

Formulario para añadir un contacto en agenda

[Descargar estos apuntes](#)

'Codelab' guiado para crear un formulario de acta de un contacto.

En el siguiente ejercicio partiremos del **ejercicio 1** del **Tema 3.6** donde **añadimos la lista de contactos**. La idea del mismo es poder añadir o editar un contacto.

Paso 1.- Añadiendo clases de utilidad para introducir campos de texto

Descarga el recurso [validacion.zip](#). Si lo descomprimes te generará una carpeta

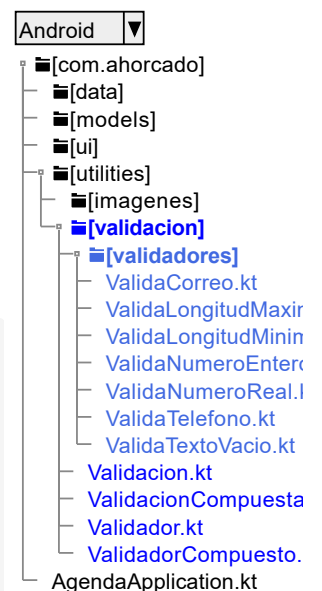
validacion que debes arrastrar a tu proyecto al bajo el paquete

com.pmdm.agenda.ui.utilities.

Nos creará el paquete validación en el que podremos encontrar las siguientes clases que se ven en el esquema de la derecha. Algunas de ellas ya las hemos tratado cuando hablamos de **TextField**.

```
// Validacion.kt -----
// Describe el resultado de una validación.
// Si hay error, se indica el mensaje de error.
// Será el UIState que reciben nuestros TextField para indicar si hay error o no.
data class Validacion(
    val hayError: Boolean,
    val mensajeError: String? = null
)
```

```
// ValidacionCompuesta.kt -----
// Es una clase de utilidad que tiene una lista de validaciones que debemos pasar antes
// de dar por válidos los datos de un formulario.
// Rehusamos para ello los objetos que representan los estados de validación de cada campo y si alguno
// de ellos tiene error lo indicaremos en la propiedad calculada de solo lectura hayError de esta clase.
class ValidacionCompuesta {
    private val validaciones = mutableListOf<Validacion>()
    fun add(validacion: Validacion): ValidacionCompuesta {
        validaciones.add(validacion)
        return this
    }
    val hayError: Boolean
        get() = validaciones.any { it.hayError }
}
```



```
// Validador.kt -----
// Abstracción (SAM) de una función de validación de campos de formulario
// como por ejemplo un email, un teléfono, etc.
// Devuelve un objeto Validacion que devolverá un estado de validación para un TextField.
fun interface Validador {
    fun valida(texto: String): Validacion
}
```

```
// ValidadorCompuesto.kt -----
// Implementa la interfaz Validador y es una clase de utilidad que tiene
// una lista de validadores que debemos pasar para un único TextField
// antes de dar po válido el dato introducido.
// Por ejemplo para un teléfono, podemos tener una validación que compruebe
// que no está vacío, otra que compruebe que tiene una longitud mínima.
// Nota: Las validaciones se ejecutan en orden y si alguna de ellas tiene error
// se devuelve el error y no se ejecutan las siguientes.
class ValidadorCompuesto(validador: Validador) : Validador {
    private val validadores = mutableListOf(validador)
    fun add(validador: Validador): ValidadorCompuesto {
        validadores.add(validador)
        return this
    }

    override fun valida(texto: String): Validacion =
        validadores.firstOrNull { it.valida(texto).hayError }?.valida(texto)
        ?: Validacion(false)
}
```

Dentro del paquete **validacion** encontramos otro paquete **validadores** que contiene las clases que implementan la interfaz **Validador**. Estas clases son las que se encargan de validar un campo de texto concreto. Por ejemplo, la clase **ValidaCorreo** se encarga de validar que un campo de texto es un correo electrónico válido.

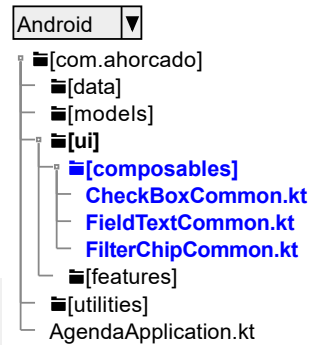
```
class ValidaCorreo(
    val mensajeError: String = "Correo no válido"
) : Validador {
    override fun valida(texto: String): Validacion =
        if (Regex("^[A-Za-z](.*)((@){1})(.{1,})(\\.\\.)(.{1,})$").matches(texto))
            Validacion(false)
        else
            Validacion(true, mensajeError)
}
```

Así vamos a tener validadores para validar que un campo de texto es un correo, un teléfono, un número entero, un número real, etc.

