Añadiendo Navegación la Agenda

Descargar estos apuntes

🎓 Caso de Estudio

Vamos a partir de nuestra Agenda de Contactos con Scaffold del caso de estudio anterior en la que teníamos dos pantallas y vamos a hacer una navegación simple entre ellas.

> **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**

Paso 1: Definiendo las rutas o destinos de navegación

En el paquete `.ui.navagation` vamos a definir la ruta raíz o pantalla inicial de nuestra navegación que será la pantalla de **Contactos** definida en `ListaContactosScreen.kt`. Para ello definimos en el paquete que acabamos de definir el archivo `ListaContactosRoute.kt` donde definiremos las funciones de extensión con las rutas a la pantalla de **Contactos** y que nos permitan navegar a la misma.

```
const val ListaContactosGraphRoute = "lista_contactos"

// Definimos la función de extensión para navegar a la pantalla de Contactos
// aunque después no la usaremos por ser la ruta raíz.
fun NavController.navigateToListaContactos(navOptions: NavOptions? = null) {
    this.navigate(ListaContactosGraphRoute, navOptions)
}
// Definimos la función de extensión para definir las rutas a dicha pantalla
// y en esta ocasión en lugar de pasar el NavController, le pasamos los
// callbacks que nos permitirán navegar desde la definición del NavHost.
fun NavGraphBuilder.listaContactosScreen(
    vm : ListaContactosViewModel,
    onNavigateCrearContacto: () -> Unit,
    onNavigateEditarContacto: (idContacto: Int) -> Unit
) { ...
```

Para definir nuestra ruta, podemos partir de la función que emitía dicha pantalla en el el `MainActivity` del caso de estudio anterior.
Fíjate que al definir los callbacks de **editar** y **crear contacto** les pasamos los callbacks que nos permitirán navegar a la pantalla `FormContactoScreen.kt` de una forma u otra.

```
        composable(
            route = ListaContactosGraphRoute,
            arguments = emptyList()
        ) {
            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 = {
                    vm.onItemListaContatoEvent(
                        ItemListaContactosEvent.OnCrearContacto(
                            onNavigateCrearContacto
                        )
                    )
                },
                onEditClicked = {
                    vm.onItemListaContatoEvent(
                        ItemListaContactosEvent.OnEditContacto(
                            onNavigateEditarContacto
                        )
                    )
                },
                onDeleteClicked = {
                    vm.onItemListaContatoEvent(ItemListaContactosEvent.OnDeleteContacto)
                }
            )
        }
    }
```

Para que compile el código anterior, tendremos que haber definido dos nuevos tipos de evento en `ItemListaContactosEvent.kt` para crear y editar un contacto.

```
sealed class ItemListaContactosEvent {
    data class OnClickContacto(val contacto : ContactoUiState)
        : ItemListaContactosEvent()
    data class OnCrearContacto(
        val onNavigateCrearContacto: () -> Unit
    ) : ItemListaContactosEvent()
    data class OnEditContacto(
        val onNavigateEditarContacto: (idContacto: Int) -> Unit
    ) : ItemListaContactosEvent()
    object OnDeleteContacto
        : ItemListaContactosEvent()
}
```

Además, en `ListaContactosViewModel.kt` deberemos gestionarlos en el switch de eventos.

```
fun onItemListaContatoEvent(e: ItemListaContactosEvent) {
    when (e) {
        ...
        is ItemListaContactosEvent.OnCrearContacto -> {
            e.onNavigateCrearContacto()
        }
        is ItemListaContactosEvent.OnEditContacto -> {
            e.onNavigateEditarContacto(contatoSleccionadoState!!.id)
        }
    }
}
```

Vamos ahora a realizar el mismo proceso para la pantalla de `FormContactoScreen.kt`. Para ello definimos en el paquete `.ui.navagation` el archivo `FormContactoRoute.kt`.

Fíjate que en este caso definimos dos funciones de extensión para navegar. Una para editar un contacto existente que recibe el **id** del mismo y otra para crear un nuevo contacto. Además, donde tengo que volver tras editar o crear un contacto, lo paso también en un callback desde la creación del `NavHost`.

```
    private const val FormContactoGraphRoute = "form_contacto"
    private const val FormContactoParameterName = "idContacto"

4   fun NavController.navigateToEditarContacto(
        idContacto: Int,
        navOptions: NavOptions? = null) {
        this.navigate("$FormContactoGraphRoute/$idContacto", navOptions)
    }
    fun NavController.navigateToCrearContacto(
        navOptions: NavOptions? = null) {
        this.navigate("$FormContactoGraphRoute/-1", navOptions)
12  }

    fun NavGraphBuilder.formContactosScreen(
        vm : ContactoViewModel,
16      onNavigateTrasFormContacto: (actualizaContactos : Boolean) -> Unit
    ) {
        composable(
            route = "$FormContactoGraphRoute/{$FormContactoParameterName}",
            arguments = listOf(
                navArgument(FormContactoParameterName) {
                    type = NavType.IntType
                }
            )
        ) { backStackEntry ->
            val idContacto :Int?
            = backStackEntry.arguments?.getInt(FormContactoParameterName, -1)
            if (idContacto != null
                && idContacto != -1
                && vm.contactoState.id != idContacto) {
                vm.setContactoState(idContacto!!)
            }
            FormContactoScreen(
                contactoState = vm.contactoState,
                validacionContactoState = vm.validacionContactoState,
                informacionEstado = vm.informacionEstadoState,
                onContactoEvent = vm::onContactoEvent,
38              onNavigateTrasFormContacto = onNavigateTrasFormContacto
            )
        }
    }
```

Para terminar de definir la navegación, vamos a definir el componente que contiene nuestro `NavHost`. En él, crearemos el `NavController` por ser el elemento más alto en la jerarquía de navegación de nuestra UI y además los `ViewModels`. Para ello, crearemos el fichero `AgendaNavHost.kt` en el paquete `.ui.navagation`. Además, si te fijas pararemos a la definición de las rutas los callbacks de navegación.

```
@Composable
fun AgendaNavHost() {
    val navController = rememberNavController()
    val vmLc = hiltViewModel<ListaContactosViewModel>()
    val vmFc = hiltViewModel<ContactoViewModel>()

    NavHost(
        navController = navController,
        startDestination = ListaContactosGraphRoute
    ) {
        listaContactosScreen(
            vm = vmLc,
            onNavigateCrearContacto = {
                vmFc.clearContactoState()
                navController.navigateToCrearContacto()
            },
            onNavigateEditarContacto = { idContacto ->
                vmFc.clearContactoState()
                navController.navigateToEditarContacto(idContacto)
            }
        )
        formContactosScreen(
            vm = vmFc,
            onNavigateTrasFormContacto = { actualizaContactos ->
                navController.popBackStack()
                if (actualizaContactos) {
                    vmLc.cargaContactos()
                }
            }
        )
    }
}
```