

Visualizar lista de contactos de la Agenda

[Descargar estos apuntes](#)

'Codelab' guiado para crear una lista de contactos de la Agenda

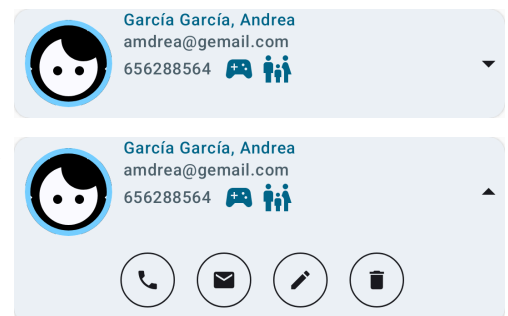
En el siguiente ejercicio partiremos del **ejercicio 3** del **Tema 3.2** donde definíamos un **componente imagen para la Agenda**. La idea del mismo es visualizar una lista de contactos de la Agenda.

Paso 1: Definir el componente para un 'item' en la lista

Deberá visualizar todos los datos de un contacto como su imagen, nombre, apellidos, teléfono y correo electrónico.

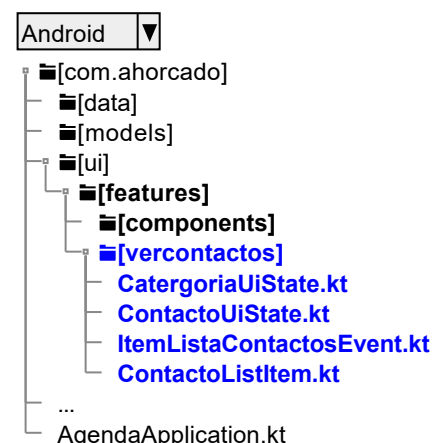
Además, mostrará en forma de iconos las categorías a las que pertenece el contacto.

Por último, al seleccionar un contacto se generará una pequeña animación para mostrarnos una serie de botones, **que aunque de momento no le vamos a dar funcionalidad**, más adelante me permitirá hacer llamadas, enviar un correo, editar o eliminar el contacto.



Crearemos pues en el paquete **features** un nuevo paquete **vercontactos** y dentro de él un nuevo componente **ContactoListItem.kt** y las clases asociadas **ContactoUiState.kt** y **ItemListaContactosEvent.kt** que serán las que nos permitan gestionar el estado del componente y los eventos que se produzcan sobre el mismo.

Veamos pues en primer lugar las clases **CatergoriaUiState.kt** y **ContactoUiState.kt** que será la que nos permita gestionar el estado del componente.



CategoriaUiState.kt

Definimos un primer data class denominado **CatergoriaUiState** que nos permitirá gestionar el estado de las categorías de un contacto. Para ello definimos una serie de propiedades booleanas que nos indicarán si el contacto pertenece o no a una categoría y una serie de propiedades de tipo **ImageVector** que nos permitirán almacenar el icono asociado a cada categoría.

```
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  
)
```

ContactoUiState.kt

Definimos ahora un data class denominado **ContactoUiState** que nos permitirá gestionar el estado de un contacto como ya hicimos en otra parte del curso. Usaremos la clase **CatergoriaUiState** para gestionar las categorías a las que pertenece el contacto.

```
data class ContactoUiState(  
    val id: Int = Instant.now().epochSecond.toInt(),  
    val nombre: String = "",  
    val apellidos: String = "",  
    val foto: ImageBitmap? = null,  
    val correo: String = "",  
    val telefono: String = "",  
    val categorias : CatergoriaUiState = CatergoriaUiState(),  
)
```

En estas clases podemos definir métodos de extensión que me ayuden a transformar los datos de un **Contacto** en el modelo en un objeto de tipo **ContactoUiState**.

🔴 **Nota:** Estas clases de estado posiblemente puedan ser reutilizadas en el futuro para otros componentes.

ItemListaContactosEvent.kt

Definiendo un `sealed class` denominado `ItemListaContactosEvent` que nos permitirá gestionar los eventos que se produzcan sobre el componente. De momento solo definiremos un par de eventos denominado `OnClickContacto` que nos permitirá gestionar el evento de pulsación sobre un contacto y `onDeleteContacto` para gestionar la eliminación del mismo. Pero más adelante tendremos más eventos como por ejemplo `onEditContacto` para editar el contacto seleccionado y al utilizar una clase sellada será más fácil de gestionar la elevación de los eventos a través de *compose*.

```
sealed class ItemListaContactosEvent {  
    data class OnClickContacto(val contacto : ContactoUiState) : ItemListaContactosEvent()  
    object onDeleteContacto : ItemListaContactosEvent()  
}
```

ContactoListItem.kt



En el siguiente fuente encontraremos la propuesta de interfaz de las imágenes anteriores para el componente `ContactoListItem`. Vamos a comentar cómo hemos estructurado el mismo. Comentando algunos de los 'composables' en los que se ha dividido el componente...

```
// Muestra los iconos con las categorías del contacto, por eso  
// recibe únicamente el estado de las categorías.  
@Composable  
fun Categorias(categoriasState: CategoriaUiState)
```