Paso 2.- Definiendo composables reutilizables

Dentro del paquete `ui` vamos a crear el paquete `composables`. Aquí irán aquellos composables 'genéricos' creados por nosotros que podamos reutilizar en diferentes partes de nuestra aplicación o candidatos en otras aplicaciones.

Dependerán únicamente de la **capa de compose** de `material` y por tanto podremos usar definiciones `foundation`, `ui` y `runtime`.



🚩 **Nota:** Lo ideal es que se implementarán en un proyecto a parte y se publicarán como librerías para que puedan ser usadas en otros proyectos.

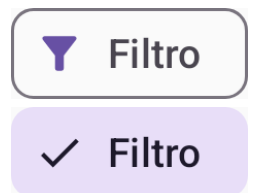
En nuestro caso tenemos:

👉 **Importante:** Tómate unos minutos en examinar el código de estos composables y entender como funcionan. Prueba las previews para ver como se ven.

- **CheckBoxCommon.kt** 🔗 : Composable que nos permite crear un `CheckBox` con una etiqueta.
- **FieldTextCommon.kt** 🔗 : Composables de utilidad que me van a permitir introducir un campo de texto con una etiqueta, una validación y en ocasiones un icono. Como por ejemplo...
 - `OutlinedTextFieldEmail` para introducir un email.
 - `OutlinedTextFieldPhone` para introducir un teléfono.
 - `OutlinedTextFieldPassword` para introducir una contraseña.
 - etc.

Puedes ver en la función `@Composable` con la previsualización para tests un ejemplo de como usar estos componentes junto a las clase con la **validacion** y los diferentes **validadores** que hemos definido en el paso anterior.

- **FilterChipCommon.kt** 🔗 : Composable que nos permite crear un `Chip` con un texto y un icono para filtrar, tal y como indica en Material3. De momento no lo vamos a usar pero añádelo también a tu proyecto. Como puedes ver en las imágenes de ejemplo, el chip tiene dos estados. Uno que indica que el filtro se puede aplicar y otro que indica que el filtro está aplicado.

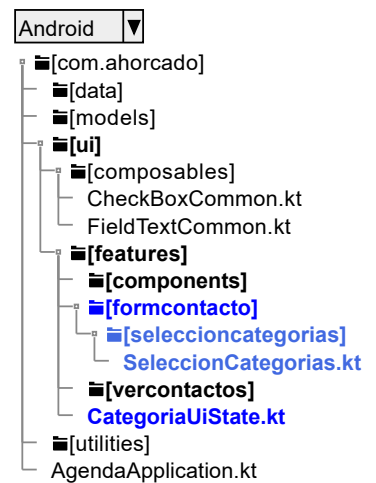


Estos *composables* junto a `ImagenContacto.kt` que ya debería estar incluido en el proyecto, los usaremos en el siguiente paso para crear el formulario de contacto.

Paso 3.- Creando componente de seleccion de categorías

En ocasiones necesitamos crear componentes que no son genéricos y que solo se van a usar en un sitio concreto de nuestra aplicación. Este es el caso de la selección de categorías que vamos a usar en el formulario de contacto. Sin embargo, el hacerlo nos va a permitir reutilizarlo y descomponer un estado complejo en estados más simples como pudiera ser el que guarda la selección de categorías.

En este caso vamos a crearlo dentro de la carpeta `features` y dentro **vamos a crear el paquete `formcontacto`** donde va a ir la implementación del mismo. Como la selección de categorías es un sub-componente del mismo, lo vamos a crear a su vez en un paquete denominado `formcontacto.seleccioncategorias`.




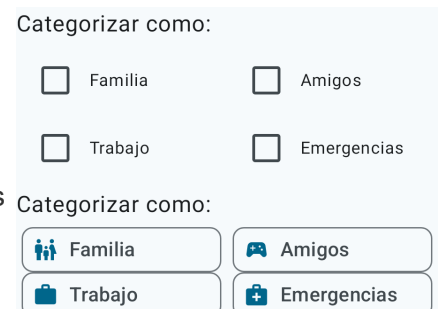
Además, vamos a reutilizar la clase de estado `CatergoriaUiState.kt` que definimos en el ejercicio 1 del tema 3.6 y que me permitía tener un estado para las categorías de un contacto y asociarlo a un icono. Deberías tenerla ya definida en `ui.features.vercontactos`. Puesto que ahora la vamos a usar en más sitios vamos a moverla a `ui.features` como se muestra en el esquema de carpetas para que su alcance abarque ambos componentes.

