Añadiendo Scaffold la Agenda

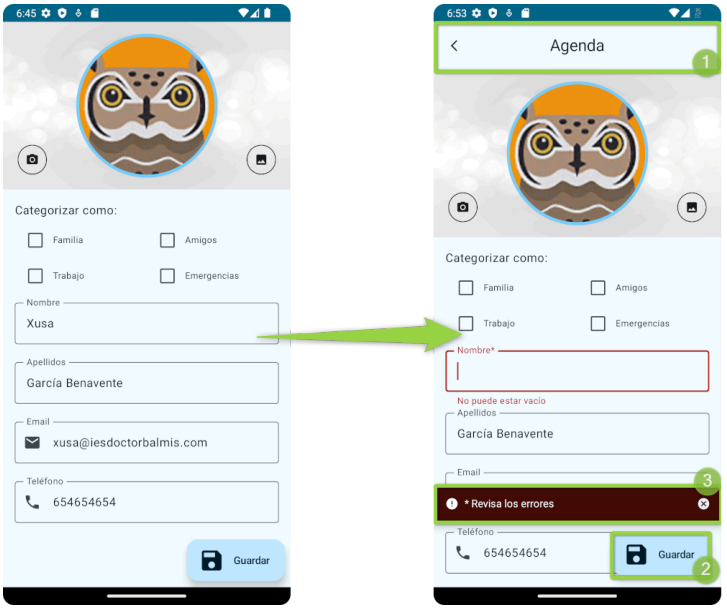
Descargar estos apuntes

🎓 Caso de Estudio

Vamos a partir de nuestra Agenda de Contactos con Intents en la que teníamos dos pantallas, una para mostrar los contactos y otra para añadirlos y vamos a añadir a ambas algún tipo de Scaffold.

Paso 1: Añadir Scaffold al formulario de añadir o editar un contacto.

Empezaremos con el formulario de contactos. Para ello, vamos a pasarlo añadir un Sacffold como el de la imagen donde tenemos:



1. Un `TopAppBar` con un título y un botón de navegación para volver atrás.
2. El `FAB` de guardar los datos del contacto, pasará a estar gestionado por el Scaffold.
3. Además el `SnackBarHost` del Scaffold, nos permitirá mostrar mensajes de error a través de un `SnackBar` personalizado tal y como se indica en la imagen.

Vemos pues el proceso de cómo realizarlo:

1. Vamos a borrar la función composable `BoxScope.Pie` que emitía el FAB o el SnackBar, pues ahora lo hará el Scaffold. **Los parámetros que recibiese `CuerpoFormulario` para usar en dicho composable ya no serán necesarios**.
2. Definiremos un composable que emitirá nuestro TopAppBar en nuestro caso usaremos `CenterAlignedTopAppBar` de Material 3 pues no vamos a tener acciones. Además, recibirá el callback para volver atrás en la navegación.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun FormContactoTopAppBar(
    scrollBehavior: TopAppBarScrollBehavior = TopAppBarDefaults.pinnedScrollBehavior(),
    onNavigateTrasFormContacto: (actualizaContactos: Boolean) -> Unit
) {
    CenterAlignedTopAppBar(
        scrollBehavior = scrollBehavior,
        title = {
            Text(text = stringResource(id = R.string.app_name))
        },
        navigationIcon = {
            IconButton(onClick = { onNavigateTrasFormContacto(false) }) {
                Icon(
                    painter = Filled.getArrowBackIosIcon(),
                    contentDescription = "Volver a lista de contactos",
                    modifier = Modifier.size(ButtonDefaults.IconSize),
                )
            }
        },
    )
}
```

3. Definiremos un composable que emitirá nuestro `FAB` del Scaffold para guardar los datos del contacto. Recibirá el callback para guardar los datos del contacto.

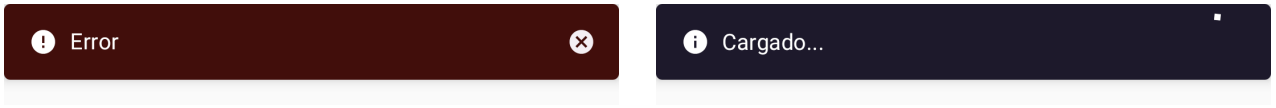
```
@Composable
fun FabGuardar(onGuardarContacto: () -> Unit) {
    ExtendedFloatingActionButton(
        text = { Text(text = "Guardar") },
        icon = {
            Icon(
                painter = Filled.getSaveIcon(),
                contentDescription = "Localized description",
                modifier = Modifier.size(FloatingActionButtonDefaults.LargeIconSize),
            )
        },
        onClick = onGuardarContacto
    )
}
```

4. Para la gestión del `SnackBar`, tenemos la clase `InformacionEstadoUiState` definida en la librería de componentes del módulo en `com.github.pmdmiesbalmis.components.manejo_errores.InformacionEstadoUiState` **InformacionEstadoUiState.kt**. Esta nos permita gestionar mensajes de estado tales como información o error en la UI. En ella, **tendremos tres estados para mostrar el SnackBar**:
   - Uno para ocultarlo
   - Uno para mostrar información.
   - Uno para mostrar un error.

   Esta clase gestionará el estado de visualización del componente definido `com.github.pmdmiesbalmis.components.ui.composables` **SnackbarCommon.kt**. Esta se *emitirá* dentro del `SnackBarHost` del Scaffold.

   ```
   fun SnackbarCommon(informacionEstado: InformacionEstadoUiState)
   ```

   Básicamente este composable nos permitirá mostrar un SnackBar con un mensaje de información con una barra de progreso circular o un SnackBar con un mensaje de error y un botón para cerrarlo.



5. Ya estamos preparados para añadir el Scaffold a `FormContactoScreen` que quedará de la siguiente manera...
   Primero redefinimos el prototipo pasando **un nuevo parámetro** `InformacionEstadoUiState` para nuestro Scaffold.

