# Tema 3.2 - Maquetando nuestra UI

Descargar estos apuntes pdf o html

# Índice

# Introducción

En el primer tema ya hemos hablado de **algunos** componentes básicos de la **capa de Runtime** como son `State`, `remember`, `@Composable` etc.

En este tema vamos a ver cómo maquetar nuestro UI utilizando diferentes componentes de compose definidos en las **capas de la librería** que hablamos en el anterior tema, como son la capa de UI, Fundation y Material.

> **Nota**
>
> Muchos de los elementos de diseño no se pueden ver aislados unos de otros. Por lo que en muchos ejemplos vamos a usar elementos que serán explicados más adelante en profundidad.

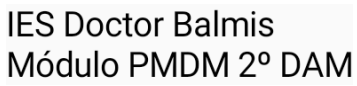
# Conceptos iniciales

Ya hemos visto que las funciones '*Composables*' pueden tener diferentes elementos, como por ejemplo dos textos.

```
@Preview (showBackground = true, name = "CabeceraPreview")
@Composable
fun Cabecera() {
    // Sin contenedor los componentes se superponen
    Text("IES Doctor Balmis")
    Text("Módulo PMDM 2º DAM")
}
```

Necesitaremos, en primer lugar, de **algún tipo de layout** que nos permita poner los elementos uno al lado de otro. Como por ejemplo el '*composable*' `Column` que los irá poniendo uno debajo de otro en vertical.

```
@Preview (showBackground = true, name = "CabeceraPreview")
@Composable
fun Cabecera() {
    Column {
        Text("IES Doctor Balmis")
        Text("Módulo PMDM 2º DAM")
    }
}
```



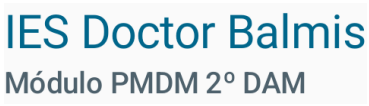

Además, muchos de los elementos de diseño tienen **parámetros** que nos permiten modificar su comportamiento. Por ejemplo, en el componente `Text` podemos ver que son bastantes.

```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    ...
    style: TextStyle = LocalTextStyle.current,
    ...
)
```

Sin embargo, la gran mayoría de ellos no son obligatorios y tienen un valor por defecto asignado en la propia declaración. Además, en varios de estos componentes, para modificarlos **deberíamos hacerlo a través del tema de la aplicación y sus estilos**.

Por ejemplo, si queremos cambiar el color y tamaño del texto, deberíamos hacerlo a través del tema base.

```
@Preview(showBackground = true, name = "CabeceraPreview")
@Composable
fun Cabecera() {
    ProyectoBaseTheme {
        Column {
            Text(
                text = "IES Doctor Balmis",
                style = MaterialTheme.typography.titleLarge,
                color = MaterialTheme.colorScheme.primary
            )
            Text(
                text = "Módulo PMDM 2º DAM",
                style = MaterialTheme.typography.titleSmall,
                color = MaterialTheme.colorScheme.secondary
            )
        }
    }
}
```



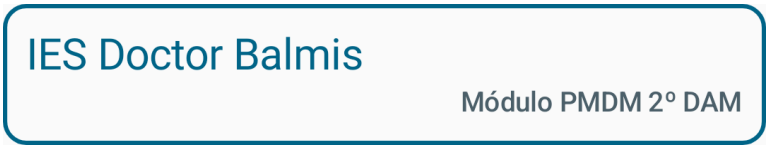

Al poner `ProyectoBaseTheme` como composable raíz de la aplicación, todos los elementos que estén dentro de él, heredarán sus estilos. Por tanto, esto no deberemos hacerlo en cada componente, sino siempre en la raíz de nuestro árbol de composables.

Los estilos y colores los hemos puesto a través de las definiciones de `MaterialTheme`, a través de ella estaremos accediendo a las definiciones del tema de Material Design que es el que viene por defecto en Compose y define `ProyectoBaseTheme`.

De esta manera, toda nuestra aplicación tendrá un aspecto coherente y consistente y podremos cambiarlo fácilmente sin tener que ir componente a componente.

Por último, tendremos una serie de **modificadores** que nos permitirán modificar el espacio que ocupan los componentes, como el `padding` o la alineación, incluso comportamientos como el de `clickable`. No obstante, como veremos más adelante, estos modificadores dependerán del tipo de layout que estemos usando o las características de propio componente.

```
@Preview(showBackground = true, name = "CabeceraPreview")
@Composable
fun Cabecera() {
    ProyectoBaseTheme {
        Column(
            modifier = Modifier
                .fillMaxWidth()
                .border(
                    width = 2.dp,
                    color = MaterialTheme.colorScheme.primary,
                    shape = MaterialTheme.shapes.medium
                )
                .padding(12.dp)
        ) {
            Text(
                text = "IES Doctor Balmis",
                style = MaterialTheme.typography.titleLarge,
                color = MaterialTheme.colorScheme.primary
            )
            Text(
                modifier = Modifier.align(Alignment.End),
                text = "Módulo PMDM 2º DAM",
                style = MaterialTheme.typography.titleSmall,
                color = MaterialTheme.colorScheme.secondary
            )
        }
    }
}
```




En los siguiente puntos vamos a hablar un poco más en profundidad de cada uno de estos conceptos: **'layouts'**, **'modifiers'** y **'Material Design'**, este último ya en el tema siguiente.

### Importante

Vamos a suponer que previsualizamos los ejemplos siguientes con el proyecto `ProyectoBase` que hemos creado anteriormente y usamos por tanto el tema `ProyectoBaseTheme`. Para no repetir el código de previsualización en cada ejemplo, vamos a usar la siguiente plantilla:

```
@Preview(showBackground = true, name = "Test")
@Composable
fun TestPreview() {
    ProyectoBaseTheme {
        // Aquí irá el componente a probar
    }
}
```

# Modificadores de Compose

- Documentación oficial: **Modificadores de Compose**
- Vídeo Español (**DevExperto**): **Modificadores de Compose**

Estos están definidos en la **capa de UI** y nos permiten modificar el comportamiento de otros componentes de capas superiores.

Los modificadores **te permiten decorar o aumentar un elemento *composable***. Por ejemplo, puedes hacer todo esto:

- **Cambiar el tamaño, el diseño y el aspecto del elemento *composable* e incluso como se debe comportar dentro de su contenedor o layout padre**.
- Agregar interacciones de nivel superior, (p. ej., hacer que un elemento sea apto para **hacer clic**, **desplazable**, **arrastrable** o **ampliable**).
- Agregar información (p. ej., etiquetas de accesibilidad)
- Procesar entradas del usuario.

Existen muchos y además dependerán, como hemos comentado, de nuestro contenedor o el scope en que nos encontremos. Por ese motivo los iremos viendo poco a poco a lo largo del curso. Sin embargo, podemos destacar los más comunes:

- `background` - Dibuja una forma de color sólido detrás del *composable*.
- `clickable` - Especifica un controlador que se llamará cuando se haga clic en el *composable*. También causa un efecto de ondulación cuando se realiza el clic.
- `clip` - Recorta el contenido *composable* a una forma especificada.
- `fillMaxHeight` - El *composable* se dimensionará para ajustarse a la altura máxima permitida por su padre.
- `fillMaxSize` - El *composable* se dimensionará para ajustarse a la altura y ancho máximos permitidos por su padre.
- `fillMaxWidth` - El *composable* se dimensionará para ajustarse al ancho máximo permitido por su padre.
- `offset` - Posiciona el *composable* a la distancia especificada desde su posición actual a lo largo del eje x e y.
- `padding` - Agrega espacio alrededor de un *composable*. Se pueden utilizar parámetros para aplicar espaciado a los cuatro lados o para especificar un relleno diferente para cada lado.
- `rotate` - Rota el *composable* en su punto central por un número especificado de grados.
- `scale` - Aumenta o reduce el tamaño del *composable* por el factor de escala especificado.