- **CatergoriaUiState.kt:** Nos define un estado para dicho componente.

```
data class CatergoriaUiState(  
    val amigos: Boolean = false,  
    val amigosIcon : ImageVector = Icons.Filled.SportsEsports,  
    val trabajo: Boolean = false,  
    val trabajoIcon : ImageVector = Icons.Filled.Work,  
    val familia: Boolean = false,  
    val familiaIcon : ImageVector = Icons.Filled.FamilyRestroom,  
    val emergencias: Boolean = false,  
    val emergenciasIcon : ImageVector = Icons.Filled.MedicalServices  
)
```

La implementación del componente de selección de categorías es la siguiente:

- **SeleccionCategorias.kt** : Encapsula o *modulariza* la funcionalidad de seleccionar una categoría para el contacto en un componente. Para ello, define dos *composables* `SeleccionCategoriasConCheckBox` y `SeleccionCategoriasConFilterChip` que me van a permitir marcar o demarcar una determinada categoría para el contacto. En el caso de los checks para asignársela o no y en el segundo para una futura funcionalidad de filtrado por categorías.



El interfaz de usuario del componente `SeleccionCategoriasConCheckBox` es el siguiente:

```

@Composable
fun SeleccionCategoriasConCheckBox(
    modifier: Modifier = Modifier,
    etiquetaGrupoState: String,
    // Recibe el estado de las categorías
    categoriaState: CategoriaUiState,
    // Eleva cualquier cambio en el estado de las categorías.
    onCategoriaChanged: (CategoriaUiState) -> Unit
)

```

Para poder mostrar los checks sin repetir el código. Definiremos la clase `CheckBoxUiState` que encapsula el estado de un check. De esta forma, podemos definir una lista de `CheckBoxUiState` y recorrerla para mostrar los checks. Para ello, el componente `CheckBoxCommon` que definimos en el paso anterior.

🚩 **Nota:** En este caso el coste de eficiencia de crear la lista al recomponerse el componente por un cambio de estado en las categorías es mínimo y no afecta al rendimiento. Además, es compensado al no tener que **repetir el código de cada check**.

```

data class CheckBoxUiState(
    val label : String = "",
    val selected: Boolean = false,
    val icon: ImageVector? = null,
    val onClick: (Boolean) -> Unit = {}
)

val contenido = listOf(
    CheckBoxUiState(
        label = "Familia",
        selected = categoriaState.familia,
        icon = categoriaState.familiaIcon,
        // Cambiamos toda la clase de estado de categorías
        onClick = { onCategoriaChanged(categoriaState.copy(familia = it)) }
    ),
    CheckBoxUiState(
        label = "Amigos",
        selected = categoriaState.amigos,
        icon = categoriaState.amigosIcon,
        onClick = { onCategoriaChanged(categoriaState.copy(amigos = it)) }
    ),
    ...
)

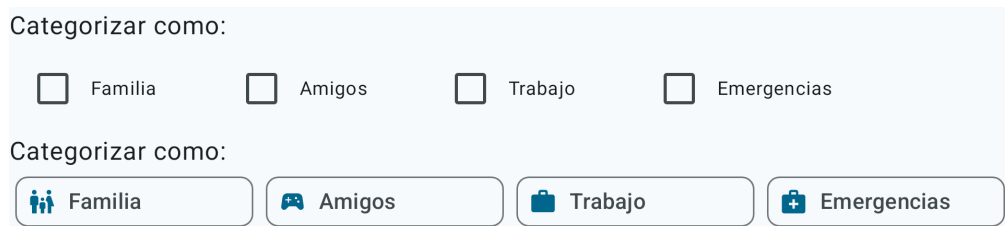
```

Ahora podemos pasar la lista de `CheckBoxUiState` a un `LazyVerticalGrid` de tal manera que se muestren en una **cuadrícula adaptable al ancho** y como hemos comentado nos evitar repetir el código de cada check, además de, no tener que hacer una maquetación compleja para mostrarlos.

```
Column(modifier = modifier) {
    Text(text = etiquetaGrupoState)

    LazyVerticalGrid(
        columns = GridCells.Adaptive(120.dp),
        contentPadding = PaddingValues(all = 4.dp),
        horizontalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(contenido) { item ->
            CheckboxWithLabel(
                label = item.label,
                checkedState = item.selected,
                enabledState = true,
                onStateChange = item.onClick
            )
        }
    }
}
```