```
OptIn(ExperimentalMaterial3Api::class)
@Composable
fun FormContactoScreen(
    contactoState: ContactoUiState,
    validacionContactoState: ValidacionContactoUiState,
    informacionEstado: InformacionEstadoUiState,
    onContactoEvent: (ContactoEvent) -> Unit,
    onNavigateTrasFormContacto: (actualizaContactos: Boolean) -> Unit
) {
    val scrollBehavior = TopAppBarDefaults.enterAlwaysScrollBehavior()
    val snackbarHostState = remember { SnackbarHostState() }
    ...
}
```

   Llamamos al `LaunchedEffect` para gestionar el estado del `SnackBar` en el `SnackBarHost` del `Scaffold` a través de la función `CorrutinaGestionSnackBar` definida en `SnackbarCommon.kt`.

```
CorrutinaGestionSnackBar(
    snackbarHostState = snackbarHostState,
    informacionEstado = informacionEstado
)
...
```

Ya podemos definir el Scaffold con su `TopAppBar`, `FAB` y `SnackBarHost`...

```
Scaffold(
    modifier = Modifier.nestedScroll(scrollBehavior.nestedScrollConnection),
    topBar = {
        FormContactoTopAppBar(
            scrollBehavior = scrollBehavior,
            onNavigateTrasFormContacto = onNavigateTrasFormContacto
        )
    },
    floatingActionButton = {
        FabGuardar(
            onGuardarContacto = {
                onContactoEvent(
                    ContactoEvent.OnSaveContacto(
                        onNavigateTrasFormContacto
                    )
                )
            }
        )
    },
    snackbarHost = {
        SnackbarHost(hostState = snackbarHostState) {
            // Emitiremos un SanackBar u otro dependiendo
            // el valor de informacionEstado
            SnackbarCommon(informacionEstado = informacionEstado)
        }
    }
)
{
    BoxWithConstraints(
        modifier = Modifier
            .fillMaxSize()
            .padding(it)
    ) {
        // Contenido del formulario anterior
    }
}
```

6. Por último, nos queda gestionar el estado de tipo `InformacionEstadoUiState` en `ContactoViewModel.kt`, sustituyéndolo el actual `verSnackBarState`. Es por ello que borraremos `verSnackBarState` y definiremos...