- `scrollable` - Habilita el desplazamiento para un *composable* que se extiende más allá del área visible del diseño en el que está contenido.
- `size` - Se utiliza para especificar la altura y el ancho de un *composable*. En ausencia de una configuración de tamaño, el *composable* se dimensionará para acomodar su contenido (denominado envoltura).

## Unidades de medida

- **dp**: Este es un valor de **dimensión** que **representa píxeles** independientes de **la densidad de píxeles (dpi) del dispositivo**. Por ejemplo:
  - **1 píxel** en un dispositivo de densidad de pantalla de **160 dpi** es igual a **1 dp**.
  - **1 píxel** en un dispositivo de densidad de pantalla de **240 dpi** es igual a **1.5 dp**.
- **sp**: Acrónimo de (Scaled Pixeles). Son como **dp** pero para fuentes. Además de tener en cuenta la densidad en dpi, también son escalados en función de las preferencias de texto del usuario.

## El orden de los modificadores es importante

El **orden de las funciones de los modificadores es importante**. Como cada función realiza cambios en el Modifier que muestra la función anterior, la secuencia afecta al resultado final.

Supongamos el código anterior de la cabecera donde aplicabamos los modificadores en el siguiente orden:

1. `fillMaxWidth` - Ajustarse al **ancho máximo** permitido por su padre.
2. `border` - Dibuja un borde alrededor del *composable* de un color y ancho especificados.
3. `padding` - Agrega espacio alrededor de un *composable* de **12 dp** .

```
@Composable
fun Cabecera() {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .border(
                width = 2.dp,
                color = MaterialTheme.colorScheme.primary,
                shape = MaterialTheme.shapes.medium
            )
            .padding(12.dp)
    ) {
        ... // Código omitido por abreviar
    }
}
```

Pero, ¿Qué sucede si **aplicamos el padding antes de dibujar el borde**?

Si te fijas en la imagen de resultado el borde se dibuja por encima del padding. Esto es porque el borde se dibuja en el borde del *composable* que en ese momento es el que tiene el tamaño original. Por lo que el padding se aplica después y por tanto el borde se dibuja por encima.


Este es un aspecto a tener en cuenta si en algún momento no obtenemos el resultado esperado.

## Combinando modificadores

Los modificadores se pueden combinar para crear un único modificador que se puede reutilizar en varios lugares. Por ejemplo, queremos añadir el borde y padding en varios lugares y no queremos **repetir código** de tal manera que al cambiarlo en un sitio se cambie en todos.

Podemos definir una **función de extensión** para el tipo `Modifiers` de la siguiente manera:

```
private fun Modifier.miBordeYPadding(
    color: Color = Color.Transparent,
    shape: Shape
) = border(width = 2.dp, color = color, shape = shape).padding(12.dp)

@Composable
fun Cabecera() {
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .miBordeYPadding()
    ) {
        ... // Código omitido por abreviar
    }
}
```

> **Importante**
>
> Si vamos a utilizar este modificador en varios sitios, lo mejor es definirlo en un **archivo separado** en algún paquete de utilidades en la jerarquía de paquetes pero dentro el paquete de `ui`. Además, todas las funciones de extensión sobre `Modifiers` deben permanecer cómo publicas para ser accesibles. Recuerda además, que cada vez que uses un modificador compuesto definido por tí, estás creando una **dependencia fuerte** entre el componente y el modificador fuera de las librerías de la capa de `compose.ui`. Por lo que deberías pensar si es necesario o no.

Otra forma de hacerlo es con `Modifier.then()` que nos permite combinar modificadores de la siguiente manera.

Imaginemos nuestro *'composable'* `Cabecera` que va a ser usado en varios sitios, pero va a tener un **background diferente** en cada uno de ellos.

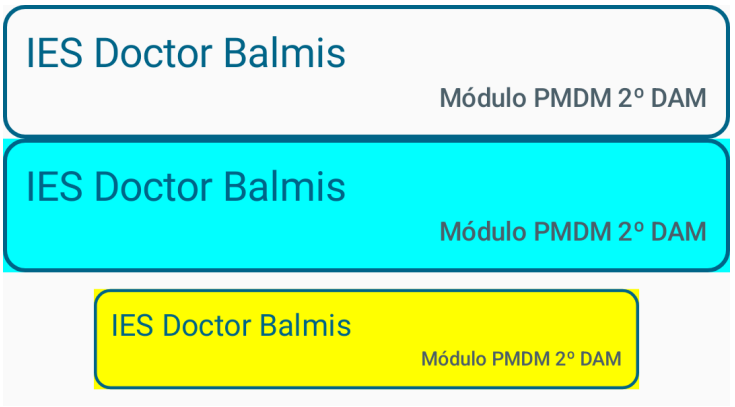
Podemos hacer lo siguiente ...

```
// Le pasamos un modificador a nuestro componente
// un modificador que se aplicará antes que cualquier otro
// le pasaremos el valor por defecto Modifier si no queremos
// aplicar ningún modificador
@Composable
fun Cabecera(modifier: Modifier = Modifier) {
    Column(
        // Asignamos el modificador y luego el operador then
        // para que se aplique después de los modificadores recibidos.
        modifier = modifier.then(
            Modifier
                .fillMaxWidth()
                .miBordeYPadding()
        )
    ) {
        ... // Código omitido por abreviar
      }
}
```

Si modificamos nuestro *'preview'* de la siguiente manera para probarlo...

```
@Preview(showBackground = true, name = "Test")
@Composable
private fun Test() {
    ProyectoBaseTheme {
        Column {
            // Sin aplicar ningún modificador
            Cabecera()
            // Cambiando el color de nuestra cabecera a Cyan
            Cabecera(modifier = Modifier.background(Color.Cyan))
            // Escalando la cabecera a 0.75 (75% del tamaño original)
            // y después aplicando el background Amarillo.
            Cabecera(
                modifier = Modifier
                    .scale(0.75f)
                    .background(Color.Yellow)
            )
        }
    }
}
```

Obtendremos el siguiente resultado ....


IES Doctor Balmis
Módulo PMDM 2º DAM
IES Doctor Balmis
Módulo PMDM 2º DAM
IES Doctor Balmis
Módulo PMDM 2º DAM

# Profundizando en el trabajo con Textos

- Documentación oficial: **Textos en Compose**
- Puedes bajar el código de los ejemplo de **EjTextosBanner.kt**

## Creando un estilo de texto

**Aunque no es lo recomendado y no corresponde al trabajo de un desarrollador**, puede darse el caso que al estar creando uno de nuestros componentes, queramos crear un estilo de texto fuera de los definidos por defecto en `MaterialTheme`. Incluso en las adaptaciones de los mismos que podamos hacer en `com.holamundo.ui.theme → Theme.kt`

> **Importante**
>
> En este caso ya no podremos hacer una función de extensión sobre `TextStyle` para modificar una de sus propiedades porque son objetos inmutables. Además, **deberemos de llevar cuidado de aplicar sobre el objeto que modifique el estilo el calificativo `remember { objeto }` para que no se cree en cada composición**.

```
private fun sombra() = Shadow(
    color = Color.Gray,
    offset = Offset(4f, 4f),
    blurRadius = 4f
)

@Composable
fun Cabecera(modifier: Modifier = Modifier) {
    Column(
        modifier = modifier.then(
            Modifier.fillMaxWidth().miBordeYPadding()
        )
    ) {
        Text(
            text = "IES Doctor Balmis",
            // Copiamos (por ser inmutable) del estilo que aplicábamos de MaterialTheme
            // y le asignamos un nuevo valor a la propiedad sombra.
            style = MaterialTheme.typography.titleLarge.copy(
                // OJO!! Para no crear el objeto sombra en cada composición le
                // aplicaremos el modificador remember.
                shadow = remember { sombra() }
            ),
            color = MaterialTheme.colorScheme.primary
        )
        ... // Código omitido por abreviar
    }
}
```

## Establecer varios estilos sobre un mismo texto

En ocasiones querremos que partes de un mismo texto tengan diferentes estilos sin tener que crear varios textos dentro de un layout `Row`. Para ello podemos usar la función `AnnotatedString` que nos permite establecer diferentes estilos a diferentes partes de un mismo texto.