Por ejemplo si tenemos a `true` las categorías `Amigos` y `Familia`. Mostrará la imagen de la derecha.



```
// Muestra todos los datos del contacto menos la imagen asociada.
// Por esa razón los recibe cada uno de los datos del contacto
// excepto la imagen.
@Composable
fun DatosContacto(
    modifier: Modifier = Modifier,
    nombre: String,
    apellidos: String,
    correo: String,
    telefono: String,
    categorias: CategoriaUiState
)
```

García García, Andrea
andrea@gmail.com
656288564  

```
// Muestra OutlinedIconButton con los iconos de las
// acciones posibles sobre el contacto seleccionado.
@Composable
fun AccionesContacto(
    onLlamarClickado: () -> Unit = {},
    onCorreoClickado: () -> Unit = {},
    onEditClickado: () -> Unit = {},
    onDeleteClickado: () -> Unit = {}
)
```

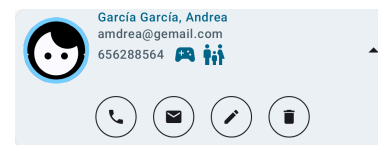


🔴 **Nota:** Los siguientes *composables* tendrán animaciones. No vamos a entrar en ellas en el presente curso por falta de tiempo. Pero, puedes aprender todos los conceptos relacionados con la misma en el [siguiente videotutorial](#).

```
@OptIn(ExperimentalLayoutApi::class)
// Muestra la imagen del contacto, los datos del contacto
// y un pequeño icono que tendrá una animación de rotación
// cuando el contacto esté seleccionado.
@Composable
fun ContenidoPrincipalCardContacto(
    contactoUiState: ContactoUiState,
    seleccionadoState: Boolean,
    modifier: Modifier = Modifier
)
```



Por último, vamos a crear la tarjeta que contendrá el componente `ContactoListItem`. En ella definiremos un `ElevatedCard` que contendrá en una columna el componente `ContenidoPrincipalCardContacto` y al estar seleccionada (`seleccionadoState == true`) mostrará el componente `AccionesContacto`. Esto provocará un cambio de tamaño del contenido del layout que realizaremos de forma programática con la función `animateContentSize`.



```

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ContactoListItem(
    modifier: Modifier = Modifier,
    contactoUiState: ContactoUiState,
    seleccionadoState: Boolean,
    onContactoClicked: (ContactoUiState) -> Unit,
    onEditClicked: () -> Unit,
    onDeleteClicked: () -> Unit
) = ElevatedCard(
    onClick = { onContactoClicked(contactoUiState) },
    modifier = modifier.then(
        Modifier
            .fillMaxWidth()
            .animateContentSize(
                animationSpec = tween(
                    durationMillis = 300,
                    easing = LinearOutSlowInEasing
                )
            )
    )
) {
    Column {
        ContenidoPrincipalCardContacto(
            contactoUiState = contactoUiState,
            seleccionadoState = seleccionadoState
        )
        if (seleccionadoState)
            AccionesContacto(
                onEditClicked = onEditClicked,
                onDeleteClicked = onDeleteClicked
            )
    }
}

```

Paso 2: Crear el feature ListaContactosScreen

Vamos a definir ahora dentro del paquete `vercontactos` un *Screen* que visualice la lista de contactos usando `LazyColumn`. Crearemos pues `ListaContactosScreen.kt` y definiremos el siguiente código, donde vamos a usar `ContactoListItem` para visualizar cada uno de los contactos de la lista.

```

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun ListaContactosScreen(
    modifier: Modifier = Modifier,
    contactosState: List<ContactoUiState>,
    contactoSeleccionadoState: ContactoUiState?,
    onContactoClicked: (ContactoUiState) -> Unit,
    onAddClicked: () -> Unit = {},
    onEditClicked: () -> Unit = {},
    onDeleteClicked: () -> Unit = {}
) {
    Box(modifier = modifier.then(Modifier.fillMaxSize())) {
        LazyColumn(
            contentPadding = PaddingValues(all = 4.dp),
            verticalArrangement = Arrangement.spacedBy(4.dp)
        ) {
            items(
                contactosState,
                key = { it.id }
            ) { contacto ->
                ContactoListItem(
                    modifier = Modifier.animateItemPlacement(),
                    contactoUiState = contacto,
                    seleccionadoState = contactoSeleccionadoState
                        ?.let { it.id == contacto.id } ?: false,
                    onContactoClicked = onContactoClicked,
                    onEditClicked = onEditClicked,
                    onDeleteClicked = onDeleteClicked
                )
            }
        }
        FloatingActionButton(
            modifier = Modifier
                .align(Alignment.BottomEnd)
                .padding(8.dp),
            onClick = onAddClicked
        ) {
            Icon(imageVector = Icons.Default.Add, contentDescription = "Crear Contacto")
        }
    }
}

