

# 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 DI con Hilt a nuestro proyecto

En primer lugar vamos a añadir el soporte para la inyección de dependencias y así poder gestionar el ciclo de vida los objetos que necesitemos en nuestra aplicación y de paso practicar un poco más su uso. Para ello, vamos a recordar alguno de los pasos descritos en el tema:

1. Añade todos los plugins y librerías en Gradle, descritas en el Tema 3.5 y sincroniza el proyecto de Gradle.
2. Añadimos la anotación `@HiltAndroidApp` en `AgendaApplication`
3. Añadimos la anotación `@AndroidEntryPoint` en `MainActivity`.
4. Vamos a marcar para inyección los siguientes constructores:

```
class ContactoDaoMock @Inject constructor() { ... }

class ContactoRepository @Inject constructor(
    private val dao: ContactoDaoMock
)
```

5. Vamos a definir el paquete `com.pmdm.agenda.di` y en él el módulo `AppModule.kt` donde vamos a definir los servicios que vamos a inyectar en nuestra aplicación. En este caso, solo tenemos el repositorio de contactos.

```

@Module
@InstallIn(SingletonComponent::class)
class AppModule {

    @Provides
    @Singleton
    fun provideContactoDaoMock() : ContactoDaoMock = ContactoDaoMock()

    @Provides
    @Singleton
    fun provideContactoRepository(
        contactoDaoMock: ContactoDaoMock
    ) : ContactoRepository = ContactoRepository(contactoDaoMock)
}

```

6. Ahora en `ListaContactosViewModel` vamos inyectar el repositorio y a indicarle a Hilt que hay un `@HiltViewModel` que cuando instancie lo asocie al ciclo de vida oportuno.

```

@HiltViewModel
class ListaContactosViewModel @Inject constructor(
    private val repository: ContactoRepository
) : ViewModel() { ... }

```

## Paso 2.- 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

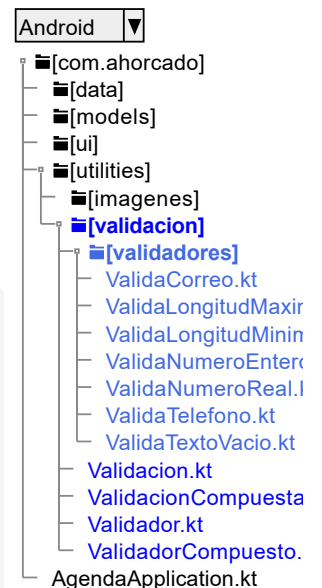
`com.pmdm.agenda.ui.utilities`.

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.

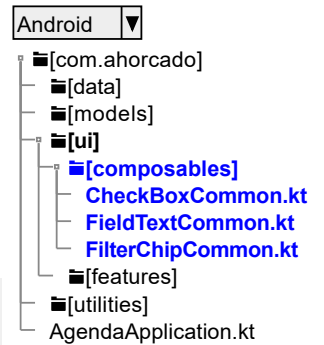
```
class ValidaCorreo(  
    val mensajeError: String = "Correo no válido"  
) : Validador {  
    override fun valida(texto: String): Validacion =  
        if (Regex("^[A-Za-z](.*)((@){1})(.{1,})(\\.)(.{1,})$").matches(texto))  
            Validacion(false)  
        else  
            Validacion(true, 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 3.- 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.

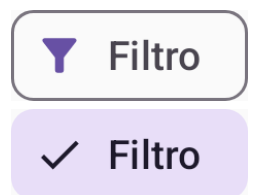
En nuestro caso tenemos:

👉 **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.

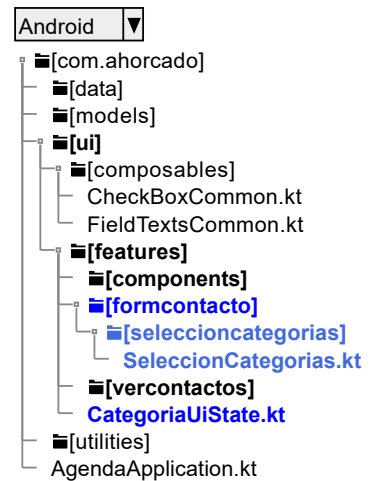


Estos *composables* junto a `ImagenContacto.kt` que ya debería estar incluido en el proyecto, los usaremos en el siguiente paso para crear el formulario de contacto.

## Paso 4.- 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.

Categorizar como:

<input type="checkbox"/> Familia	<input type="checkbox"/> Amigos
<input type="checkbox"/> Trabajo	<input type="checkbox"/> Emergencias

Categorizar como:

 Familia	 Amigos
 Trabajo	 Emergencias

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)) }
    ),
    ...
)

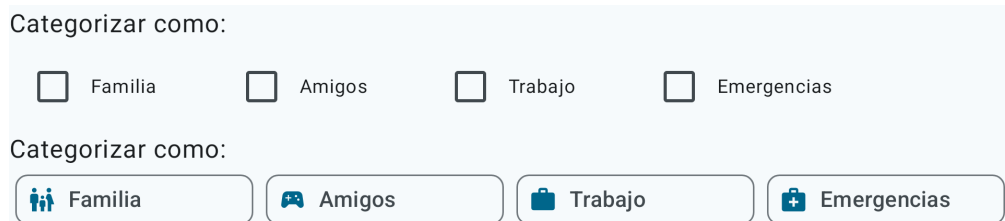
```