```
var informacionEstadoState: InformacionEstadoUiState by mutableStateOf(
    InformacionEstadoUiState.Oculta())
    private set
```

Ahora en la gestión de `ContactoEvent` al producirse un error en la validación.

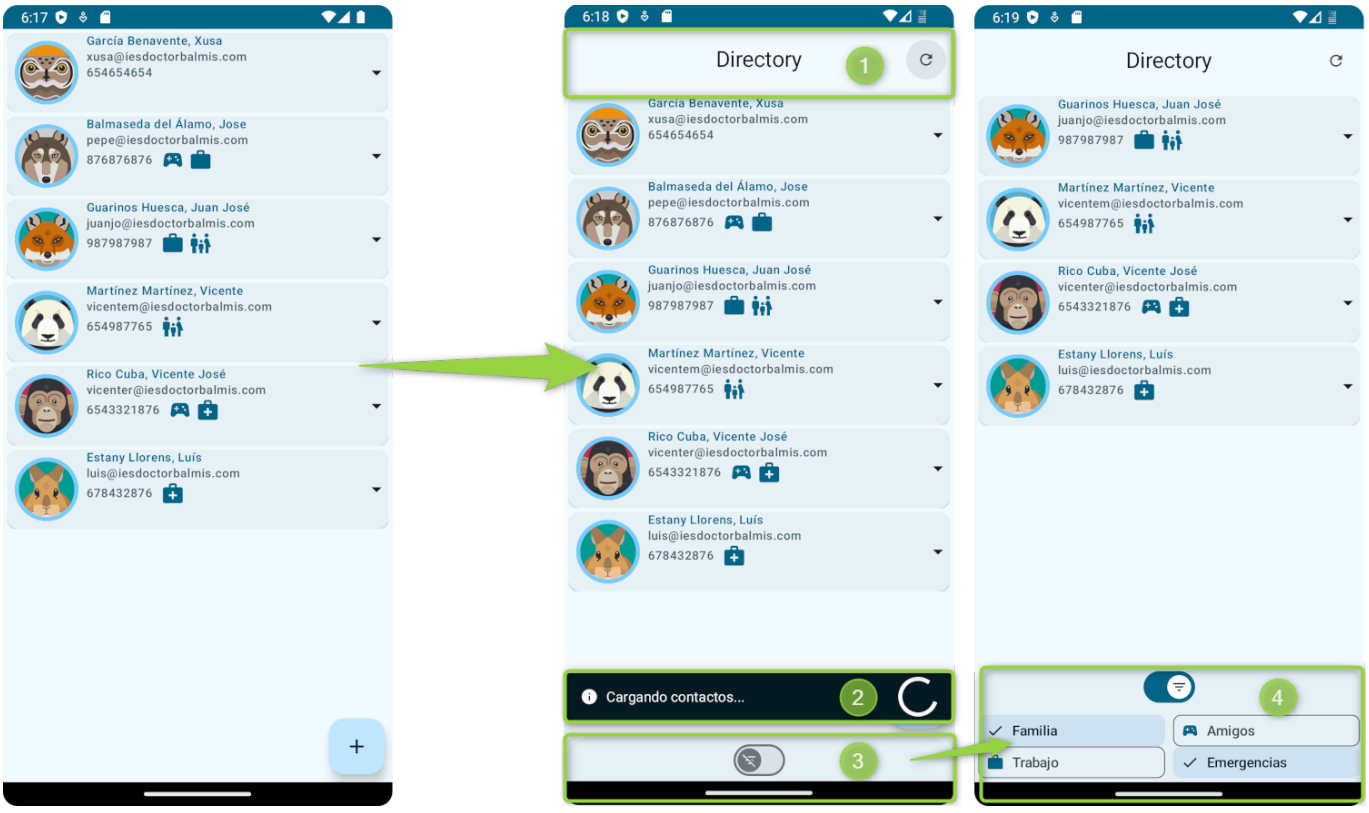
```
    ...
    is ContactoEvent.OnDismissError -> {
        validacionContactoState = ValidacionContactoUiState()
    }

    is ContactoEvent.OnSaveContacto -> {
        validacionContactoState = validadorContacto.valida(contactoState)
        if (!validacionContactoState.hayError) {
            ...
            // Guardamos o editamos el contacto
        } else {
            informacionEstadoState = InformacionEstadoUiState.Error(
                mensaje = validacionContactoState.mensajeError!!,
                onDismiss = { informacionEstadoState = InformacionEstadoUiState.Oculta()
            )
        }
    }
    ...
```

Ya solo quedaría pasar este estado desde el VM al `FormContactoScreen` en la `MainActivity`. Puedes definir este composable para probarlo.