**AnnotatedString** es una clase de datos que contiene lo siguiente:

- Un valor `Text`
- Una List de `SpanStyleRange`, equivalente al estilo intercalado con el rango de posición dentro del valor de texto
- Una List de `ParagraphStyleRange` que especifica la alineación del texto, la dirección del texto, la altura de la línea y el estilo de sangría del texto.

`TextStyle` ya lo hemos usado y es para uso en el elemento componible Text, mientras que **SpanStyle** y **ParagraphStyle** se usan en AnnotatedString.

Además, siempre vamos a poder convertir un `TextStyle` en un `SpanStyle` y un `ParagraphStyle` con la funciones `toSpanStyle()` y `toParagraphStyle()` respectivamente.

La diferencia entre `SpanStyle` y `ParagraphStyle` es que ParagraphStyle se puede aplicar a un párrafo completo, mientras que SpanStyle puede aplicarse a nivel de carácter. Una vez que una parte del texto se marca con un ParagraphStyle, esa parte queda separada del resto como si tuviera feeds de líneas al principio y al final. Por poner un símil, es como si aplicáramos a ese texto las etiquetas `<p>...</p>` de HTML.

> **Nota**
>
> No vamos a ver todas las combinaciones y usos por su complejidad. Pero puedes visitar este enlace a la **documentación oficial**.

```
@Composable
fun Cabecera(modifier: Modifier = Modifier) {
    Column(
        modifier = modifier.then(
            Modifier
                .fillMaxWidth()
                .miBordeYPadding(
                    color = MaterialTheme.colorScheme.primary,
                    shape = MaterialTheme.shapes.medium
                )
        )
    ) {
        ... // Código omitido por abreviar
        Text(
            modifier = Modifier.align(Alignment.End),
            style = MaterialTheme.typography.titleSmall,
            color = MaterialTheme.colorScheme.secondary,
            text = buildAnnotatedString {
                append("Módulo ")
                withStyle(
                    style = SpanStyle(
                        fontSize = MaterialTheme.typography.titleMedium.fontSize,
                        color = MaterialTheme.colorScheme.inversePrimary
                    )
                ) {
                    append("PMDM") // Aplicamos el estilo solo a PMDM
                }
                append(" 2º DAM")
            })
    }
}
```

# Brushes (Pinceles)

- Documentación oficial: **Pinceles en Compose**

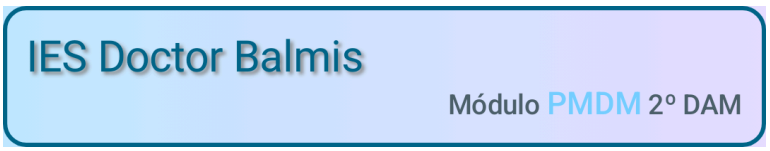
Aunque este también es un **tema más avanzado y específico de diseñadores** gráficos. **Vamos a verlo por encima** para que sepas que existe y puedas usarlo en caso de que tengas algún pincel definido en el tema que quieras usar.

En Compose, los pinceles se utilizan para pintar formas y trazos en un lienzo o canvas. Pero también **los podemos usar para pintar el fondo de un *composable* como un layout a través del modificador** `background`.

Por ejemplo, si añadimos un pincel a nuestra cabecera.

```
@Composable
fun Cabecera(modifier: Modifier = Modifier) {
    // Creamos el objeto pincel con remember para que no se cree en cada composición.
    val colorI = MaterialTheme.colorScheme.primaryContainer
    val colorF = MaterialTheme.colorScheme.tertiaryContainer
    val pincel = remember { Brush.horizontalGradient(listOf(colorI, colorF)) }
    Column(
        modifier = modifier.then(
            Modifier
                // Aplicamos el pincel al background al principio del modificador.
                // Recuerda que el orden importa.
                .background(pincel)
                .fillMaxWidth()
                .miBordeYPadding(
                    color = MaterialTheme.colorScheme.primary,
                    shape = MaterialTheme.shapes.medium
                )
        )
    ) {
    }
}
```

Obtendremos el siguiente resultado ...



Supongamos que queremos pintar un fondo de un texto o un layout con una imagen a modo de pincel. Podemos hacerlo de la siguiente manera...

Descarga la imagen **balmis.png** y ponla en la carpeta `res/drawable` de tu proyecto.

```
@Composable
fun Cabecera(modifier: Modifier = Modifier) {
    // En este caso no podemos usar remember porque imageResource
    // es una función composable y no puede ser llamada desde una función remember.
    val pincel = ShaderBrush(ImageShader(ImageBitmap.imageResource(id = R.drawable.balmis)))
    Column(
        modifier = modifier.then(
            Modifier
                .background(pincel)
                .fillMaxWidth()
                .miBordeYPadding(
                    color = MaterialTheme.colorScheme.primary,
                    shape = MaterialTheme.shapes.medium
                )
        )
    ) {
        ... // Código omitido por abreviar
    }
}
```

Obtendremos el siguiente resultado ...


**Importante**

El usar una imagen como pincel es **más util para pintar textos** que para pintar layouts. Ya que en el caso de los layouts, la imagen se repite en el eje x e y. Por lo que si el layout es más grande que la imagen, se verá la imagen repetida. En el caso de querer que el layout tenga un fondo con una imagen, lo mejor es usar un layout `Box` y asignarle un componente `Image` de fondo, sobre el que tendremos un mejor control de visualización como veremos en el siguiente tema.

# Layouts básicos

- Documentación oficial: **Conceptos básicos de diseño**
- ⭐ Vídeo en Inglés (Android Developers): **Lista de reproducción**
- Vídeo en Inglés (Android Developers): **Workshop about basic layouts in Compose**
- Vídeo en Español (DevExperto): **Box, Column y Row en Jetpack Compose**

Los layouts son los encargados de **distribuir** los elementos de nuestra UI. En Compose, los layouts son también **composables** y por tanto, podemos anidarlos unos dentro de otros.

Los básicos están definidos en la **capa de Fundation** y son los siguientes:

## Surface

**Surface** es un **layout** que nos permite añadir una **forma**, **color de fondo** y **elevación** a un *composable* que no los tenga. Es decir, que podemos aplicarle un **sombreado** y una **elevación** como si fuera un `CardView` de XML.

Pero lo más interesante, es que **nos permite establecer un tema diferente** al que tengamos en el resto de la aplicación. Ya que este se aplica automáticamente a todos su contenido. Por eso veremos que si le asignamos un color, el color de los textos cambia en función del tema.