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.

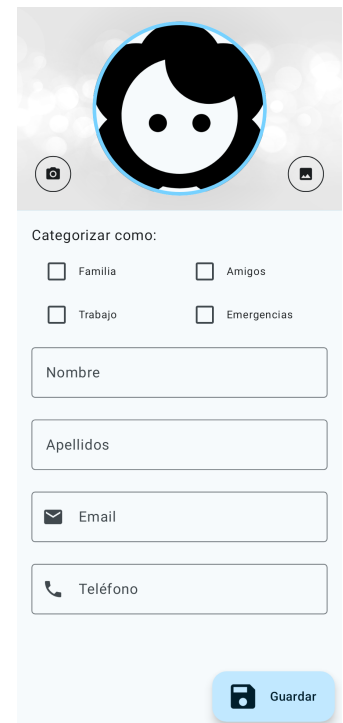




## Paso 5.- Creando el formulario de contacto

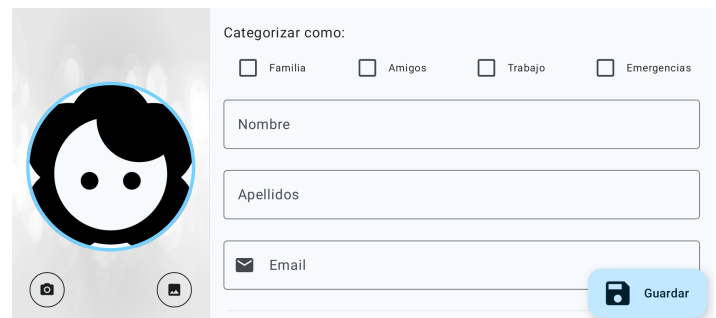
Ya tenemos todos los elementos necesarios para crear el formulario de contacto. Para ello, vamos a crear el componente `FormContactoScreen` dentro del paquete `ui.features.formcontacto` que hemos creado en el pasos anteriores. En él vamos a tener pues los siguientes elementos como se muestra en la imágenes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



The mockup shows a contact form in portrait orientation. At the top is a circular profile picture placeholder with a camera icon on the left and a gallery icon on the right. Below the image are four checkboxes for categorization: 'Familia', 'Amigos', 'Trabajo', and 'Emergencias'. Underneath 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 'Guardar' (Save) button with a floppy disk icon.

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.



The mockup shows the contact form in landscape orientation. The profile picture placeholder is on the left. To its right, the categorization checkboxes are arranged in a single row. Below them, the text input fields for 'Nombre', 'Apellidos', and 'Email' are stacked vertically. The 'Guardar' button remains 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.

