likes.

Vamos a empezar definiendo la vista de este último elemento (*ButoonLike*) y posteriormente su funcionalidad. Lo almacenaremos, siguiendo la lógica de nuestra arquitectura, como un elemento

composable que puede ser utilizado en otras vistas con posterioridad con el nombre **`ButtonLike.kt`**, dentro de la carpeta *composable\buttonlike*.

```
@Composable
fun ButtonLike(
    numberOfLikes: Int,
    modifier: Modifier = Modifier
) {
    Row(
        modifier = modifier.then(
            Modifier
                .clip(RoundedCornerShape(50))
                .background(Color(0xFFFFCC80))
        ),
        verticalAlignment = Alignment.CenterVertically,
    ) {
        IconButton(
            onClick = {}
        ) {
            Icon(
                Icons.Filled.Favorite,
                contentDescription = null,
                tint =  Color.Black
            )
        }
        Text(
            modifier = Modifier.padding(end = 10.dp),
            text = numberOfLikes.toString(),
            color = Color.White
        )
    }
}
```

Pasemos a la funcionalidad...

La funcionalidad de este componente es la siguiente. Cuando pulsamos sobre el icono de favorito, incrementaremos o decrementaremos el contador de likes dependiendo de si ya estaba o no seleccionado, además debemos cambiar el color del mismo para que se marque o desmarque esta selección, concretamente rojo indica seleccionado y negro no seleccionado.

Como la vista se va a recomponer cuando pulsemos sobre la icono va a ser necesario sí o sí controlar el *state*. En este caso particular el estado está compuesto de dos datos, el contador de likes y si está o no seleccionado el icono *favoritos*.

Según nuestra arquitectura, dentro de la carpeta *composable\buttonlike* ya tenemos defida la vista de este elemento en el archivo *ButtonLike.kt* y definiremos el estado *ButtonLikeUiState.kt*

Veamos en primer lugar este último archivo.

```
data class ButtonLikeUiState(
    val numberOfLikes: Int,
    val iLike: Boolean)
```

Es necesario incluir las siguientes modificaciones las mosdificaciones en el elemento *ButtonLike.kt* para elevar el *state*.

```
    @Composable
    fun ButtonLike(
3       iLike: ButtonLikeUiState,
        modifier: Modifier = Modifier,
5       onILikePressed: () -> Unit,
    ) {
        Row(
            modifier = modifier.then(
                Modifier
                    .clip(RoundedCornerShape(50))
                    .background(Color(0xFFFFCC80))
            ),
            verticalAlignment = Alignment.CenterVertically,
        ) {
            IconButton(
                onClick = onILikePressed
            ) {
                Icon(
                    Icons.Filled.Favorite,
                    contentDescription = null,
21                  tint =  if (iLike.iLike) Color.Red else Color.Black
                )
            }
            Text(
                modifier = Modifier.padding(end = 10.dp),
26              text = iLike.numberOfLikes.toString(),
                color = Color.White
            )
        }
    }
```

A continuación definimos el elemento *CardRecipeDay*. Que debe también elevar el estado.

```
fun CardRecipeDay(
    iLike: ButtonLikeUiState,
    recipeName: String,
    recipeDesc:String,
    recipeChef:String,
    onILikePressed: () -> Unit
) {
    Card(
        modifier = Modifier
            .padding(horizontal = 20.dp)
            .fillMaxWidth()
            .wrapContentHeight(),
        //.scale(ratio),
        elevation = CardDefaults.cardElevation(defaultElevation = 10.dp),
        shape = RoundedCornerShape(20.dp)
    ) {
        RecipeDay(
            iLike= iLike,
            recipeName = recipeName,
            recipeDesc =recipeDesc ,
            recipeChef = recipeChef,
            onILikePressed=onILikePressed
        )
    }
}
```

Como podemos observar, no es más que un elemento **`Card`** que envuelve a una función *composable* que se llama **`RecipeDay`**, cuya definición es la siguiente.

```
@Composable
fun RecipeDay(
    iLike: ButtonLikeUiState,
    recipeName: String,
    recipeDesc:String,
    recipeChef:String,
    onILikePressed: () -> Unit
) {
    Box(
        modifier = Modifier
            .height(200.dp)
            .padding(20.dp)
    ) {
        Text(
            modifier = Modifier.align(Alignment.TopStart),
            text =recipeName,
            style = MaterialTheme.typography.bodyMedium,
            color = Color.Gray
        )

        Text(
            modifier = Modifier.align(Alignment.CenterStart),
            text = recipeDesc,
            style = MaterialTheme.typography.titleMedium,
            fontWeight = FontWeight.Bold
        )

        Row(
            modifier= Modifier
                .align(Alignment.BottomStart)
                .fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically,
            horizontalArrangement = Arrangement.SpaceBetween){
            Text(
                text = recipeChef,
                style = MaterialTheme.typography.labelSmall,
                color = Color.Red
            )
            ButtonLikeScaled(
                iLike=iLike,
                scale=0.8f,
                onILikePressed=onILikePressed
            )
            //ButtonLike(iLike=iLike,onILikePressed=onILikePressed)
        }
    }
}
```

Donde **`ButtonLikeScale`** se define como sigue (dentro de *ButtonLike.kt*).

```kotlin
@Composable
fun ButtonLikeScaled(
    iLike: ButtonLikeUiState,
    scale:Float,
    modifier: Modifier = Modifier,
    onILikePressed: () -> Unit
) {
    //val scale = 0.8f
    Box(
        modifier = modifier.then(Modifier.scale(scale))
    ) {
        ButtonLike(
            iLike = iLike,
            onILikePressed = onILikePressed
        )
    }
}
```

Y nos permite escalar el componente al ser envuelto por un **`Box`**.

✏️ Ejercicio Propuesto CardRecipe. Con pulsación sobre *Likes* y elevar el estado al padre.

Aquí tenemos un enlace donde se encuentra información sobre este componente que ofrece Material Design.