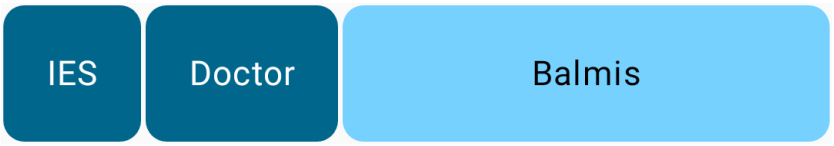
Por esta razón, vamos a usarlo para definir un componente `TextoConForma` que de ahora en adelante usaremos para probar los diferentes layouts de los ejemplos.

```
@Composable
fun TextoConForma(
    modifier: Modifier = Modifier,
    texto : String = "Hola Mundo",
    color : Color = MaterialTheme.colorScheme.primary) {
    Surface(
        modifier = modifier.then(Modifier.padding(1.dp)),
        color = color,
        shape = RoundedCornerShape(10.dp)
    ) {
        Text(
            modifier = Modifier.padding(20.dp),
            textAlign = TextAlign.Center,
            text = texto)
    }
}
```

Si usamos el siguiente `preview` para probarlo ...

```
@Preview(showBackground = true, name = "Test")
@Composable
fun Test() {
    ProyectoBaseTheme {
        Row {
            TextoConForma(texto = "IES")
            TextoConForma(texto = "Doctor")
            // Fíjate en este caso como el color del texto se adapta al color de fondo.
            // Esto no sucederá en otros layouts como Box o Column.
            TextoConForma(
                // Intentará ocupar el ancho que quede libre.
                modifier = Modifier.fillMaxWidth(),
                texto = "Balmis",
                color = MaterialTheme.colorScheme.inversePrimary)
        }
    }
}
```

Obtendremos el siguiente resultado ...




# Box

- Documentación oficial (capa **fundation**): **Box**
- Vídeo en Inglés (**Stevdza-San**): **Box Layout**
- Descarga el código de los ejemplos: **Box.kt**

```
@Composable
inline fun Box(
    modifier: Modifier = Modifier,
    contentAlignment: Alignment = Alignment.TopStart,
    propagateMinConstraints: Boolean = false,
    content: @Composable BoxScope.() -> Unit
): Unit
```

`Box` se dimensionará para ajustarse al contenido, sujeto a las restricciones de su contenedor.

Cuando los hijos son más pequeños que el padre, por defecto se posicionarán dentro del Box, según la alineación de contenido. Para especificar individualmente las alineaciones de los diseños secundarios, use el modificador `BoxScope.align`.

Por defecto, el contenido se medirá sin las restricciones mínimas entrantes de la Box, a menos que `propagateMinConstraints` sea `true`. Si `propagateMinConstraints` se establece en `true`, el tamaño mínimo establecido en la Box también se aplicará al contenido, mientras que de lo contrario el tamaño mínimo solo se aplicará al `Box`.

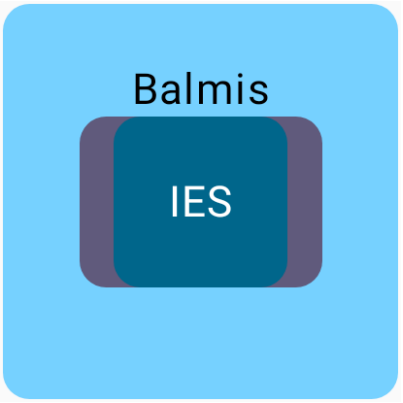
**Ejemplo**: Se puede establecer `propagateMinConstraints` a `true` cuando el Box tiene contenido en el que no se pueden especificar modificadores directamente y **se necesita establecer un tamaño mínimo en el contenido del** `Box`.

Es **importante** tener en cuenta que, cuando el `Box` tiene más de un hijo, estos **se apilarán uno encima del otro** (posicionados como se explica arriba) en el orden de composición.

En el siguiente ejemplo todos los *composables* `TextoConForma` se apilarán uno encima del otro en el centro porque hemos indicado `contentAlignment = Alignment.Center` y como hemos comentado el tamaño del `Box` se ajustará al más grande en la pila.

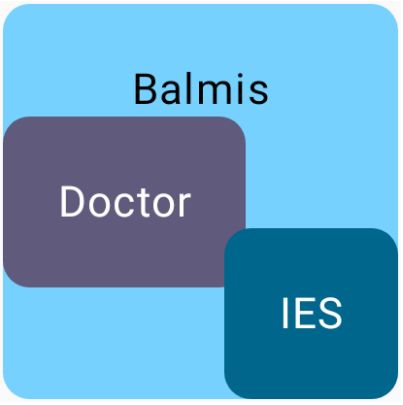
> ⚠ **Cuidado**
>
> Si pusiéramos el `TextoConForma` de mayor dimensión al final. **este taparía a todos los demás**.

```
@Composable
fun BoxApiladoAlCento() {
    Box (contentAlignment = Alignment.Center) {
        TextoConForma(
            modifier = Modifier.size(150.dp, 150.dp),
            texto = "Balmis",
            color = MaterialTheme.colorScheme.inversePrimary)
        TextoConForma(
            texto = "Doctor",
            color = MaterialTheme.colorScheme.tertiary)
        TextoConForma(
            texto = "IES")
    }
}
```




En el siguiente ejemplo, hemos ajustado de forma independiente la alineación de cada uno de los *composables* `TextoConForma`.

```
@Composable
fun BoxConAlineacionesIndependientes() {
    Box {
        TextoConForma(
            modifier = Modifier.align(Alignment.Center)
                               .size(150.dp, 150.dp),
            texto = "Balmis",
            color = MaterialTheme.colorScheme.inversePrimary)
        TextoConForma(
            modifier = Modifier.align(Alignment.CenterStart),
            texto = "Doctor",
            color = MaterialTheme.colorScheme.tertiary)
        TextoConForma(
            modifier = Modifier.align(Alignment.BottomEnd),
            texto = "IES")
    }
}
```




Esto es posible, porque dentro del Composable `Box` tenemos acceso a un `BoxScope` (**ámbito o alcance del Box**) que nos permite aplicar modificadores a cada uno de los hijos de forma independiente. Estos modificadores adicionales que aparecen son...

- `align()`: Alinea el hijo dentro del área de contenido de la Box utilizando el valor de alineación especificado.
- `matchParentSize()`: Dimensiona el hijo al que se aplica el modificador para que coincida con el tamaño de la Box principal.

> **Importante**
>
> Piénsalo, ¿Qué sentido tendría aplicar un modificador `align()` a nuestro `TextoConForma` si no estuviera dentro de algo que lo contuviera?. Además, tenemos que saber como dispone nuestro contenedor a su continente para saber de que formas podemos alinear.
> Esto nos va asuceder también con otros layouts como `Column` o `Row` que veremos a continuación.

# Column y Row

- Documentación oficial (capa **fundation**): **Column**
- Documentación oficial (capa **fundation**): **Row**
- Vídeo en Inglés (**Stevdza-San**): **Rows and Columns**
- Vídeo en Inglés (**Philipp Lackner**): **Rows and Columns**
- Descarga el código de los ejemplos: **ColumnYRow.kt**

`Column` y `Row` son layouts *composables* que colocan a sus *hijos* en una secuencia **vertical y horizontal respectivamente**.

> **Importante**
>
> Al igual que sucede con el `Box` los hijos dentro de un `Column` o un `Row` estarán dentro de un ámbito o alcance determinado por el `ColumnScope` o el `RowScope` respectivamente. Esto permitirá que sus hijos puedan aplicar modificadores adicionales como `weight` o `align` relacionados con su contenedor respectivo.