1. `ContactoUiState.kt` que guardaba el estado de un contacto en la lista y que teníamos en el paquete `ui.features.vercontactos`. La reutilizaremos ahora para guardar los datos del contacto en el formulario por tanto la moveremos a `ui.features` junto a `CatergoriaUiState.kt`
2. Además del estado de un estado de los datos del contacto, vamos a necesitar un estado que gestione la validación de los mismos. Para segregar responsabilidades, vamos a definirlo en una clase de estado propia denominada `ValidacionContactoUiState.kt` la cual va a contener el estado del objeto `Validacion` para los diferentes datos.

```
data class ValidacionContactoUiState(
    val validacionNombre: Validacion = Validacion(false),
    val validacionApellidos: Validacion = Validacion(false),
    val validacionCorreo: Validacion = Validacion(false),
    val validacionTelefono: Validacion = Validacion(false),
    val validacionContacto: Validacion = Validacion(false)
)
```

3. Vamos a definir otra clase denominada **ValidadorContacto.kt** que encapsule la lógica de validación de los datos del contacto. Para ello, vamos a definir los **'Validadores'** que necesitamos para cada campo de texto y un método **valida** que nos devuelva una instancia de la clase de utilidad que definimos al principio del ejercicio **ValidacionCompuesta** con el resultado de la validación de todos los campos.

Aquí tenemos pues, toda la lógica de validación de los campos de texto que podremos usar tanto el *'preview'* del formulario como el el propio ViewModel que gestione el *'screen'*. Esta lógica en aplicaciones más complejas debería ir en la **capa de dominio**.

Por ejemplo, para el texto introducido en el **TextField** de **teléfono** vamos a tener la siguiente *'validador'*:

```
val validadorTelefono = ValidadorCompuesto(ValidaTextoVacio("No puede estar vacío"))
    .add(ValidaLongitudMinimaTexto(9, "El teléfono debe tener 9 caracteres"))
    .add(ValidaLongitudMaximaTexto(18, "El teléfono debe tener 18 caracteres"))
    .add(ValidaTelefono("El teléfono no es válido"))
```

Porteriormente en el ViewModel que gestione el formulario, podremos usarlo de la siguiente forma:

```
var validacionContactoState = mutableStateOf(ValidacionContactoUiState())
var validadorContacto = ValidadorContacto() // Será inyectado

validacionContactoState = validacionContactoState.copy(
    validacionTelefono = validadorContacto.validadorTelefono.valida(telefono)
)
```

4. Definiremos un interfaz sellado **ContactoEvent.kt** dentro de **ui.features.formcontacto**, que abstraiga los eventos que se pueden producir en el formulario.

Vamos a tener en cuenta los siguientes eventos. Aunque ahora mismo no los vamos a gestionar todos de momento.

```
sealed interface ContactoEvent {
    data class OnChangeCategoria(val categoria: CategoriaUiState) : ContactoEvent
    data class OnChangeNombre(val nombre: String) : ContactoEvent
    data class OnChangeApellidos(val apellidos: String) : ContactoEvent
    data class OnChangeCorreo(val correo: String) : ContactoEvent
    data class OnChangeTelefono(val telefono: String) : ContactoEvent
    data class OnChangeFoto(val foto: ImageBitmap) : ContactoEvent
    data class OnSaveContacto(
        val onNavigateTrasFormContacto: (actualizaContactos : Boolean) -> Unit)
        : ContactoEvent
    data object OnDismissError : ContactoEvent
}
```


Con todo esto ya podemos definir el interfaz de usuario del formulario de contacto. Para ello, vamos a crear el componente `FormContactoScreen` dentro del paquete `ui.features.formcontacto`.

El interfaz de usuario del mismo es el siguiente:

```
@Composable
fun FormContactoScreen(
    contactoState: ContactoUiState,
    validacionContactoState: ValidacionContactoUiState,
    onContactoEvent: (ContactoEvent) -> Unit,
    onNavigateTrasFormContacto: (actualizaContactos: Boolean) -> Unit
)
```

Como ves, hemos agrupado estados relacionados y hemos intentado separar la gestión de los eventos. Por último, recibiremos una función `onNavigateTrasFormContacto` que nos permitirá navegar a la pantalla de lista de contactos una vez guardado el contacto y que de momento no vamos a darle funcionalidad.

Puedes descargar el siguiente enlace el código de este componente y añadirlo a tu proyecto.

