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.

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.

```
@Serializable
object ListaContactosRoute

// 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<ListaContactosRoute> {
        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 como vamos a recibir el id del contacto a editar, definimos un data class con un campo opcional para el id. Además, definimos la función de extensión que nos permitirá navegar al destino con la pantalla de `FormContactoScreen.kt`.

```
1  @Serializable
2  data class FormContactoRoute(val id: Int? = null)

   fun NavGraphBuilder.formContactoDestination(
       vm : ContactoViewModel,
       onNavigateTrasFormContacto: (actualizaContactos : Boolean) -> Unit
   ) {
       composable<FormContactoRoute> { backStackEntry ->
9          vm.setContactoState(backStackEntry.toRoute<FormContactoRoute>().id)

           FormContactoScreen(
               contactoState = vm.contactoState,
               validacionContactoState = vm.validacionContactoState,
               informacionEstado = vm.informacionEstadoState,
               onContactoEvent = vm::onContactoEvent,
               onNavigateTrasFormContacto = onNavigateTrasFormContacto
           )
       }
   }
```

Importante

Para acceder al parámetro de navegación hemos usado `backStackEntry.toRoute<FormContactoRoute>().id`. Pero como comentábamos en los apuntes puede ser que esta composición se realice varias veces con lo que al *'setear'* el id en el VM volveremos a burcar los datos del contacto en la base de datos y perderemos cualquier modificación es por eso que en la función `setContactoState` del `ContactoViewModel` deberemos comprobar si ya tenemos el contacto cargado....

```
    fun setContactoState(idContacto: Int?) {
2       if (idContacto != null && idContacto != contactoState.id) {
            viewModelScope.launch {
                editandoContactoExistenteState = true
                val c: Contacto = contactoRepository.get(idContacto)
                    ?: throw ContactoViewModelException(
                        "El id $idContacto no existe en la base de datos"
                    )
                contactoState = c.toContactoUiState()
                validacionContactoState = validadorContacto.valida(contactoState)
            }
        }
    }
```

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 = ListaContactosRoute
    ) {
        listaContactosDestination(
            vm = vmLc,
            onNavigateCrearContacto = {
                vmFc.clearContactoState()
                navController.navigate(FormContactoRoute())
            },
            onNavigateEditarContacto = { idContacto ->
                vmFc.clearContactoState()
                navController.navigate(FormContactoRoute(idContacto))
            }
        )
        formContactoDestination(
            vm = vmFc,
            onNavigateTrasFormContacto = { actualizaContactos ->
                navController.popBackStack()
                if (actualizaContactos) {
                    vmLc.cargaContactos()
                }
            }
        )
    }
}
```

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**