Deberemos tener en cuenta que, por defecto, los elementos no se desplazan. Consulta los modificadores `Modifier.verticalScroll` y `Modifier.horizontalScroll` para obtener este comportamiento

**Cuando el tamaño de la columna o fila es mayor que la suma de los tamaños de sus elementos hijos**, se puede especificar un `verticalArrangement` o `horizontalArrangement` para definir la posición de los elementos hijos dentro.

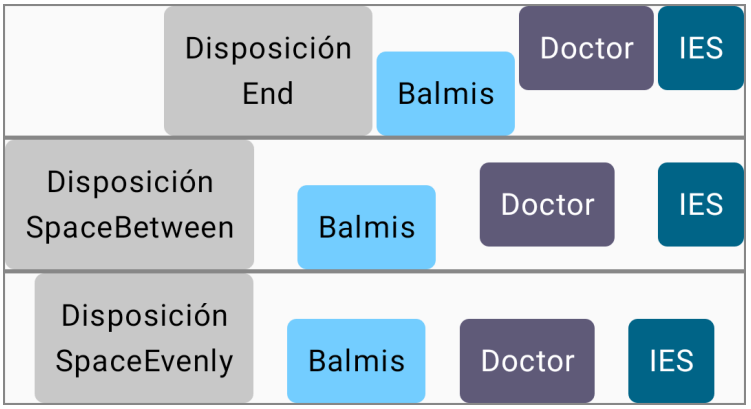
Veamos un ejemplo de disposiciones en un `Row`.

Todo lo aplicado en este ejemplo se puede aplicar también a un `Column` de forma análoga pues su comportamiento es idéntico pero en el `eje y`.

```
// Definimos un componente Row con una propiedad horizontalArrangement
// y otra verticalAlignment para definir la disposición de los elementos
@Composable
fun MyRow(
    horizontalArrangement: Arrangement.Horizontal,
    verticalAlignment : Alignment.Vertical
) {
    Row(
        // Aplicamos un modificador para que se ajuste al ancho máximo
        // y le aplicamos un borde para que se vea el tamaño que ocupa.
        modifier = Modifier.fillMaxWidth().border(1.dp, Color.Gray),
        horizontalArrangement = horizontalArrangement,
        verticalAlignment = verticalAlignment
    ) {
        // La primera caja en la fila me indicará la disposición
        // que se ha aplicado a todo el Row.
        val disposicion = horizontalArrangement.toString()
        TextoConForma(
            texto = "Disposición\n" + disposicion.substring(
                disposicion.indexOf('#') + 1
            ),
            color = Color.LightGray
        )
        // Ignorará la alineación vertical general del Row y se alineará
        // en la parte baja del row siempre.
        TextoConForma(
            modifier = Modifier.align(Alignment.Bottom),
            texto = "Balmis",
            color = MaterialTheme.colorScheme.inversePrimary
        )
        // Si no indico nada la alineación vertical será la general del Row
        TextoConForma(
            texto = "Doctor",
            color = MaterialTheme.colorScheme.tertiary
        )
        TextoConForma(texto = "IES")
    }
}
```

Vamos a usar el *composable* `MyRow` para probar las diferentes disposiciones que podemos aplicar a un `Row`. Para ell, definimos un *composable* `RowsWithColumns` que contendrá un `Column` con varios hijos `MyRow`.

```
@Composable
fun ColumnsWithRows() {
    Column(
        modifier = Modifier.fillMaxWidth()
    ) {
        // Todos se disponen en horizontal al final del Row (Derecha)
        // Todos se alinean en vertical arriba del Row salvo lo que
        // tengan una alineación personalizada.
        MyRow(
            horizontalArrangement = Arrangement.End,
            verticalAlignment = Alignment.Top
        )
        // Todos se disponen en horizontal espaciados de forma equitativa
        // salvo el primero y el último que se ajustarán a los extremos.
        // Todos se alinean en vertical centrados del Row salvo lo que
        // tengan una alineación personalizada.
        MyRow(
            horizontalArrangement = Arrangement.SpaceBetween,
            verticalAlignment = Alignment.CenterVertically
        )
        // Todos se disponen en horizontal espaciados de forma equitativa.
        // Todos se alinean en vertical abajo del Row salvo lo que
        // tengan una alineación personalizada.
        MyRow(
            horizontalArrangement = Arrangement.SpaceEvenly,
            verticalAlignment = Alignment.Bottom
        )
    }
}
```



## Pesos en Column y Row

El diseño `Column` es capaz de **asignar alturas máximas respecto al alto total del Column** a los hijos, según sus pesos proporcionados mediante el modificador `ColumnScope.weight` lo mismo sucederá con `Row` pero con `RowScope.weight` para **asignar anchos máximos respecto al ancho total del Row**.

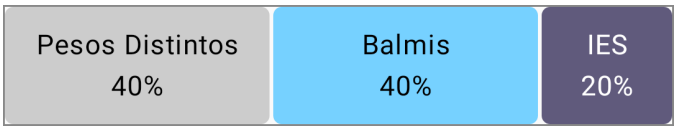
En el siguiente ejemplo vamos a definir *composable* `RowConPesosIguales` para probar los pesos que podemos asignar a los hijos de un `Row`. A todos los hijos les asignaremos el mismo peso para que se repartan el espacio disponible de forma equitativa y al ser **3** cada uno ocupará **1/3** del espacio disponible. Además, fíjate que todas las cajas tienen el modificador `fillMaxWidth()` para intentar expandirse al máximo dentro del espacio que le corresponda según el peso.

```
@Composable
fun RowConPesosIguales() {
    Row(
        modifier = Modifier.fillMaxWidth().border(1.dp, Color.Gray),
        horizontalArrangement = Arrangement.SpaceEvenly,
        verticalAlignment = Alignment.CenterVertically
    ) {
        TextoConForma(
            modifier = Modifier.fillMaxWidth().weight(1/3f),
            texto = "Pesos\nIguales",
            color = Color.LightGray
        )
        TextoConForma(
            modifier = Modifier.fillMaxWidth().weight(1/3f),
            texto = "Balmis",
            color = MaterialTheme.colorScheme.inversePrimary
        )
        TextoConForma(
            modifier = Modifier.fillMaxWidth().weight(1/3f)
                .align(Alignment.Top),
            texto = "IES",
            color = MaterialTheme.colorScheme.tertiary
        )
    }
}
```


Realmente el valor que le pasa al modificador `weight` es un `Float` que representa el porcentaje o tanto por uno del espacio disponible que ocupará el hijo.

En el siguiente ejemplo, las dos primeras cajas ocuparán el **0.4f o el 40%** del espacio disponible y la tercera el **0.2f o el 20%**. Todos los valores deben sumar **1.0f o 100%** del espacio disponible.

```
@Composable
fun RowConPesosDistintos() {
    Row(
        modifier = Modifier.fillMaxWidth().border(1.dp, Color.Gray),
        horizontalArrangement = Arrangement.SpaceEvenly
    ) {
        TextoConForma(
            modifier = Modifier.fillMaxWidth().weight(0.40f),
            texto = "Pesos Distintos\n40%",
            color = Color.LightGray
        )
        TextoConForma(
            modifier = Modifier.fillMaxWidth().weight(0.40f),
            texto = "Balmis\n40%",
            color = MaterialTheme.colorScheme.inversePrimary
        )
        TextoConForma(
            modifier = Modifier.fillMaxWidth().weight(0.20f),
            texto = "IES\n20%",
            color = MaterialTheme.colorScheme.tertiary
        )
    }
}
```




Si un hijo no **tiene un peso asignado**, se le pedirá su altura o ancho preferido antes de que se calculen los tamaños de los hijos con pesos proporcionalmente a su peso en función del espacio disponible restante.