[FormContactoScreen.kt](#)  y a continuación vamos a describir su funcionamiento antes de integrarlo en con su **ViewModel**.

Hemos dividido el componente en dos partes principales. Una es la Cabecera y otra el Cuerpo del formulario. Que distribuiremos dependiendo de la orientación del dispositivo con `BoxWithConstraints`. Además, **cada componente descompone el estado, pasando solo los datos que necesita y los eventos que gestiona**.

```
@Composable
fun FormContactoScreenVertical(...) {
    Column {
        CabeceraFoto(...)
        CuerpoFormulario(...)
    }
}
```

**CuerpoFormulario** es un **box** que contiene:

1. Un column con scroll vertical que contendrá el componente de selección de categorías y los **OutlinedTextField** para introducir los datos del contacto.
2. Un **pie** que mostrará superpuesto al column un **FAB** que nos permitirá guardar los cambios o un **SnackBar** que nos indicará si tenemos algún error de validación a resolver.

```
@Composable
private fun CuerpoFormulario(...) {
    Box {
        Column() { ... }
        Pie()
    }
}
```

👉 **Importante:** Se presenta un problema y es que los componentes que contiene **Pie** pueden tener una alineación diferente dentro del **BoxScope** del **Box** que contiene a **Pie**, pero si definimos **Pie** como un composable **'normal'** no podemos acceder al **BoxScope** del **Box** que lo contiene. Es por eso que haremos que **Pie** sea una función de extensión (composable) de **BoxScope**. Esto, me permitirá acceder al **BoxScope** del **Box** que lo contiene y por tanto alinear los componentes que contiene de la forma que queramos. Eso sí, es fácil de deducir que **nuestro componente Pie solo podrá ser usado dentro de un Box**.

```
@Composable
private fun BoxScope.Pie(...) {
    if (validacionContactoState.validacionContacto.hayError) {
        SnackBar(
5         modifier = Modifier.align(Alignment.BottomCenter),
            ...) {...}
    } else {
        ExtendedFloatingActionButton(
9         modifier = Modifier.align(Alignment.BottomEnd),
            ...
        )
    }
}
```

Termina de echarle un vistazo al código y verás que el resto es fácil de entender. Además se ha incorporado el componente de M3 **OutlinedIconButton** para cambiar la imagen del contacto. Sin embargo, de momento no tienen funcionalidad.

## Paso 6.- Creando el ViewModel del formulario de contacto

Básicamente deberemos pasar la funcionalidad del *'preview'* de test promocionado `FormContactoScreenTest` al mismo. Vemos

Crearemos la clase pues `ContactoViewModel.kt` 🔗 junto a su screen en el paquete `ui.features.formcontacto`. Veamos el código comentado...

```
@HiltViewModel
class ContactoViewModel @Inject constructor(
    // Inyectamos el repositorio y el validador de contactos
    private val contactoRepository: ContactoRepository,
    private val validadorContacto: ValidadorContacto
) : ViewModel() {
    class ContactoViewModelException(message: String) : Exception(message)

    // Este estado me indicará si estoy editando un contacto existente o creando uno nuevo
    var editandoContactoExistenteState: Boolean = false
        private set
    // Estados para el contacto y su validación. Solo son modificables desde el ViewModel
    var contactoState by mutableStateOf(ContactoUiState())
        private set
    var validacionContactoState by mutableStateOf(ValidacionContactoUiState())
        private set

    // Este método se llamará cuando queramos editar un contacto ya existente
    // justo antes de navegar o cargar el formulario FormContactoScreen.
    // Recibimos el id del contacto a editar y lo buscamos en el repositorio.
    fun setContactoState(idContacto: Int) {
        // Indicamos que estamos editando un contacto existente, para hacer un update al guardar.
        editandoContactoExistenteState = true
        val c: Contacto = contactoRepository.get(idContacto)
            ?: throw ContactoViewModelException("El id $idContacto no existe en la base de datos")
        // Tras buscarlo en el repositorio, lo convertimos a ContactoUiState y lo asignamos al estado.
        contactoState = c.toContactoUiState()
        // Se actualiza el estado de validación, aunque no debería haber errores.
        validacionContactoState = validadorContacto.valida(contactoState)
    }

    // Este método se llamará cuando queramos crear un nuevo contacto.
    // O borrar su estado.
    fun clearContactoState() {
        editandoContactoExistenteState = false
        contactoState = ContactoUiState()
        validacionContactoState = ValidacionContactoUiState()
    }
    ...
}
```