```


Vamos a definir un preview para la lista de contactos que acceda directamente a `ContactoDaoMock` . Este preview además de para visualizar la lista nos puede servir para tener una proto-implementación de la lógica de negocio de la aplicación que irá en el `ViewModel` .

```
@Preview(
    name = "PORTRAIT",
    device = "spec:width=360dp,height=800dp,dpi=480",
    showBackground = true
)
@Composable
fun ListaContactosScreenTest() {
    val contactos = ContactoDaoMock().get().map {
        ContactoUiState(
            id = it.id,
            foto = it.foto?.let { Imagenes.base64ToBitmap(it) },
            nombre = it.nombre,
            apellidos = it.apellidos,
            telefono = it.telefono,
            correo = it.correo,
        )
    }

    // Definimos un estado que nos permitirá gestionar el contacto
    // actualmente seleccionado y su manejador
    var contactoSeleccionadoState: ContactoUiState? by remember { mutableStateOf(null) }
    var onContactoClicked: (ContactoUiState) -> Unit = { c ->
        contactoSeleccionadoState =
            if (contactoSeleccionadoState == null
                || c.id != contactoSeleccionadoState!!.id) c.copy()
            else null
    }

    ListaContactosScreen(
        modifier = Modifier.fillMaxSize(),
        contactosState = contactos,
        contactoSeleccionadoState = contactoSeleccionadoState,
        onContactoClicked = onContactoClicked,
        onAddClicked = {},
        onEditClicked = {},
        onDeleteClicked = {}
    )
}
```

Paso 3: Definir el ViewModel

Ahora vamos a definir un `ViewModel` para la pantalla de la lista de contactos. Para ello crearemos dentro del paquete `vercontactos` el fuente `ListaContactosViewModel.kt`.

👉 **Importante:** Si estas utilizando inyección de dependencias puedes seguir usándola aunque en este caso no lo vamos a hacer por simplificar. Además, si la has implementado es un buen momento para hacerlo y terminar de entender el concepto.

```

class ListaContactosViewModel : ViewModel() {
    // Creamos un repositorio de contactos. Mejor inyectarlo con Hilt
    private val contactoRepository = ContactoRepository();
    // Estado que nos permitirá gestionar el contacto actualmente
    // seleccionado, valdrá null si no hay ninguno
    var contatoSleccionadoState: ContactoUiState? by mutableStateOf(null)
        private set
    // Creamos un estado con la lista de contactos y llamamos a una
    // función privada que se encargará del mapeo entre el Modelo y el UiState
    private var _listaContactosState by mutableStateOf(getContactos())
    // El getter solo nos permite acceder al estado pero no modificarlo
    val listaContactosState: List<ContactoUiState>
        get() = _listaContactosState
    // Función privada que nos permite obtener la lista de contactos
    // mapeando entre el Modelo y el UiState
    private fun getContactos(): MutableList<ContactoUiState> = contactoRepository.get()
        .map { it.toContactoUiState() }.toMutableList()
    // Método que nos permite gestionar los eventos que se produzcan en
    // nuestro caso la selección de un contacto y su eliminación
    fun onItemListaContatoEvent(e: ItemListaContactosEvent) {
        when (e) {
            is ItemListaContactosEvent.OnClickContacto -> {
                contatoSleccionadoState =
                    if (contatoSleccionadoState?.id != e.contacto.id) e.contacto else null
            }
            is ItemListaContactosEvent.OnDeleteContacto -> {
                contactoRepository.delete(contatoSleccionadoState!!.id)
                // Una vez hemos borrado deberemos actualizar la lista de
                // contactos para que se actualice el estado de la misma
                _listaContactosState = getContactos()
            }
        }
    }
}

```

Paso 4: Vamos usar nuestro Screen con el ViewModel

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val vm : ListaContactosViewModel by viewModels()
        setContent {
            AgendaTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    ListaContactosScreen(
                        contactosState = vm.listaContactosState,
                        contactoSeleccionadoState = vm.contatoSleccionadoState,
                        onContactoClicked = { c ->
                            vm.onItemListaContatoEvent(
                                ItemListaContactosEvent.OnClickContacto(c))
                        },
                        onDeleteClicked = {
                            vm.onItemListaContatoEvent(
                                ItemListaContactosEvent.OnDeleteContacto)
                        }
                    )
                }
            }
        }
    }
}
```

La imagen final de la aplicación será la siguiente. En ella podemos ver que al seleccionar un contacto se nos muestran una serie de botones que nos permitirán realizar acciones sobre el mismo. En este caso solo tenemos implementada la eliminación del contacto.

Además, tenemos un botón flotante que nos permitirá añadir un nuevo contacto. En este caso no tenemos implementada la funcionalidad pero más adelante lo haremos.