Ten en cuenta que, si el `Column` o el `Row` tienen aplicado su modificador de **scroll**, se **ignorarán los pesos proporcionados**, ya que el espacio disponible restante será infinito.

# FlowColumn y FlowRow

Aunque en esta sección solo vamos a hablar de `FlowRow` por no extendernos. Todo lo que se diga es aplicable a `FlowColumn` de forma análoga. Además, este tipo de layout no son muy comunes en aplicaciones de uso general y sí posiblemente en los **casos en que se necesite un diseño más complejo y adaptable a diferentes tamaños de pantalla**.

- Documentación oficial (capa **fundation**): **FlowRow**

```
@Composable
@ExperimentalLayoutApi
inline fun FlowRow(
    modifier: Modifier = Modifier,
    horizontalArrangement: Arrangement.Horizontal = Arrangement.Start,
    verticalArrangement: Arrangement.Vertical = Arrangement.Top,
    maxItemsInEachRow: Int = Int.MAX_VALUE,
    content: @Composable FlowRowScope.() -> Unit
): Unit
```

`FlowRow` es un diseño que rellena elementos **de izquierda a derecha** (ltr) en diseños LTR o de derecha a izquierda (rtl) en diseños RTL y **cuando se queda sin espacio**, se mueve a la siguiente "fila" o "línea" colocada en la parte inferior y, a continuación, continúa rellenando elementos hasta que se agotan los elementos. Por hacer un símil con **css** podría equivaler a un `display: flex; flex-wrap: wrap;`.

Si especificamos un `maxItemsInEachRow` al `FlowRow`, este se rellenará la fila hasta que se acaben los elementos o se llegue al número máximo de elementos por fila que hemos especificado y saltará a la siguiente si queda alguno.

Si especificamos un `RowScope.weight` a sus elementos, este peso se aplicará en función de los elementos que le acompañen en la misma fila.

Por ejemplo, supongamos las siguientes condiciones:

- Añadimos **12 elementos** a nuestro `FlowRow`.
- Todos tienen un **peso de** `0.5f`.
- El `máximo de elementos por fila es de 5`.
- 👁️ Nuestro `FlowRow` **tiene el suficiente ancho** para albergar 5 elementos.

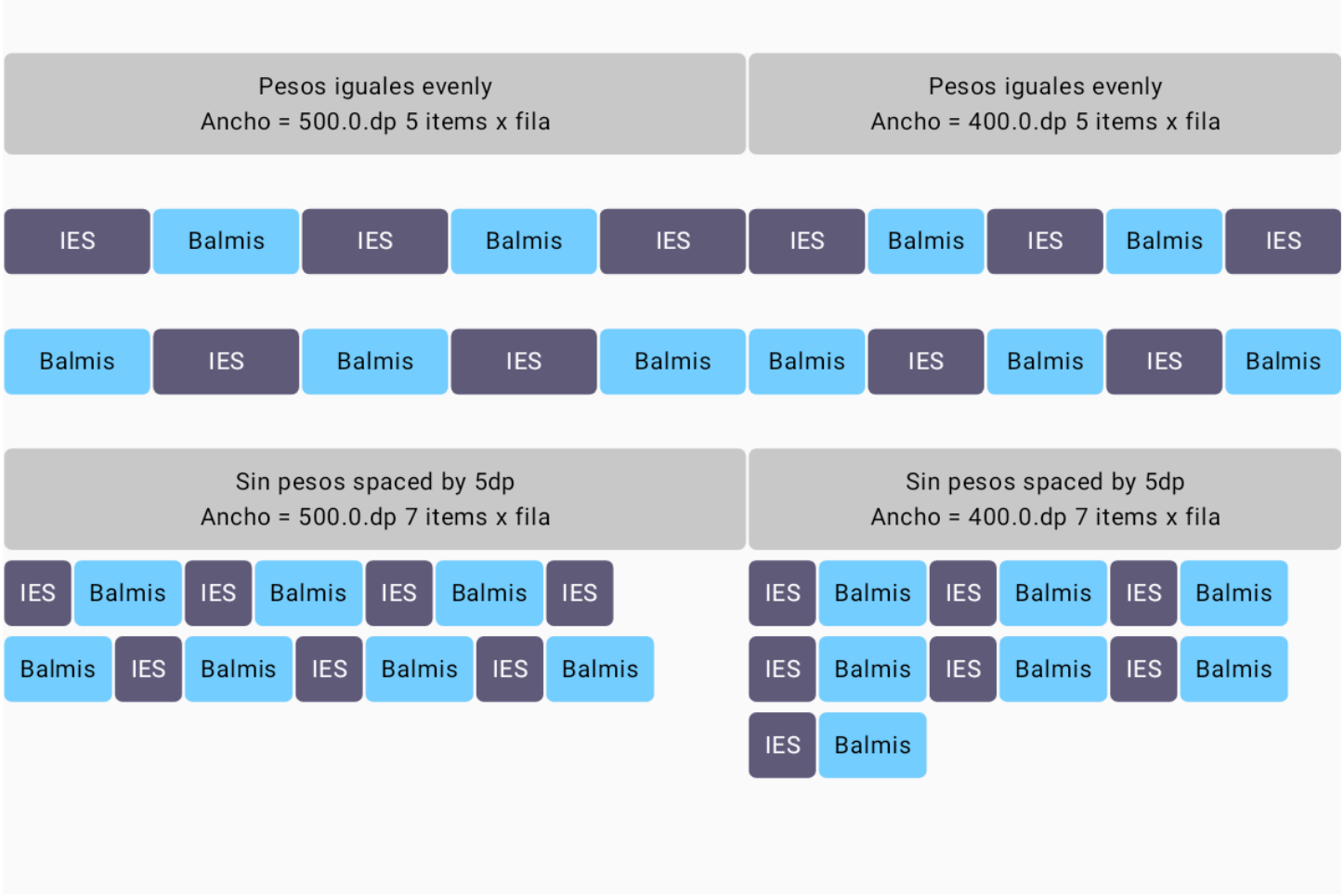
entonces...

- En las **dos primeras filas** se colocarán **5 elementos** cada una el peso total será de `0.5 x 5 = 2.5`. Por tanto, el anchó de cada elemento ocupará `0.5 / 2.5 = 0.2` esto es un **20%** del ancho de la fila.
- En la **última fila** se colocarán **2 elementos**, que son los que quedan. Por tanto, el peso total será de `0.5 x 2 = 1` por lo que el anchó de cada elemento ocupará `0.5 / 1 = 0.5` esto es un **50%** del ancho de la fila.

Esta es solo una de los **cientos de casuísticas** que se pueden dar. Por lo que te recomiendo que pruebes y experimentes con este layout.

**Ejemplo**: Vamos a ver algunas combinaciones que puedes descargar del siguiente fuente **FlowRow.kt**.

En este caso mostraremos la pantalla de previsualización que nos ha devuelto el código de ejemplo.




Para crear esta composición hemos usado el siguiente código de **Preview** con un **área de 900 dp x 600 dp**. Además, los **FlowRow q la izquierda** tienen un área de **500 dp x 300 dp** y **los a la derecha** de **400 dp X 300 dp**.