```
    @Composable
    fun VerFormContacto(vm : ContactoViewModel = hiltViewModel() ) {
        FormContactoScreen(
            contactoState = vm.contactoState,
            validacionContactoState = vm.validacionContactoState,
            informacionEstado = vm.informacionEstadoState,
            onContactoEvent = vm::onContactoEvent,
            onNavigateTrasFormContacto = {}
        )
    }
```

Paso 2: Añadir Scaffold la pantalla de listar contactos

Vamos a pasar la pantalla de listar contactos a un `BottomSheetScaffold` como el de la imagen...



En el vamos a colocar:

1. Un `TopAppBar` con un título y un botón de acción para actualizar la lista de contactos.
2. Un SnackBarHost para mostrar mensajes de estado donde utilizaremos el `SnackbarCommon` que definimos en el paso anterior.
3. Una hoja inferior desplegable modal (`BottomSheet`) que contendrá el componente con `FilterChips` que definimos en temas anteriores y que se encuentra en el `.ui.features.components.SeleccionCategorias.kt` . Este composable nos permitirá filtrar los contactos por categorías.
4. Además, definiremos un `sheetDragHandle` específico para nuestra hoja inferior, que será un componente `Switch` que nos permitirá activar o desactivar los filtros.

> **Nota**
>
> Puesto que al usar un `BottomSheetScaffold` , no tendremos slot para el FAB, este seguirá mostrándose en el composable con el contenido principal como sucedía hasta ahora.

También, modificaremos `ListaContatosViewModel` para que gestione los nuevos estados de la pantalla, así como el filtrado de los contactos y la acción recarga de los mismos.

Vemos pues el proceso de cómo realizarlo:

1. Primero definiremos el `sheetDragHandle` personalizado para nuestro `BottomSheet`. Como hemos comentado, **recibirá un estado que me indicará si el filtrado está activo o no** y un callback para cambiar dicho estado.

```
@Composable
fun SwitchActivacionFiltrado(
    modifier: Modifier = Modifier,
    filtradoActivoState: Boolean,
    onActivarFiltradoClicked: (Boolean) -> Unit
) {
    Switch(
        modifier = modifier,
        checked = filtradoActivoState,
        onCheckedChange = onActivarFiltradoClicked,
        thumbContent = {
            Icon(
                painter = if (filtradoActivoState) Filled.getFilterListIcon()
                else Filled.getFilterListOffIcon(),
                contentDescription = null,
                modifier = Modifier.size(SwitchDefaults.IconSize),
            )
        }
    )
}
```

2. Ahora definiremos la `CenterAlignedTopAppBar` para el `BottomSheetScaffold`.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListaContactosTopAppBar(
    scrollBehavior: TopAppBarScrollBehavior = TopAppBarDefaults.pinnedScrollBehavior(),
    onActualizaContactos: () -> Unit,
) {
    CenterAlignedTopAppBar(
        scrollBehavior = scrollBehavior,
        title = {
            Text(text = stringResource(id = R.string.app_name))
        },
        actions = {
            IconButton(onClick = onActualizaContactos) {
                Icon(
                    painter = Filled.getRefreshIcon(),
                    contentDescription = "Actualiza contactos",
                    modifier = Modifier.size(ButtonDefaults.IconSize),
                )
            }
        }
    )
}
```

3. Vamos ahora de definir la diferentes partes de nuestro BottomSheetScaffold. Teniendo en cuenta que toda la lógica va estar en nuestro ViewModel. En primer lugar definiremos el prototipo, los estados que manejará localmente y los `LandchedEffect` para gestionar que nuestro `BottomSheet` siempre esté visible y el mostrar o ocultar el `SnackBar` en función del estado de `InformacionEstadoUiState`.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListaContactosScreen(
    modifier: Modifier = Modifier,
    contactosState: List<ContactoUiState>,
    contactoSeleccionadoState: ContactoUiState?,
    filtradoActivoState: Boolean,
    filtroCategoriaState: CatergoriaUiState,
    informacionEstadoState: InformacionEstadoUiState,
    onActualizaContactos: () -> Unit,
    onActivarFiltradoClicked: (Boolean) -> Unit,
    onFiltroModificado: (CatergoriaUiState) -> Unit,
    onContactoClicked: (ContactoUiState) -> Unit,
    onAddClicked: () -> Unit,
    onEditClicked: () -> Unit,
    onDeleteClicked: () -> Unit
) {
    val scrollBehavior = TopAppBarDefaults.pinnedScrollBehavior()
    val scaffoldState = rememberBottomSheetScaffoldState()
    val snackbarHostState = remember { SnackbarHostState() }
    val scope = rememberCoroutineScope()