Por último vamos a definir el método que gestiona los eventos que se producen en el formulario. Para ello, vamos a definir el método `onContactoEvent` que recibe un evento de tipo `ContactoEvent` y que se encarga de gestionarlo.

```

fun onContactoEvent(e: ContactoEvent) {
    when (e) {
        is ContactoEvent.OnChangeCategoria -> {
            contactoState = contactoState.copy(categorias = e.categoria)
        }
        is ContactoEvent.OnChangeNombre -> {
            contactoState = contactoState.copy(nombre = e.nombre)
            validacionContactoState = validacionContactoState.copy(
                validacionNombre = validadorContacto.validadorNombre.valida(e.nombre)
            )
        }
        is ContactoEvent.OnChangeApellidos -> {
            contactoState = contactoState.copy(apellidos = e.apellidos)
            validacionContactoState = validacionContactoState.copy(
                validacionApellidos = validadorContacto.validadorApellidos.valida(e.apellidos)
            )
        }
        is ContactoEvent.OnChangeCorreo -> {
            contactoState = contactoState.copy(correo = e.correo)
            validacionContactoState = validacionContactoState.copy(
                validacionCorreo = validadorContacto.validadorCorreo.valida(e.correo)
            )
        }
        is ContactoEvent.OnChangeTelefono -> {
            contactoState = contactoState.copy(telefono = e.telefono)
            validacionContactoState = validacionContactoState.copy(
                validacionTelefono = validadorContacto.validadorTelefono.valida(e.telefono)
            )
        }
        is ContactoEvent.OnChangeFoto -> {
            contactoState = contactoState.copy(foto = e.foto)
        }
        is ContactoEvent.OnDismissError -> {
            validacionContactoState =
                validacionContactoState.copy(validacionContacto = Validacion(false))
        }
        is ContactoEvent.OnSaveContacto -> {
            // Validamos todo el contacto
            validacionContactoState = validadorContacto.valida(contactoState)
            // Si no hay errores, lo guardamos o actualizamos en el repositorio
            if (!validacionContactoState.validacionContacto.hayError) {
                val c: Contacto = contactoState.toContacto()
                if (editandoContactoExistenteState) {
                    contactoRepository.update(c)
                } else {
                    contactoRepository.insert(c)
                }
            }
            // Tras guardar seguiremos iremos a la pantalla correspondiente
        }
    }
}

```

```
        // por ahora no hacemos nada
        e.onNavigateTrasFormContacto(true)
    }
}
}
```



Si no tenemos las funciones `insert` y `update` definidas deberemos añadirlas al repositorio y al dao correspondiente.

```
// ContactoRepository.kt -----
fun insert(contacto: Contacto) = dao.insert(contacto.toContactoMock())
fun update(contacto: Contacto) = dao.update(contacto.toContactoMock())

// ContactoDaoMock.kt -----
fun insert(contacto: ContactoMock) = contactos.add(contacto)
fun update(contacto: ContactoMock) {
    val index = contactos.indexOfFirst { u -> u.id == contacto.id }
    if (index != -1) contactos[index] = contacto
}
```

## Paso 7.- Sustituyendo en el MainActivity la Lista por el Formulario

Cambiamos la creación del vm por el del `ContactoViewModel` y le indicamos que queremos editar un contacto existente con id 1.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val vm : ContactoViewModel by viewModels() // Creamos el ViewModel
    // Le indicamos que queremos editar un contacto existente con id 1
    vm.setContactoState(1)
    setContent {
        AgendaTheme {
            Surface(
                modifier = Modifier.fillMaxSize(),
                color = MaterialTheme.colorScheme.background
            ) {
                FormContactoScreen(
                    contactoState = vm.contactoState,
                    validacionContactoState = vm.validacionContactoState,
                    onContactoEvent = vm::onContactoEvent,
                    onNavigateTrasFormContacto = {}
                )
            }
        }
    }
}
```