El código del preview sería ...

```
@Preview(
    showBackground = true, name = "FlowColumnYRowPreview",
    device = "spec:width=900dp,height=600dp,dpi=480"
)
@Composable
fun FlowColumnYRowPreview() {
    ProyectoBaseTheme {
        Column()
        {
            Row()
            {
                FlowRowConPesos(500.dp, 300.dp)
                FlowRowConPesos(400.dp, 300.dp)
            }
            Row()
            {
                FlowRowSinPesos(500.dp, 300.dp)
                FlowRowSinPesos(400.dp, 300.dp)
            }
        }
    }
}
```

En los `FlowRow` **con pesos de la parte superior** tendremos:

1. `maxItemsInEachRow = 5`
2. `horizontalArrangement = Arrangement.Start` Sin efectos al intentar todos los elementos expandirse al máximo ancho.
3. `verticalArrangement = Arrangement.SpaceEvenly` Rellena el espacio verticalmente entre las filas de elementos de forma equitativa.
4. `verticalAlignment = Alignment.CenterVertically` De cada elemento de forma personalizada al centro verticalmente en la fila que le corresponde.
5. Todos los elementos intentan expandirse al máximo en el eje X con `fillMaxWidth()`.
6. Todos los elementos tienen un peso de `1 / 10 = 0.1f` por lo que si hay 5 elementos en la fila cada uno ocupará `0.1 / 0.5 = 0.2` esto es un **20%** del ancho de la fila.

El código que genera los **FlowRow** con pesos de la parte superior sería ...

```
@OptIn(ExperimentalLayoutApi::class)
@Composable
fun FlowRowConPesos(ancho: Dp, alto: Dp) {
    val maxHijosPorFila = 5
    val filas = 2
    val hijos = maxHijosPorFila * filas
    FlowRow(
        modifier = Modifier.size(ancho, alto),
        horizontalArrangement = Arrangement.Start,
        verticalArrangement = Arrangement.SpaceEvenly,
        maxItemsInEachRow = maxHijosPorFila
    ) {
        TextoConForma(
            modifier = Modifier.fillMaxWidth(),
            texto = "Pesos iguales evenly\n"
                    + "Ancho = ${ancho} ${maxHijosPorFila} items x fila",
            color = Color.LightGray
        )
        for (i in 1..ceil(hijos / 2f).toInt()) {
            TextoConForma(
                modifier = Modifier
                    .fillMaxWidth()
                    .weight(1 / hijos.toFloat())
                    .align(Alignment.CenterVertically),
                texto = "IES",
                color = MaterialTheme.colorScheme.tertiary
            )
            TextoConForma(
                modifier = Modifier
                    .fillMaxWidth()
                    .weight(1 / hijos.toFloat()),
                texto = "Balmis",
                color = MaterialTheme.colorScheme.inversePrimary
            )
        }
    }
}
```

En los `FlowRow` **sin pesos de la parte inferior** tendremos:

1. `maxItemsInEachRow = 7`
2. `horizontalArrangement = Arrangement.Start` Ahora sí tiene efectos.
3. `verticalArrangement = Arrangement.spacedBy(5.dp)` Rellena el espacio verticalmente entre las filas de elementos con un espacio de **5 dp**.

4. Al ser 7 elementos en el `FlowRow` inferior-izquierdo le caben los 7 y los alinea a Start. Pero en el `FlowRow` inferior-derecho solo caben 6 elementos por lo que hay una fila más donde coloca los 2 que le quedan.

El código que genera los **FlowRow** con pesos de la parte superior sería ...

```
@OptIn(ExperimentalLayoutApi::class)
@Composable
fun FlowRowSinPesos(ancho: Dp, alto: Dp) {
    val maxHijosPorFila = 7
    val filas = 2
    val hijos = maxHijosPorFila * filas
    FlowRow(
        modifier = Modifier.size(ancho, alto),
        horizontalArrangement = Arrangement.Start,
        verticalArrangement = Arrangement.spacedBy(5.dp),
        maxItemsInEachRow = maxHijosPorFila
    ) {
        TextoConForma(
            modifier = Modifier.fillMaxWidth(),
            texto = "Sin pesos spaced by 5dp\n"
                    + "Ancho = ${ancho} ${maxHijosPorFila} items x fila",
            color = Color.LightGray
        )
        for (i in 1..ceil(hijos / 2f).toInt()) {
            TextoConForma(
                texto = "IES",
                color = MaterialTheme.colorScheme.tertiary
            )
            TextoConForma(
                texto = "Balmis",
                color = MaterialTheme.colorScheme.inversePrimary
            )
        }
    }
}
```

### Consideraciones importantes

- `FlowRow` y `FlowColumn` **no están pensados para representar datos de una fuente de datos**. Para eso tenemos `LazyColumn`, `LazyRow` y `LazyGrid` que trataremos en temas posteriores.
- Su uso está pensado para **diseños dinámicos o responsive**, esto es, que se ajusten a las dimensiones de la pantalla de forma fluida. Por ejemplo: Queremos que ciertos elementos se distribuyan al girar la pantalla sin hacer un diseño específico o nuestro diseño tiene que adaptarse a Móvil, Android Tv o Desktop.
- En ocasiones los cálculos pueden ser complejos y **no se obtiene el resultado esperado**. Por lo que es recomendable **probar y experimentar** con los diferentes parámetros que nos ofrece.

# Imágenes

- Documentación oficial: **Imágenes en Jetpack Compose**
- Documentación oficial: **Component Image Fundation Layer**

Usaremos el elemento composable **Image** para mostrar un gráfico en la pantalla.

Encontraremos diferentes sobrecargas del *composable* `Image` que nos permitirán cargar diferentes formas de declarar las imágenes.

```
@Composable
fun Image(imageVector: ImageVector, ...)

@Composable
fun Image(bitmap: ImageBitmap, ...)

@Composable
fun Image(painter: Painter, ...)

...
```

# Formas de manejar un recurso imagen

1. **Painter**
   **Abstracción para algo que se puede dibujar**. Dibuja en un área delimitada especificada y ofrece algunos mecanismos de alto nivel que los usuarios pueden utilizar para configurar cómo se dibuja el contenido.
   Para cargar una imagen (por ejemplo: PNG, JPEG, WEBP) o un recurso vectorial del disco, usa la API de `painterResource` con tu imagen de referencia.

   ```
   // La opción más adecuada para obtener un recurso dibujable en Compose
   val painter = painterResource(id = R.drawable.balmis)
   ```

2. **ImageBitmap**
   Representa un mapa de bits de imagen. Para cargar una imagen (por ejemplo: PNG, JPEG, WEBP) del disco, usa la API de `imageResource` definida en `ImageBitmap`. Podemos usar el constructor `BitmapPainter` para obtener un `Painter` a partir de un `ImageBitmap` y así asignarla a la abstracción `Painter`.

```
val imageBitmap : ImageBitmap = ImageBitmap.imageResource(id = R.drawable.balmis)
val painterBitmap = remember(imageBitmap) { BitmapPainter(imageBitmap) }
```

Para pasar de `Bitmap` de Android a `ImageBitmap` podemos usar el método asociado `asImageBitmap()`.
Pasar de `ImageBitmap` de Compose a `Bitmap` de Android podemos usar el método asociado `asAndroidBitmap()`.

```
val bitmap : Bitmap = ...
val imageBitmap : ImageBitmap = bitmap.asImageBitmap()
```

3. **ImageVector**
   Representa un vector de imagen. Para cargar un recurso vectorial del disco, usa la API de `vectorResource` definida en `ImageVector`. Además, podemos usar la función `rememberVectorPainter` para obtener un `PainterVector` a partir de un `ImageVector` y así asignarla a la abstracción `Painter` además de **recordarla** en la recomposición.

```
val imageVector : ImageVector = ImageVector.vectorResource(id = R.drawable.balmis)
val painterVector : PainterVector = rememberVectorPainter(imageVector)
```

Podemos también cargar un icono de los proporcionados por Material como imagen vectorial.

```
var painterFavorite = rememberVectorPainter(image = Icons.Filled.Favorite)
```