    // Corrutina que muestra el bottom sheet si no está visible
    LaunchedEffect(
        key1 = !scaffoldState.bottomSheetState.isVisible,
        block = { scaffoldState.bottomSheetState.show() }
    )

    // Gestión del estado del SnackBar como en la pantalla anterior
    CorrutinaGestionSnackBar(
        snackbarHostState = snackbarHostState,
        informacionEstado = informacionEstadoState
    )
    ...
```

Veamos las ranuras del `BottomSheetScaffold`:

```
        BottomSheetScaffold(
            sheetContent = {
                SeleccionCategoriasConFilterChip(
                    categoriaState = filtroCategoriaState,
                    onCategoriaChanged = onFiltroModificado
                )
            },
            sheetDragHandle = {
                SwitchActivacionFiltrado(
                    filtradoActivoState = filtradoActivoState,
                    // Cuando se desactive el filtrado,
                    // cerraremos nuestro BottomSheet
                    onActivarFiltradoClicked = { filtradoOn ->
                        onActivarFiltradoClicked(filtradoOn)
                        scope.launch {
                            if (!filtradoOn) {
                                scaffoldState.bottomSheetState.show()
                            }
                        }
                    }
                )
            },
            sheetPeekHeight = 45.dp,
            sheetShape = BottomSheetDefaults.ExpandedShape,
            scaffoldState = scaffoldState,
            modifier = Modifier.nestedScroll(scrollBehavior.nestedScrollConnection),
            topBar = {
                ListaContactosTopAppBar(
                    scrollBehavior = scrollBehavior,
                    onActualizaContactos = onActualizaContactos
                )
            },
            snackbarHost = {
                SnackbarHost(hostState = snackbarHostState) {
                    SnackbarCommon(informacionEstado = informacionEstadoState)
                }
            }
        ) {
            // Contenido principal de la pantalla con la lista y el FAB
        }
    }
```

Ahora realizaremos los cambios en el ViewModel en `ListaContactosViewModel.kt` para que gestione los nuevos estados de la pantalla, así como el filtrado de los contactos y la acción recarga de los mismos.

Añadiremos pues las siguientes propiedades...

```
var filtradoActivoState: Boolean by mutableStateOf(false)
    private set

var filtroCategoriaState: CatergoriaUiState
by mutableStateOf(CatergoriaUiState())
    private set

var informacionEstadoState: InformacionEstadoUiState
by mutableStateOf(InformacionEstadoUiState.Oculta())
    private set
```

Añadiremos una nueva **función de extensión para filtrar los contactos** que cumplan con el predicado definido en el estado `filtroCategoriaState`.

```
private fun ContactoUiState.predicadoFiltro(): Boolean {
    var cumpleFiltro = !filtradoActivoState

    if (!cumpleFiltro) // Está en alguna de las categoirias seleccionadas
        cumpleFiltro = categorias.familia && filtroCategoriaState.familia ||
                categorias.amigos && filtroCategoriaState.amigos ||
                categorias.trabajo && filtroCategoriaState.trabajo ||
                categorias.emergencias && filtroCategoriaState.emergencias

    if (!cumpleFiltro) // El contacto no está en ninguna de las categorias seleccionadas
        cumpleFiltro = !categorias.familia && !filtroCategoriaState.familia &&
                !categorias.amigos && !filtroCategoriaState.amigos &&
                !categorias.trabajo && !filtroCategoriaState.trabajo &&
                !categorias.emergencias && !filtroCategoriaState.emergencias

    return cumpleFiltro
}
```

```
fun onActivarFiltradoClicked() {
    filtradoActivoState = !filtradoActivoState
    cargaContactos()
}

fun onFiltroModificado(categorias: CatergoriaUiState) {
    filtroCategoriaState = categorias.copy()
    cargaContactos()
}

private fun deseleccionaContacto() {
    contatoSleccionadoState = null
}

// Función de suspensión que accede a los datos del repositorio y
// si es necesario algún filtro lo aplica.
private suspend fun getContactos(): List<ContactoUiState> = contactoRepository.get()
    .map { it.toContactoUiState() }
    .filter { c -> c.predicadoFiltro() }
    .toList()