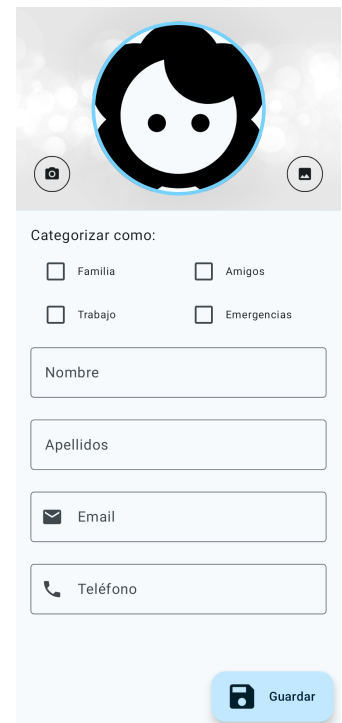
Fíjate en la imagen de abajo como al ser el ancho de las columnas adaptable, se ajustan al ancho de la pantalla. Además, al ser un `LazyVerticalGrid` se ajusta a la altura de los elementos que contiene.



Paso 4.- Creando el formulario de contacto

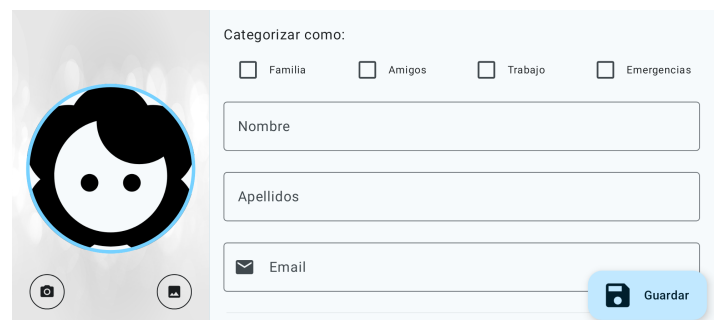
Ya tenemos todos los elementos necesarios para crear el formulario de contacto. Para ello, vamos a crear el componente `FormContactoScreen` dentro del paquete `ui.features.formcontacto` que hemos creado en el pasos anteriores. En él vamos a tener pues los siguientes elementos como se muestra en la imágenes de ejemplo:

1. Una `ImagenContacto` que creamos en ejercicios anteriores, que además irá en un `Box` con dos `OutlinedIconButton` de **Material3** que **en el futuro** nos permitirán cambiar la imagen del contacto ya sea haciendo una foto o seleccionándola de la galería de imágenes.
2. El componente `SeleccionCategoriasConCheckBox` que acabamos de crear para seleccionar las categorías.
3. Varios `OutlinedTextField` definidos en `FieldTextCommon.kt` para introducir los datos del contacto.
4. Un FAB que nos permitirá guardar las modificaciones.
5. Por último, un `SnackBar` que nos indicará si tenemos algún error de validación a resolver.



A vertical screenshot of the contact form. At the top is a circular profile picture placeholder with a camera icon on the left and a gallery icon on the right. Below this is a section titled 'Categorizar como:' with four checkboxes: 'Familia', 'Amigos', 'Trabajo', and 'Emergencias'. Following the categories are four text input fields labeled 'Nombre', 'Apellidos', 'Email' (with an envelope icon), and 'Teléfono' (with a phone icon). At the bottom right is a blue button with a save icon and the text 'Guardar'.

Además, para mejorar la experiencia de usuario y como ya vimos en el el juego del ahorcado. Vamos a usar un `BoxWithConstraints` para en el caso de que giremos el dispositivo a horizontal, mostrar el formulario en dos columnas. Teniendo en cuenta la introducción de datos debería ser *Scrollable* verticalmente.



A horizontal screenshot of the contact form. The profile picture placeholder is on the left. To its right, the 'Categorizar como:' section has four checkboxes: 'Familia', 'Amigos', 'Trabajo', and 'Emergencias'. Below these are three text input fields: 'Nombre', 'Apellidos', and 'Email' (with an envelope icon). A blue 'Guardar' button is at the bottom right.

Sin embargo, antes de empezar a definir el interfaz. Vamos a concretar las clases de estado que vamos a necesitar para el formulario y de gestión de eventos sobre el mismo.

TODO: Aquí COmentar que hay que cambiar de ubicación `ContactoUiState.kt`