## Parámetros más comunes

Tendremos más parámetros según la sobrecarga de la función *composable* `Image` entre los que podemos destacar:

- Un texto `contentDescription` que se mostrará si no se puede cargar la imagen. La herramienta de accesibilidad `TalkBack` lee la descripción del contenido, por lo que debes asegurarte de que el texto sea significativo si se lee en voz alta y se traduce.
- Modificadores con `Modifier`. (Veremos alguno de utilidad en los ejemplos)
- Un `Alignment` que especifica cómo se alinea la imagen dentro de su espacio asignado.
- Un `alpha` que especifica la transparencia de la imagen. Es un valor entre `0.0f` y `1.0f`.
- Un `ColorFilter` que especifica un filtro de color para aplicar a la imagen.
- Un `ContentScale` que especifica cómo se escala la imagen dentro de su espacio asignado. Sus valores pueden ser...
  - `Crop`: Escala la imagen para que llene el espacio asignado, recortando la imagen si es necesario.
  - `FillBounds`: Escala la imagen para que llene el espacio asignado, sin recortar la imagen.
  - `Fit`: Escala la imagen para que quepa dentro del espacio asignado, sin recortar la imagen.
  - `FillHeight`: Escala la imagen para que llene la altura del espacio asignado, sin recortar la imagen.
  - `FillWidth`: Escala la imagen para que llene el ancho del espacio asignado, sin recortar la imagen.
  - `Inside`: Escala la imagen para que quepa dentro del espacio asignado, recortando la imagen si es necesario.
  - `None`: No escala la imagen.

# Cargando Imágenes de forma asíncrona (Coil)

> **Nota**
>
> **Coil** es la librería recomendada por Google para cargar imágenes en Android con Jetpack Compose.
> Sin embargo, existen muchas otras como **Glide**, **Landscapist**, etc.

Para cargar imágenes de forma **asíncrona** podemos usar la librería **Coil**. Esta librería nos permite cargar imágenes desde una URL, un archivo o un recurso de Android. Además, nos permite transformar la imagen antes de mostrarla, por ejemplo, para redimensionarla o aplicarle un filtro. Además, también nos permite **cachear las imágenes** para que no tengamos que volver a descargarlas.

Veamos un ejemplo de como cargar una imagen desde una URL:

1. Añadimos la dependencia de **Coil con Gradle**:

```
// En libs.version.toml
[versions]
coil = "2.7.0"

[libraries]
coil-compose = { group = "io.coil-kt", name = "coil-compose", version.ref = "coil" }

// En el build.gradle.kt (del módulo app)
dependencies {
    ...
    implementation(libs.coil.compose)
}
```

2. Puesto que vamos a cargar una imagen desde una URL, debemos añadir el permiso de acceso a Internet en el `AndroidManifest.xml`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application>
        ...
    </application>
```

3. Ahora ya simplemente deberemos usar el composable `AsyncImage` que pos proporciona **Coil** con unas propiedades muy similares a las de `Image` por ejemplo:

```
import coil.compose.AsyncImage

@Preview
@Composable
fun Imagen() = AsyncImage(
    model = "https://pmdmiesbalmis.github.io/B3_Capa_UI/assets/imagenes/logo.png",
    contentDescription = null,
    contentScale = ContentScale.Crop,
    modifier = Modifier
        .size(200.dp)
        .border(2.dp, MaterialTheme.colorScheme.primary)
)
```

Fíjate que donde pone model es donde especificamos la URL de la imagen. Pero también podríamos hacer `model = R.drawable.imagen` o `model = File("ruta de la imagen")`, etc. (**Ver documentación**).

### 🎓 Caso de estudio:

Vamos a redibujar nuestra cabecera, pero esta vez en lugar de dibujar la imagen de fondo con un `Brush`, vamos a usar un componente `Image` que sería más adecuado. Para seguir los siguientes ejemplos necesitaremos descargar el siguientes recursos y arrastrarlos a `res/drawable`: **balmis.png** que ya hemos usado y **logo.png** que usaremos en los ejemplos. También, pudes descargarte el código de los ejemplos en **EjImagenBanner.kt**.

La imagen de nuestro ejemplo será la siguiente ...


Definimos una función `sombra` que ya hemos usado en este tema.

```
private fun sombra() = Shadow(
    color = Color.Gray,
    offset = Offset(4f, 4f),
    blurRadius = 4f
)
```

Voy modularizar el UI en varios composables y de paso vamos a pasar el lambda de un composable como parámetro.

```
@Composable
// Pasamos un lambda como parámetro con el composable que
// renderizará el contenido de la cabecera
fun Cabecera5(contenido: @Composable () -> Unit = {}) {
    // Definimos una caja donde superponderemos la
    // imagen de fondo y el contenido alineados al centro.
    Box(
        modifier = Modifier.fillMaxWidth(),
        contentAlignment = Alignment.Center
    ) {
        // Ponemos la imagen de fondo de la fachada del Balmis
        Image(
            modifier = Modifier
                .fillMaxWidth()
                .border(
                    width = 2.dp,
                    color = MaterialTheme.colorScheme.primary,
                    shape = MaterialTheme.shapes.medium
                )
                // Recortamos la iamagen con el shape definido en Tema (Material).
                .clip(MaterialTheme.shapes.medium),
            // Obtenemos Painter del recurso de imagen de la fachada.
            painter = painterResource(id = R.drawable.balmis),
            contentDescription = "Fachada del IES Balmis",
            // La imagen se escala para que ocupe todo el espacio recortándose
            contentScale = ContentScale.Crop
        )
        // Superponemos el contenido de la cabecera a la imagen.
        contenido()
    }
}
```

A continuación definimos un composable **`Contenido`** que será el contenido de la cabecera que pasaremos por parámetro. En este caso, será una fila con el logo y el texto del IES Balmis de ejemplos anteriores.

```
@Composable
private fun Contenido() {
    // El cóntenido irá en una fila con el logo 20% y el texto 80%.
    // Además, lo alinearemos verticalmente al centro.
    Row(
        // Debemos intentar ocupar todo el espacio en ancho.
        modifier = Modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.Start,
        verticalAlignment = Alignment.CenterVertically
    ) {
        Image(
            modifier = Modifier.weight(0.2f), // Ocupa en ancho el 20% de la fila
            painter = painterResource(id = R.drawable.logo),
            contentDescription = "Logo IES Balmis",
            // Cambiamos el color de la imagen para usar el color primario
            // de nuestro tema de Material.
            colorFilter = ColorFilter.tint(
                MaterialTheme.colorScheme.primary
            ),
            // La imagen se escala para que ocupe todo el espacio recortándose.
            // Esto es, El alto de la imagen de fondo y el ancho del 20% de la fila.
            contentScale = ContentScale.Crop
        )
        Text(
            text = "IES Doctor Balmis",
            modifier = Modifier
                .scale(1.5f)    // Escalamos el texto al 150%
                .weight(0.8f),  // Ocupa en ancho el 80% de la fila
            style = MaterialTheme.typography.titleLarge.copy(
                shadow = remember { sombra() }
            ),
            color = MaterialTheme.colorScheme.primary,
            // Alienamos el texto al centro del 80% que ocupa en la fila.
            textAlign = TextAlign.Center
        )
    }
}
```

Por último, en nuestro @Preview definimos el contenido de la cabecera con el composable **`Contenido`**.

```
@Preview(showBackground = true, name = "ImagenesPreview")
@Composable
fun ImagenesPreview() {
    ProyectoBaseTheme {
        Cabecera5 { Contenido() }
    }
}
```