// Función que se encarga de cargar los contactos y gestionar los estados
// notificados en el SnackBarHost. Si se produce algún error, se mostrará
// un SnackBar con un mensaje de error.
fun cargaContactos() {
    deseleccionaContacto()
    informacionEstadoState = InformacionEstadoUiState.Informacion(
        mensaje = "Cargando contactos...",
        muestraProgreso = true
    )
    viewModelScope.launch {
        runCatching {
            contactosState = getContactos()
        }.onSuccess {
            informacionEstadoState = InformacionEstadoUiState.Oculta()
        }.onFailure {
            Log.d("ListaContactosViewModel", "Cargando Contactos: ${it.localizedMessage}")
            informacionEstadoState = InformacionEstadoUiState.Error(
                mensaje = "Error al cargar los contactos",
                onDismiss = { informacionEstadoState = InformacionEstadoUiState.Oculta() }
            )
        }
    }
```

```
}

// Nada más crearse el viewModel cargamos los contactos
init { cargaContactos() }
```

Por último, nos queda la gestión del borrado de un contacto gestionando los mensajes de estado y error en el `SnackBarHost` de forma similar a como lo hemos hecho al cargar los contactos.

```
fun onItemListaContatoEvent(e: ItemListaContactosEvent) {
    when (e) {
        is ItemListaContactosEvent.OnClickContacto -> {
            contatoSleccionadoState =
                if (contatoSleccionadoState?.id != e.contacto.id) e.contacto else null
        }

        is ItemListaContactosEvent.OnDeleteContacto -> {
            informacionEstadoState = InformacionEstadoUiState.Informacion(
                mensaje = "Borrando a ${contatoSleccionadoState!!.nombre}...",
                muestraProgreso = true
            )
            viewModelScope.launch {
                runCatching {
                    contactoRepository.delete(contatoSleccionadoState!!.id)
                    contactosState = getContactos()
                }.onSuccess {
                    informacionEstadoState = InformacionEstadoUiState.Oculta()
                }.onFailure {
                    Log.d(
                        "ListaContactosViewModel",
                        "Borrando Contacto: ${it.localizedMessage}"
                    )
                    informacionEstadoState = InformacionEstadoUiState.Error(
                        mensaje =
                        "Error al borrar el contacto ${contatoSleccionadoState!!.nombre}",
                        onDismiss = {
                            informacionEstadoState = InformacionEstadoUiState.Oculta()
                        }
                    )
                }
            }
        }
    }
}
```

Ya solo quedaría pasar estos estados desde el VM al `ListaContactosScreen` en la `MainActivity`. Puedes definir este composable para probarlo.

```
@Composable
fun VerListaContactos(vm : ListaContactosViewModel = hiltViewModel()) {
    ListaContactosScreen(
        contactosState = vm.contactosState,
        contactoSeleccionadoState = vm.contatoSleccionadoState,
        filtradoActivoState = vm.filtradoActivoState,
        filtroCategoriaState = vm.filtroCategoriaState,
        informacionEstadoState = vm.informacionEstadoState,
        onActualizaContactos =  { vm.cargaContactos() },
        onActivarFiltradoClicked = { vm.onActivarFiltradoClicked() },
        onFiltroModificado = { categorias -> vm.onFiltroModificado(categorias) },
        onContactoClicked = { c ->
            vm.onItemListaContatoEvent(ItemListaContactosEvent.OnClickContacto(c))
        },
        onAddClicked = {},
        onEditClicked = {},
        onDeleteClicked = {
            vm.onItemListaContatoEvent(ItemListaContactosEvent.OnDeleteContacto)
        }
    )
}
```

Solución

Si te surge alguna duda o tienes dificultades para completar este caso de estudio. Puedes descargar la solución de este caso de estudio del siguiente enlace: **propuesta de solución**